SPECIFYING REACTIVE SYSTEM BEHAVIOR

THÈSE N^{O} 2588 (2002)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

DANS LE DOMAINE INFORMATIQUE

PAR

Shane SENDALL

B.Sc. with Honours I in Computer Science, University of Queensland, Brisbane, Australie
et de nationalité néo-zélandaise

acceptée sur proposition du jury:

Prof. A. Strohmeier, directeur de thèse
Prof. C. Atkinson, rapporteur
Prof. R. Keller, rapporteur
Prof. A. Wegmann, rapporteur

Lausanne, EPFL
2002

Abstract

Fundamentally, the development of software applications involves dealing with two distinct domains: the real world and software domains; the two converge at the point where a software application is used to make an unsatisfactory real world situation into a satisfactory one. Thus, software application development is a problem solving activity that assumes a problem has been identified and a software application is desired to address this problem.

In this context, it is necessary to take measures that ensure the solution will be both adequate and appropriate with respect to the problem. In particular, it is of utmost importance that the problem in hand and the application's role in helping to solve it are satisfactorily understood by the development team. If this condition is not observed then the application produced is doomed to be inadequate and/or inappropriate, independently of the capabilities of the available technologies and resources, and also independently of other wicked aspects of software development: constantly changing requirements, time-to-market pressures, significant social, political, ethical or economic issues in the project, etc.

The principal objective of this thesis was to improve the state-of-the-art of specifications that are used to communicate to the development team the behavior of the (future) system. In addressing this objective, this work initially involved defining the essential requirements of specifications that could ensure that the development team has a precise, correct and common understanding of the way the system is required to behave.

As a result of analyzing the identified requirements, two general kinds of specifications were distinguished and perceived to be necessary to address the requirements adequately; one that addresses the concerns of the designers, providing a precise description of the system responsibilities; and one that addresses the concerns of the stakeholders in general, providing an informal description of the goals that the stakeholders have against the system. The first specification is referred to as the Behavioral Design Contract and the second one is referred to as the Behavioral Stakeholders Contract.

In this thesis, these two specifications were concretely realized as part of the ANZAC approach. The ANZAC approach defines two work artifacts called the ANZAC use case descriptions and the ANZAC specification, which express the Behavioral Stakeholders Contract and the Behavioral Design Contract, respectively. ANZAC use case descriptions offer an informal and usage-oriented description of the concordant goals that the stakeholders have against the system. An ANZAC specification offers a precise, operational description of the system's responsibilities in servicing all possible requests that it can receive over its lifetime; it uses a restricted subset of the Unified Modeling Language (UML) and its Object Constraint Language (OCL).

In the ANZAC approach, the ANZAC use case descriptions are developed following the ANZAC use case framework. This framework defines the context, purpose, style and form of an ANZAC use case description, and it provides a goal-based approach to use case elicitation. Once a number of ANZAC use case descriptions are established, they can be refined to an ANZAC specification. This refinement procedure is (informally) defined by the ANZAC mapping technique.

An ANZAC specification is developed by the description of three models, which each express a different but complementary view of the system. These three models are called the Concept Model, the Operation Model, and the Protocol Model. The Concept Model defines an abstract system state space in terms of concepts from the problem domain, the Operation Model describes the effect of system operations on the system state, and the Protocol Model defines the correct behavior of the system in terms of its (allowable) input protocol.

As a "proof of concept", this thesis demonstrates the ANZAC approach applied to an elevator control system, which is used to show how ANZAC offers a clean approach for capturing the Behavioral Stakeholders and Design Contract. The elevator case study demonstrates the mapping between the Behavioral Stakeholders Contract and the Behavioral Design Contract using the ANZAC mapping technique. It also highlights the difference in the level of precision and formality that can be found between ANZAC use case descriptions and an ANZAC specification. Furthermore, it demonstrates some of the more advanced features of the ANZAC approach, in particular, its ability to specify performance constraints and concurrent behavior.

Version Abrégée

Fondamentalement, le développement des applications logicielles implique deux domaines distincts: le domaine du monde réel et le domaine du logiciel. Ces derniers convergent là où une application logicielle est employée pour transformer une situation insatisfaisante du monde réel en une situation satisfaisante. Ainsi, le développement d'applications logicielles est une activité visant à résoudre des problèmes de réalisation considérant qu'un problème a été identifié et qu'une application logicielle est désirée pour répondre à ce problème.

Dans ce contexte, il est nécessaire de prendre des mesures qui assurent une solution adéquate et appropriée au problème. En particulier, il est de la plus grande importance que le problème traité et le rôle de l'applications aidant à le résoudre soient compris par l'équipe de développement d'une manière satisfaisante. Si cette condition n'est pas respectée, alors l'application créée est condamnée à être inadéquate et/ou non appropriée, indépendamment des capacités des technologies et des ressources disponibles, et aussi indépendamment des autres aspects ardus du développement de logiciel: le changement constant des besoins, la pression d'une mise sur le marché rapide, les issues sociales, politiques, morales ou économiques du projet, etc.

Ceci dit, cette thèse aborde deux défis qui doivent être relevés dans chaque projet de développement d'application logicielle. Premièrement, trouver un moyen approprié pour justifier et négocier le rôle que l'application jouera dans la résolution du problème identifié. Et finalement, trouver un moyen approprié de communiquer cette connaissance aux membres de l'équipe de développement qui sont chargés de trouver une solution logicielle, sous une forme qui leur est accessible et satisfasse leurs besoins comme concepteurs. Cette thèse rapporte les moyens employés pour faire face au premier défi en tant que « stakeholders contract », ainsi que les moyens employés pour faire face au second défi en tant que « design contract ». Dans ce contexte, cette thèse étudie des techniques et des approches qui peuvent être employées pour spécifier le comportement des systèmes réactifs et qui peuvent être utilisées comme « stakeholders contract » et/ou « design contrat » sur le comportement du système. (Dans la suite, nous nous référerons respectivement au « BSC » et au « BDC »).

Dans la première partie de cette thèse, un ensemble de critères qui définissent certaines qualités et caractéristiques souhaitables pour le « BSC » et le « BDC » sont donnés. En utilisant ces critères, quelques approches courantes s'accordant avec le thème de ce travail sont examinées. L'analyse des diverses approches montre qu'il y a un certain nombre de déficiences dans les approches actuelles pour répondre à ces critères. Ce résultat est employé comme motivation pour établir une nouvelle approche, dont l'objectif est d'améliorer l'état actuel de la question à l'aide de langages et d'outils communément utilisé. Par conséquent, le reste de la thèse est consacré à l'établissement de cette approche, appellée ANZAC.

L'approche ANZAC utilise des langages et des outils courants, mais également beaucoup de techniques bien établies de technologie de programmation, telles que pré- et post-conditions, et des principes, tels que la séparation des intérêts. Elle répond aussi bien au « BSC » qu'au « BDC » pour ce qui est de l'importance de les traiter les deux de manière unifiée, suggérant une approche simple. Cependant, en raison des intérêts distincts liés à chacun des deux contrats, deux objets de travail séparés sont employés par ANZAC (un pour chaque contrat). ANZAC définit un procédé systématique, appelé la technique de corrélation ANZAC, pour obtenir le « BDC » à partir du « BSC ». Ceci favorise le tracabilité entre les deux contrats et établit une corrélation informelle entre eux.

L'aspect d'ANZAC décrivant le « BSC » est défini par les « use cases ». Le cadre proposé est une spécialisation (et clarification) des travaux sur les « use cases ». Le modèle d'ANZAC décrivant le « BDC » est défini en utilisant un sous-ensemble restreint du « Unified Modeling Language » (UML) et de son « Object Constraint Language » (OCL). Cette spécification ANZAC décrit le comportement du système respectif en utilisant un modèle qui offre trois vues complémentaires. Chaque vue capture un aspect distinct du système: état, opération et activité.

La dernière partie de cette thèse montre ANZAC appliqué à une étude de cas d'un système de contrôle d'ascenseur. Elle permet de valider l'approche et montre à quel point ANZAC offre une approche propre, concise et précise pour capturer les « BSC » et les « BDC ». L'étude de cas de l'ascenseur démontre la corrélation entre les « BSC » et les « BDC » utilisant la technique de corrélation ANZAC. Elle met également en évidence la différence au niveau de la précision qui peut exister entre les deux contrats. En outre, elle démontre certaines des caractéristiques les plus avancées d'ANZAC, en particulier sa capacité à décrire des contraintes de performance et les comportements concurrents.

Acknowledgements

There are a number of people that have contributed in various ways to my work over the years. I would like to express my gratitude to them.

First of all, I would like to thank Alfred Strohmeier for supervising my work and for introducing me to the work area addressed by this thesis. In particular, he motivated me to look into specifying concurrency and synchronization requirements as part of software inception activities, which I now believe to be a very interesting (and surprisingly under-explored) area of research. He also encouraged me and made it possible to go to numerous international conferences. Furthermore, his comments greatly improved the quality of the work that is presented in this document. In particular, the Operation Schema notation proposed in this thesis was much more homogeneous thanks to his vast experience in (programming) language design. I would also like to thank him for much "fatherly" advice that he has given to me over the years.

Second, I would like to thank all the jury members Colin Atkinson, Rudolf Keller, and Alain Wegmann for having taken the time and effort to review my thesis and to serve on my examination board. Their comments have improved the quality of the final manuscript.

Third, I am greatly indebted to Jörg Kienzle. His review of this thesis not only removed many typographical and grammatical errors, but also his conscientious comments were extremely constructive and have improved the quality of this thesis. He was also very kind in providing me with his thesis template, which certainly saved me some initial "start-up" effort, and I would also like to thank him for some useful comments on my defense presentation. Also, I would like to thank my "roomie" and the "next-in-line" Mohamed Kandé for making my time at the lab more fun and pleasant and for providing a spring-board for bouncing ideas and problems off. I would also like to thank him for some good comments on the introduction of this document and on my defense presentation. Also, I would like to thank the many students that I had the pleasure to work closely with Marcos Perez Jurado, Slavisa Markovic, Ankur Gupta, Mahim Mishra, Milena Ilic, Rida Hamdan, and Marco Bonetti. I would like to thank Didier Buchs for helping me with the French abstract and some sticky areas of semantics in Operation Schemas. And, I would like to thank the other members of the Software Engineering Laboratory (LGL), Anne Crettaz Schlageter, Benjamin Barras, Raul Silaghi, David Hürzeler, Stanislav Chachkov, Adel Besrour, Xavier Caron, Sandro Costa, Rodrigo Garcia-Garcia, and Juan-Miguel Gomez for an enjoyable atmosphere at the lab. I would also like to thank past members who made quite some effort to welcome me and who assisted me in early times Thomas Wolf, Stéphane Barbey, Enzo Grigio, Nicolas Guelfi, Mathieu Buffo, Cécile Peraire, Giovanna Di Marzo and Gabriel Eckert. Also, I would like to thank the EPFL computer science third year students of the past three years, they taught me many things about making Operation Schemas usable and learnable.

Fourth, J'aimerais remercier mes beau-parents suisses Gérald et Marie-Françoise Müller pour leur soutien inconditionnel et pour m'avior accepté instantanément comme membre de leur famille depuis Nöel 1997.

Fifth, I would like to thank my parents Wayne and Wendy and my sister Debi. They have been a constant support to me right from when my ideas were not oriented towards computers. They have themselves made many sacrifices for my sake, and they accepted without question my crazy plans of doing my doctorate on the other side of the world in Switzerland. In particular, I am sure that I would have not taken this task on without the support and encouragement of my father. Many a time I reminisce on the good ol' days. It is probably best put in the words of (the late) Douglas Adams: "*In those days, spirits were brave, the stakes were high, men were REAL men, women were REAL women, and small furry creatures from Alpha Centauri were REAL small furry creatures from Alpha Centauri.*"

Sixth, I would also like to thank my wife Fabienne. She has been a tremendous support both physically and mentally, and she has kept me company through the good and not so good days. She has always been there for me and she has made my world that much more happier. I look forward to spending the rest of my days with her hand-in-hand.

Finally, how can I begin to thank the big J for what he did a couple of thousand years ago and continues to do today. He gives me endless inspiration and he never ceases to amaze me with his love. What can I say other than Hallelujah!

Table of Contents

List of Figures

Chapter 1: Introduction

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements."
Frederick P. Brooks, Jr. [Bro95]

1.1 Setting the Scene

Nowadays, software-intensive systems play a role in almost all areas of our life, professional and private—a coverage that continues to expand as new possibilities for applying software solutions are explored and technologies for interoperation of systems are improved. Software applications have become the primary interface to services provided by vast networks of systems. However, in spite of our reliance and inevitable dependence on software, it too often falls short of the purpose for which it was intended or functions in unexpected ways [Pre99].

Fundamentally, the development of software applications involves dealing with two distinct domains: the real world and software domains[1]; the two converge at the point where a software application is used to make an unsatisfactory real world situation into a satisfactory one. Thus, software application development is a problem solving activity that assumes a problem has been identified and a software application is desired to address this problem. This activity is a very difficult task to master in general, where Brook's four properties of

1. referred to as the machine and the world by Jackson [Jack95]

the “irreducible essence of modern software systems”: complexity, conformity, changeability and invisibility [Bro87], highlight some of the variables at play in this task.

In this context, it is necessary to take measures that ensure the solution will be both adequate and appropriate with respect to the problem. In particular, it is of utmost importance that the problem in hand and the application’s role in helping to solve it are satisfactorily understood by the development team. If this condition is not observed then the application produced is doomed to be inadequate and/or inappropriate, independently of the capabilities of the available technologies and resources, and also independently of other wicked aspects of software development, such as, constantly changing requirements, time-to-market pressures, significant social, political, ethical or economic issues in the project, etc.

In other words, a principal challenge that must be faced in each and every software project is to ensure that the development team has a precise, complete, correct and common understanding of what they should build. This challenge is the theme of this thesis.

1.2 General Approach and Focus

The approach that I take in this thesis is based upon the assumption that the information about what system should be built is formally communicated to the development team as a specification. The reason for this assumption is that a specification allows every member of the development team to have a common statement of what to build and it is a resource that can be revisited when necessary.

The approach of this thesis refers to the above mentioned specification as the *Design Contract*—since it represents the contract on design correctness that must be observed by the development team. I now introduce a number of characteristics and qualities that are desirable for a Design Contract to possess.

Firstly, a Design Contract should explicitly describe the interface between the system and its environment, otherwise it would be unsure exactly which responsibilities are those of the system versus those of the environment (and there would be no way to decide whether the system has been overspecified by the Design Contract). This is particularly important for systems that have strong constraints imposed on them by their environment—which is often the case for reactive systems. Also, the system is evaluated with respect to its usage in its environment. Therefore, a Design Contract that clearly defines the interface between the system and its environment would provide a basis for validating the system’s role in that environment.

Secondly, the desire to build an appropriate system suggests that it is not only essential to construct a specification for designers to follow, but also to provide a justification of that specification [Smi96]. A justification for a Design Contract entails making sure that each constraint that it places on the design of the system is aligned with and traceable to the (system-related) goals of the various parties that have a vested interest in the software application. These parties are commonly referred to as the *stakeholders* of the system.

Lastly, and probably most obviously, the Design Contract should be given in a form that is accessible to the developers and it should provide all the information needed to design the system and no more, hence no under- or over-specification [Par72, McD94]. In other words, the Design Contract should facilitate the job of designers. An overspecified Design Contract implies that it expresses premature design decisions[1], which would mean it would need to be modified as these decisions change—an undesirable situation. Whereas, an underspecified Design Contract means that the designers do not have sufficient information to be autonomous, degrading its usefulness as the contract on design. Thus, a Design Contract that provides strictly only the essential information permits designers to produce creative and innovative solutions that may not have been initially envisaged, but that still meet the requirements of the stakeholders.

In review of these desired characteristics of a Design Contract, the second one suggests that a Design Contract can not be constructed without considering the goals of the stakeholders for the system. It is unlikely however that the Design Contract should explicitly describe the stakeholders' goals for the system, because much of the contextual information that is necessary to describe a goal is at least superfluous from the perspective of the designers. In some cases, such information would cause confusion during design, because it describes situations that cannot be detected or acted upon by the system. And, such information can make it difficult for designers to know whether the system should or should not provide a means to deal with the situation. Therefore, explicitly describing the stakeholders' goals would conflict in general with the objective for a "designer-friendly" Design Contract, which is the third (desired) characteristic.

This apparent divergence between the second and third characteristics reveals two disparate requirements on the Design Contract: provide a valid description of the stakeholders' goals against the system, and provide a precise and focused description of the system to design. The approach taken in this work to address these two different requirements involved defining another specification that explicitly captures the concordant goals that the stakeholders have against the system. I refer to this specification as the *Stakeholders Contract*, since it signifies a contractual agreement[2] between stakeholders.

By addressing the requirements (driven by the desired characteristics) with two different specifications, it is necessary to ensure that they can be nevertheless coherently combined to fulfill the overall objective of communicating the system to build to the development team. Such a relationship suggests an explicit bridge between them. Further motivation for an explicit relationship between the two specification comes from the desire

1. "For every complex problem, there is a solution that is simple, neat, and wrong." -- H. L. Mencken
2. In this context, the term 'contract' can mean anything from a very informal agreement to a full-blown formal agreement that is bound by law.

to ensure that the Design Contract is justified with respect to the interests of stakeholders (derived from the second characteristic). In this respect, the Stakeholders Contract is used to establish the Design Contract, where the Stakeholders Contract is refined into the Design Contract and traceability is set up between the stakeholders' goals described by the Stakeholders Contract and the system responsibilities described by the Design Contract.

Rather than aiming at facilitating the job of the designers, like the Design Contract, the aim of the Stakeholders Contract is to facilitate the job of stakeholders in understanding, negotiating and agreeing upon the (software) system requirements. By focusing on goals, one can have a greater confidence that the Stakeholders Contract adequately and appropriately captures the requirements. Ideally, the Stakeholders Contract should be in a form that is accessible to all stakeholders, both technical and non-technical alike, which means that no assumptions can be made about the (technical) background of stakeholders: it must allow for the "lowest common denominator". Also, the Stakeholders Contract should allow one to capture the subtleties and complexities of the informal real world, since this activity considers the real world domain and not only the software domain.

Classically speaking, the Stakeholders and Design Contracts can be thought of as part of the *Software Requirements Specification* [TD97] for a project. However, a Software Requirements Specification generally includes additional information, e.g., project constraints, project drivers, assumptions, (social, political, ethical, economic and technical) issues, etc.

The focus of this thesis is on the Stakeholders and Design Contracts. In addition to establishing a suitable formalism for the Stakeholders and Design Contracts, the approach taken in this thesis determines a bridge between the two specifications, which ensures that refinement and interplay is possible between them.

1.3 Influences

The development of an application can be logically viewed as being made up of many steps taking one from vision to realization; this is essentially the process of stepwise refinement [Wir71]. However, as is well-documented (e.g. [WK00, Pre97]), there is an interplay between the different steps that take one from concept to implementation. The concern for feedback, incremental delivery, and parallel work during the development process, originally highlighted by Boehm's spiral model [Boe86], suggests that the development process as a whole is not simply a one-way refinement.

One factor that suggests the activity of defining the Stakeholders and Design Contracts is intertwined with other software development activities is problem complexity. At the heart of this complexity is: size of the problem, complexity of the system's environment, and complexity in communication among the stakeholders. In what follows, I refer to the Stakeholders and Design Contracts as "the specifications" to be concise.

- *Size of the problem*: The size of the problem that is to be addressed by the system affects the way and ease with which a team gathers, analyzes and manages information about it. In general, the larger the amount of information, the more difficult it is to keep the specifications consistent, concise and up-to-date, and the more likely that they will evolve along with the project as issues are clarified en route.
- *Complexity of the system's environment*: A system is often constrained to interact with its environment in a certain way and within certain rules and expectations. Due to the possible complexity of these constraints, it may be necessary, and even more productive, to allow the specifications to evolve as these constraints are better understood.
- *Complexity in communication among the stakeholders*: Software systems may have a large number of stakeholders, who can have different backgrounds, roles, domain knowledge, technical competencies, and agendas. As a consequence, stakeholders will often have different objectives and perceptions of the problem domain that lead to conflicts of interests between them. In such cases, it may not be possible to resolve all conflicts upfront. It may be necessary to instead experiment with different realizations to convince stakeholders one way or the other, requiring the specifications to be updated as the conflicts are resolved.

Project dynamism is another factor that can drive an iterative and incremental development of the two specifications. At the heart of this factor is: requirements instability, time-to-market pressures, and uncertainty of which system to build.

- *Requirements instability*: in addition to changes made to requirements for technical reasons, the requirements may evolve over time due to changes in the client's perceived needs or changes to the constraints on the project that are driven by external influences. This means that as requirements are changed, the specifications will need to be brought up-to-date.
- *Time-to-market pressures*: competition in the business world can mean that the window of opportunity is all important. In such cases, a project can be given a short period of time within which to produce the software. Consequently, only features that make the current release are specified before design and implementation activities are undertaken [BLP+01]. In such cases, the specifications will only provide a description of partial system capability.
- *Uncertainty in which system to build*: the details of all the requirements may not be known or well understood initially, e.g., the product may be used to explore a new market. In such cases, it may be desirable to allow the specifications to evolve as the problem is better understood, where the choices made during design and implementation affects what can be judged to be a satisfactory future situation for the system (and for the endeavor of the stakeholders).

In consideration of the above points, it would seem necessary to ensure that the Stakeholders and Design Contracts are malleable to change, facilitate the synchronization of changes between them, and offer support for incremental description.

1.4 Scope

As the title implies, this work is about “Specifying Reactive System Behavior”. Hence, it is necessary to narrow the scope on the context given in the previous sections, which lies somewhere within the broad area of “Requirements & Analysis” (using the terms of the classical development lifecycle model[1]).

- Reactive Systems

  I consider only software systems that are reactive in nature. This means that the system interacts with its environment over time in an organized manner, where stimuli arrive in an endless and perhaps unexpected order [HP85]. The correct treatment of these stimuli means that a reactive system must “know” whether it is in an appropriate situation to serve the requests and notifications that are implied by the stimuli.

  In terms of classification, reactive systems exclude data-intensive systems, such as, data management systems, and they exclude transformational systems, where all input is given at one time or as a continuous stream, e.g., compilers, batch processes, media stream filters, numerical mathematics applications, etc. A large percentage of distributed systems, and particularly internet-based systems, are reactive in nature. This covers many business systems, since they need to be able to react to requests in a timely manner and in an endless cycle.

  Reactive systems often must deal with an inherently concurrent environment, i.e., they must deal with stimuli from different sources at the same time, and they often have timeliness requirements (e.g., real-time systems are subject to strong performance constraints). Due to its interactive nature, a reactive system is bound to its environment (e.g., embedded systems are ruled by strong hardware constraints).

- Behavioral Specification

  I focus on specifications that capture at most the behavioral aspects of the Stakeholders and Design Contracts. This means that I investigate only specifications that involve functional requirements and some extra-functional requirements, such as, timing constraints and concurrency. As I do not address the complete Stakeholders and Design Contracts, I refer to the aspect addressed in this thesis as the *Behavioral Stakeholders Contract* and the *Behavioral Design Contract*, respectively. Having said that, the “Behavioral” label does not tell the full story: I do not address all kinds of extra-functional requirements that constrain the behavior of the system, such as, security requirements, usability requirements, etc. [RR00]

1. Requirements & Analysis, Architecture & Design, Implementation, and Testing.

- Mainstream Languages and Tools

  One of the main objectives of this thesis was to ensure that all proposals made could be realized using mainstream languages and tools. The reason is simple: this work is concerned with software development, and therefore the link to software development practice should be clear.

  The techniques investigated in this thesis are principally model-based. The reason for a bias towards model-based languages is because models naturally support abstraction, projection, and decomposition. And I believe that in trying to understand what needs to be built, these three principles are crucial for managing the inherent complexity of modern software systems.

  Considering the current status of the Unified Modeling Language (UML) as the "lingua franca" of software modeling, I chose UML as the target language for this work, along with its constraint language, called the Object Constraint Language (OCL).

1.5 Contributions

A number of contributions have been made by this thesis toward the theory and practice of specifying the behavior of reactive systems in the earlier stated context. The main contributions of this thesis are the following:

- A study of the issues involved in specifying the behavior of reactive systems.

  I present an analysis of the issues involved in specifying the behavior of reactive systems for the purpose of communicating this information to the development team.

- The ANZAC approach.

  The ANZAC approach provides a means to develop the Behavioral Stakeholders and Design Contracts for reactive systems. Each contract is defined by a separate work artifact, which are referred to as the ANZAC use case descriptions and the ANZAC specification, respectively. The ANZAC approach defines a systematic procedure for deriving the ANZAC specification from the ANZAC use case descriptions, called the ANZAC mapping technique, which promotes traceability between the two contracts and also establishes a bridge between them.

- A clarification of various aspects in UML.

  Descriptive style: UML is principally a language for modeling software designs and implementations, hence its notations mainly have a prescriptive flavor. In this work, it is shown (in addition to use cases) how UML can be used in a descriptive, yet rigorous manner. As a consequence, this work offers another dimension to UML users for modeling software systems.

  Use Cases: In this work, I introduce the idea of a use case framework, within which the development of use cases targeted for a particular context and purpose can be achieved. Due to UML's high profile in software development practice, it would arguably be very beneficial if UML were to clarify its stance on use cases by defining a set of best practice use case frameworks.

Object Constraint Language: In using OCL to describe pre- and postconditions for operations, there is a considerable amount of issues that are not addressed by the language. In this work, I clarify a number of those issues, such as, frame assumptions, creation and destruction of objects, declarations, structuring pre- and postconditions, and message sending.

State Machines: Several limitations of UML state machines for modeling the interaction protocol of concurrent systems are highlighted and addressed by this work. In particular, a clear separation between the description of orthogonal and concurrent behavior using state machines is proposed, and the notions of auto-concurrent state, auto-orthogonal state, actor-activity-state, view-state, and synchronization transitions were introduced to better describe reactive system behavior. Also, a way to specify timing constraints using state machines with a number of small additions to UML is proposed.

- An enhancement of the current state-of-the-art for goal-based use case descriptions.

  Extending the work of Cockburn on goal-based use case descriptions, I clarify a number of issues in capturing system-usage goals with use cases. For instance, I clarify how a use case can still be related to a goal even if is not triggered by an external actor.

- An enhancement of the current state-of-the-art for specifying operations with pre- and postconditions.

  Concurrency: A number of contributions are made for specifying concurrent operations (i.e., operations that are subject to execution periods that may overlap in time with those of other operations). These include: a study and proposal of what safety and liveness properties should be enforced by the correctness rule for pre- and postconditions; a reinterpretation of postconditions for concurrent operations; and the introduction of a number of language constructs that allow one to capture synchronization requirements related to the access of shared resources.

  Incremental Descriptions: It is shown that the traditional case-analysis style for writing pre- and postconditions is prone to “case explosion”. An incremental style of description is proposed, which is not prone to “case explosion”, and arguably offers a more intuitive approach. Also, a general principle is established that defines the meaning of an incremental style in postconditions.

  Asynchronous & Synchronous Messaging: It is shown how postconditions can be used to assert that messages are output with the execution of the operation. The sending of asynchronous and synchronous messages, and the throwing and catching of exceptions are addressed.

- Guidelines and techniques for systematically describing concurrent system behavior.

  A number of guidelines and techniques are introduced by this work for the purposes of describing concurrent system behavior. Some of these include: the inherent concurrency guideline, the divide-by-actor technique, and the divide-by-collaboration technique.

1.6 Plan

The central element of this thesis is the development of the ANZAC approach, which offers a systematic way to capture the Behavioral Stakeholders Contract and the Behavioral Design Contract for reactive systems. As a consequence, six of the ten chapters of this thesis describe the ANZAC approach.

Chapter 2: *Key Concepts and Related Work*
This chapter defines some key concepts for this work and provides an overview of related work. The description of related work covers some approaches and techniques that are closely linked to the broad area of "Requirements and Analysis" and have particular relevance to the ANZAC approach. This chapter provides motivation for the ANZAC approach, and it sets the scene for the description of the ANZAC approach.

Chapter 3: *Specifying the Behavioral Stakeholders Contract*
This chapter introduces the part of the ANZAC approach that involves defining the Behavioral Stakeholders Contract, called the ANZAC use case framework. It motivates the choice of description, details how the ANZAC use case framework builds upon related work, and provides a definition for an ANZAC use case. It also details the elements, style, and format of ANZAC use case descriptions, explains the kinds of requirements that can be captured with them, and explains how other kinds of requirements can be related to them.

Chapter 4: *Specifying the Behavioral Design Contract*
This chapter introduces the part of the ANZAC approach that involves defining the Behavioral Design Contract, called the ANZAC specification. It provides an overview of the three different models of the system that form an ANZAC specification: Concept Model, Operation Model and Protocol Model, and it discusses their relationship to ANZAC use case descriptions.

Chapter 5: *Concept Model*
This chapter presents the Concept Model. It describes the purpose of the Concept Model, provides some guidelines for constructing Concept Models, explains the notation used to represent them, discusses ways to define additional constraints to capture system invariants, and provides an example of a Concept Model for a system.

Chapter 6: *Operation Model*
This chapter presents the Operation Model. An Operation Model is described by a collection of Operation Schemas, where each Operation Schema describes a system operation by pre- and postconditions. This chapter focuses on explaining and detailing Operation Schemas. It proposes a language and form for Operation Schemas; intro-

duces a number of constructs, rules, assumptions, and interpretations that apply to them; describes how they can be used to express that messages are sent to actors during the execution of the operation; and proposes some enhancements to them for the purpose of specifying the behavior of concurrent operations.

Chapter 7: *Protocol Model*
This chapter presents the Protocol Model. It describes the various concepts that are used to express a Protocol Model, explains the way those concepts are represented in a Protocol Model, proposes some guidelines that may assist specifiers of Protocol Models, and addresses the issue of modelling concurrent system behavior. Also, it proposes a possible way to integrate time-based properties in Protocol Models for the purpose of specifying timing constraints.

Chapter 8: *ANZAC Mapping Technique*
This chapter presents the ANZAC mapping technique, which provides a systematic procedure for going from ANZAC use case descriptions to an ANZAC specification for a system.

Chapter 9: *Elevator Case Study*
This chapter applies the ANZAC approach to an elevator control system. It takes the reader progressively through the development of the system using the ANZAC approach, offering a full treatment of it. In particular, it presents the ANZAC use case descriptions and the ANZAC specification for the elevator control system, and it illustrates the steps taken to map the first to the second one using the ANZAC mapping technique. The concurrent behavior and timing requirements of the elevator control system are also addressed.

Chapter 10: *Conclusion*
This chapter restates the main objective of this thesis, gives an overview of the general approach taken in this thesis, and summarizes the solution proposed by this thesis. It provides a critical review of the proposed solution, and it describes some areas of further research.

This document also contains three appendices: Appendix A provides a summary of the UML's constraint language called the Object Constraint Language (OCL), Appendix B describes tool support for ANZAC specifications, and Appendix C contains the declarations and definitions of Parameterized Predicates and Functions that are used in the ANZAC specification for the elevator control system (Chapter 9).

Chapter 2:
Key Concepts and Related Work

This chapter defines some key concepts for this work and provides an overview of related work. The description of related work covers some approaches and techniques that are closely linked to the broad area of "Requirements and Analysis" and have particular relevance to the ANZAC approach. This chapter provides motivation for the ANZAC approach, and it sets the scene for the description of the ANZAC approach, which comes in the following chapters.

2.1 Key Concepts

In this section, I provide an explanation of some of the concepts that reoccur throughout this document and that are particularly important to this work.

2.1.1 Abstraction, Decomposition, Projection and Precision

For coping with the description of large and complex systems, the principles of abstraction, decomposition, and projection prove to be extremely useful tools. The principle of abstraction ensures that the description has only those features that are deemed pertinent to a particular purpose, being a one-to-many map to the concepts in the domain of discourse. According to Jackson [Jack00], abstraction involves "Taking away detail considered unnecessary for the purpose in hand". This removal of unnecessary detail facilitates simple and general descriptions and reduces the amount of complexity that one needs to deal with at a given moment during the development lifecycle. Abstraction lends itself to encapsulation, classification, and generalization. Encapsulation establishes an opaque barrier around those details that are irrelevant to the abstraction. Classification groups entities with a similar role as members of a single class (or kind). Generalization expresses similarities between different concepts.

The principle of decomposition ensures that the properties of the whole system follow from the properties of its parts, which lends itself nicely to recursion. Decomposition is the main technique for constructing hierarchical descriptions in a top-down fashion. The principle of projection separates different aspects of the system into multiple viewpoints, where the partition is assumed to be reasonably orthogonal [Nus94]. Each viewpoint provides a theme on top of which abstraction and decomposition can be applied.

In communicating the important aspects of the problem to designers, it is important that these three principles be used in conjunction with precision. A precise description is one that expresses the problem in focus in a clear, exact and unambiguous way, while still keeping within the bounds of the abstraction. One of the most attractive characteristics of precision is that it forces the specifier to make explicit many underlying assumptions that may not have otherwise been stated. With respect to the purposes of the Stakeholders and Design Contracts, the more precise that a specification is, the more easy it would be to contest its validity and to demonstrate that it has been fulfilled by the realized system.

2.1.2 Inherent Concurrency

In defining the Design Contract for a system, it is important to avoid overspecifying the problem by introducing design decisions that are not requirements. However, it important, on the other hand, that the designers are still given information about the system that helps them understand the inherent nature of the system in its environment. To this end, it would be useful to provide designers with information about any inherent concurrency that a system may exhibit. A system is inherently concurrent if it must handle activities that can happen simultaneously in the external world [Bac98]. This means that an inherently concurrent system interacts with external entities (in its environment) that can make requests on it independently of other external entities. Appropriate information about the inherent concurrency of a system may include: which resources are shared by concurrent operations, and the inherent synchronization constraints between concurrent operations.

2.1.3 Correctness

The correctness of a system is often gauged from its ability to satisfy certain safety and liveness properties—terms that were coined by Owicki and Lamport [OL82]. According to Andrews [And00], a *safety property* is “A property of a program that asserts that nothing bad will ever happen—namely, that the program never enters a bad state”, and a *liveness property* is “A property of a program that asserts that something good will eventually happen—namely, that the program eventually reaches a good state”.

For a system that executes operations in sequence (i.e., one after the other), the safety and liveness properties correspond to assertions on operations requiring correct results and termination. The first one is a safety property, because it requires that the operation does nothing “bad”. And, the second one is a liveness property, because operation termination indicates progress (for a sequential system).

For concurrent systems, safety and liveness properties can be more elaborate, because concurrent behavior introduces a number of new problems: the possibility of interference between concurrent activities, deadlock between two or more concurrent activities, race conditions, and starvation of an activity[1].

Interference is a result of a lack of synchronization between concurrent activities. For example, two operations that update a resource at the same time may result in resource corruption—a form of interference. To avoid any implementations that could corrupt resources by interference, a safety property could be defined to assert that at most one activity is allowed to update a resource at any one time.

Deadlock occurs when two or more concurrent activities are all waiting on access to a resource that the other has (current and) exclusive rights to. Absence of deadlock is an important safety property for concurrent systems. Note that a consequence of the system being in a deadlocked state is that none of the deadlocked activities can proceed, which would mean that any related liveness property would also be unfulfilled.

No activity starvation means that activities do not wait forever to be serviced. A liveness property therefore could state that the activity makes progress by asserting that the system will eventually reach a certain state from a given one. Most liveness properties of concurrent systems depend on fairness, which is concerned with guaranteeing that activities get the chance to make progress even in the presence of synchronization dependencies between activities. Note that operation termination alone is not sufficiently strong to guarantee progress in a concurrent system. The level of fairness is generally determined by the scheduling policy of the runtime environment.

Traditionally, a scheduling policy is classed as either: unconditionally fair, weakly fair, or strongly fair [And00]. Intuitively, a scheduling policy is *unconditionally fair* if all activities are eventually allowed access to an unguarded shared resource, where unguarded means that an activity does not wait on a condition for it to proceed. It is *weakly fair* if it is unconditionally fair and all activities are eventually allowed access to a guarded shared resource, where the guard becomes true and stays true until it has been evaluated by the activity. It is *strongly fair* if it is unconditionally fair and all activities are eventually allowed access to a guarded shared resource, where the guard does not stay false.

In practice, the design choices for ensuring liveness properties of a system may conflict with the safety properties of the system and vice versa. Therefore in some cases, it may be necessary to make less assertions about liveness to ensure safety.

1. Safety and Liveness properties are nicely classified and explained in [NC00].

2.2 Related Work

This thesis work can be related to many different approaches and techniques that touch the broad area of “Requirements and Analysis”. However, this thesis has a particular similarity and relation to the following work[1]:

- Unified Modeling Language
- Fusion
- Use Cases
- Catalysis
- Pre- and Postconditions
- Model-Based Formal Specifications
- Rely- and Guarantee-Conditions

In what follows, I present the above mentioned work, and I highlight how each one is related to the ANZAC approach.

2.2.1 The Unified Modeling Language

Models are a useful tool for describing information, because one can take advantage of the power of abstraction to concentrate on those aspects deemed relevant to the specifier. However, they have the disadvantage of being a point of confusion for those that are uncomfortable with the representation used. Therefore, the familiarity of the notation and the way that it is represented is important for ease of understanding. Furthermore, it is important to note that graphical representations of models have proved popular for those contexts where there are perceived cognitive gains in using a graphical description compared to a textual one.

Approaches based on graphical models of software were popularized in software development circles firstly by structured software development methods in the 70’s and early 80’s and since then by object-oriented software development methods [Dav90, Wie98, EG00]. In the 90’s, the number of object-oriented modeling approaches continued to grow, each one proposing a different notation. Over the last few years, the software development industry has gone through the process of standardizing visual modeling notations. The Unified Modeling Language (UML) is the product of this effort; it unifies the scores of notations that currently exist in the industry.

UML was an initiative of the three most prominent (object-oriented) methodologists, Booch, Jacobson, and Rumbaugh [BRJ98, RJB98, Fow99b]. The language has since gained significant industry support from various organizations, via the UML Partners Consortium, and has been submitted to and approved by the Object Management Group (OMG) as a standard (November, 1997). Since then, the majority of software modeling techniques and

1. Note that there are literally scores of object-oriented methods and formal specification languages that have indirect relevance to this work.

approaches proposed use UML notations. The effect of this standard notation not only means a greater possibility for tool compatibility, but it also unites research on improving expressiveness and precision of a single language. Using UML notations has many benefits, including the following ones:

- UML offers a common language, uniting different development methods in terms of notation and vocabulary, and allowing tool interoperability between different vendors.
- UML provides a rich set of notations that can be used to describe many different aspects of the software under development. It offers seven different types of diagrams in all: activity, implementation, interaction, sequence, state, static structure, and use case diagrams.
- UML provides extension mechanisms that can be selectively applied to UML model elements. These extension mechanisms facilitate methodologists to define extensions to UML for a particular (and warranted) purpose. For instance, a number of initiatives have defined domain-specific extensions of UML by making use of the extension mechanisms.
- UML contains a textual constraint language called the Object Constraint Language (OCL) [WK98]. OCL is a formal language for writing expressions whose principles are based on set theory and first-order predicate logic. OCL can be used in various ways to add precision to UML models beyond the capabilities of the graphical diagrams. Two common uses of OCL are the definition of system invariants, and the definition of pre- and postconditions for operations.
- UML provides a semantic base in the form of a metamodel that defines well-formed models, and the relationships between model elements.

In considering UML, it is important to recognize that it is only a language: it is process independent and therefore does not prescribe how its notations should be used.

Due to its heritage in object-oriented methods, UML has a bias towards notations that are prescriptive rather than descriptive in nature—with use case diagrams being the only obvious exception. As a consequence, UML is principally a language for modeling software designs and implementations with few explicit options for describing the real world domain of a problem (other than a use case diagram).

Through the ANZAC approach, this thesis illustrates how UML can be used in a descriptive, yet rigorous manner. In doing so, it promotes the sound practices of black-box descriptions in UML. Furthermore, ANZAC offers a novel way of addressing concurrency and timing constraints using UML. In particular, the ANZAC specification shows how one can describe concurrent system behavior in terms of correctness constraints without the need to commit to a particular concurrency control mechanism (since in UML, concurrency is described at the operation-level by indicating whether an operation is either: sequential, guarded, or concurrent). Also, it shows how timing constraints can be defined using the combination of UML state machines and OCL. As a result, this thesis offers another dimension to UML users for modeling software systems.

2.2.2 Fusion

The second-generation object-oriented development method Fusion [CAB+94] offers a step-by-step process that involves the three classic phases of the waterfall model: analysis, design and implementation. It has principal origins in first-generation methods OMT (Object Modeling Technique) [RBP+91] and Booch [Boo94]. A mapping from Fusion notations to UML notations has been proposed by Atkinson [Atk97].

One of the most appealing aspects of the Fusion method is its clear designation of the system boundary, which facilitates a clean separation between the description of the system's responsibilities in servicing the requests of its environment (analysis) and the description of the internal collaborations between objects that realize those responsibilities (design). In what follows, I concentrate on the analysis phase because it is of direct relevance to this work.

The analysis phase of Fusion abstracts from the details of the internal workings of the system, treating the system as a black-box. It places a particular emphasis on defining the system interface and the information that is relevant to the system for the purposes of fulfilling requests from its environment. These two aspects of the system are described by the object model and the interface model.

The *object model* defines the static structure of the information in the system. The object model is later refined into a system object model, which makes a separation between the (object) classes that lie within the system boundary and those ones that lie outside it. Neither the classes inside the object model nor those ones inside the system object model have methods (i.e., responsibilities) assigned to them, because "analysis" classes describe concepts of the problem domain rather than software components. The object model is very similar to the one proposed by OMT, which was heavily influenced by entity-relationship modelling techniques and graphical notations [Che76].

According to [CJ93], the *interface model*, which consists of the operation model and life-cycle model, defines the input and output communication of the system. All communication between the system and its environment is by events. The communication is asynchronous, which means the sender does not wait for the event to be received. Events that are incoming to the system can cause the system to change its state and to output events to its environment. An input event and the corresponding effect are called a system operation. Note that in the analysis phase, Fusion assumes that all system operations occur instantaneously, i.e., the event is communicated from the sender to the system and the corresponding operation is executed in zero-time.

The *operation model* declaratively specifies the behavior of the system operations by defining their effect in terms of state changes to the system and events that are output to the environment. The operation model is expressed as a series of schemata, where there must be at least one schema for each system operation. According to the authors, the form and usage of a schema was inspired by the formal specification languages Z and VDM (which are discussed later). A schema is written in (structured) natural language, and it defines the pre- and postconditions of a system operation. The pre- and postconditions describe the effect that the operation has on the state of the system (as described by the system object model), and the events that are output. By expressing event sending as part of the postcondition, it is possible to clearly state under which circumstances events are output.

According to [CJ93], the *life-cycle model* describes the behavior of the system from the perspective of how the system communicates with its environment over its lifetime. If the system receives an event that is not allowed according to the life-cycle model, then the system simply rejects it, leaving the state of the system unchanged. The life-cycle model is written using a regular expression grammar, which is similar to the graphical one proposed by the JSD method (Jackson System Development) [Jack83]. Scenarios of system usage can also be used to help construct the life-cycle model.

The object, operation and life-cycle models complement each other nicely without significant overlap[1], and hence, they exhibit a clean separation of concerns. Also, Fusion provides a useful set of guidelines for cross-checking the consistency between the models, which is important because the system is described by the combination of the three models.

The ANZAC specification has a number of similarities to Fusion's analysis models. In particular, an ANZAC specification makes use of the idea of non-overlapping descriptions that each express a different but complementary aspect of the system. Furthermore, it defines three models: Protocol, Concept and Operation Model, which follow more or less the separation of concerns exhibited by Fusion analysis models (life-cycle, object, operation model, respectively). There are however differences between the two in terms of the formalisms used, role in development, and types of systems targeted, which I now explain.

An ANZAC specification uses UML notations. Apart from the clear advantage of tool support for the standard, the UML notation that is used by an ANZAC specification has a number of expressive advantages over its corresponding Fusion's analysis notations. An ANZAC specification uses a restricted UML protocol[2] state machine, with some small enhancements, to define the Protocol Model. Similarly to the lifecycle model, the Protocol Model is used to describe the temporal ordering of operations. The state machine of the Protocol Model allows one to better capture complex system behavior due to the explicit notion of state. Also, as is explained in the Chapter 7, the Protocol Model provides a basis from which one can define concurrent system behavior and timing constraints. Also, the Protocol Model could be seen rather as an enhancement of OMT's dynamic model—the predecessor of Fusion's lifecycle model, which uses the statechart notation [RBP+91]. Note that state-

1. This is one advantage that Fusion has over OMT, whose three analysis models are not so cleanly divided.
2. UML describes a protocol state machine as a special kind of state machine that has operations on transitions rather than action expressions [pp. 2-170, Omg01].

charts, originally defined by Harel [Har88], are also the basis of UML state machines [pp. 2-145, Omg01].

Instead of the (extended) entity-relationship diagram of Fusion's object model, the Concept Model of an ANZAC specification uses a UML class diagram to define the relevant information of the problem domain. A useful feature of UML for this purpose was the ability to provide a class diagram that is specifically tailored for the purpose of capturing the Concept Model. This was made possible through UML's stereotype extension mechanism, as is discussed in Chapter 5. In addition, system invariants can be formalized on class diagrams by the use of OCL constraints, which allows a more complete and precise description of the state space of the system.

Fusion's schemas are written using structured natural language. Natural language can be used to specify all required properties, imposes no implementation bias (in theory), and requires no special training to read and write. It nevertheless has the disadvantages that it can be ambiguous (more than one interpretation), inelegant (clumsily stated), noisy (extraneous text), obscure (lack of clarity in what is being conveyed), contradictory (two or more aspects impossible to satisfy simultaneously), inconsistent (different ways of representing the same thing), and incomplete (omissions) [Mey85]. Furthermore, there is no possibility to automate checks for these aspects, beyond the most superficial syntactic level. From the designer's point of view, and particularly for large specifications, these negative aspects and in particular the variable level of detail that usually results from the use of natural language may prove to be less attractive than a formal language that is tailored for the purpose of specifying system-level operations.

In the Operation Model of an ANZAC specification, operations are specified using OCL, which ensures a more consistent level of detail, allows precision by the reducing the chances of ambiguity, and is amenable to automated checking. In addition, an Operation Model is able to express the correctness contract on concurrent operations, as is discussed in Chapter 6.

This last aspect also points out the difference between the systems that are targeted by an ANZAC specification and the Fusion analysis models. On the one hand, the analysis models of Fusion can only be used in the development of principally sequential systems and a restricted subset of concurrent systems (only those systems with concurrent operations that have disjoint state spaces), due to the assumption that system operations occur instantaneously. On the other hand, an ANZAC specification can be used to specify both sequential and concurrent systems, which may even have explicit timing requirements.

The ANZAC approach also makes a deliberate effort to justify the ANZAC specification with respect to stakeholder goals by providing an explicit connection to the ANZAC use case descriptions (ANZAC's Behavioral Stakeholders Contract). As a consequence, it is less essential to make the ANZAC specification accessible to non-technical stakeholders, because negotiation with the stakeholders is generally performed at the level of the ANZAC use case descriptions. This point is exemplified by the use of a formal language to define operations in the Operation Model of an ANZAC specification.

Another difference between the ANZAC approach and Fusion analysis is the prescribed relationship to design. The ANZAC approach makes some attempt to distance itself from the idea that the state space as defined by the Concept Model can be seamlessly mapped to design objects in an object-oriented design (see Chapter 6 for a discussion of this point). However, Fusion arguably does not go far enough in distancing itself from the label of a “data-driven” object-oriented approach[1], especially since it uses the same object model for both the analysis and design phases. This last point can mislead developers into thinking that a good object-oriented design can be obtained by just adding methods to analysis objects with the addition of a few controller objects.

2.2.3 Use Cases

Since their introduction in the late 80’s [Jac87], use cases have proven to be an immensely popular software development tool [Coc97, SW98, Fow99a, Jar99, KG00, LW00, CL01]. Originally, use cases were proposed as black-box descriptions of system functionality, expressing system usage. And, they were promoted as a basis for software construction and testing activities. Although use cases are still very much applicable within this scope, the envelope of use cases has been expanded to cover numerous contexts and purposes, some of which even address non-software systems, e.g., organizations [JEJ95, Coc00].

Narrowing the wide-ranging contexts and purposes possible with use cases, some of the characteristics and qualities of use cases that make them a popular choice as an artifact for eliciting, analyzing, documenting, and validating the functional requirements of a software application are given below.

- Use cases usually relate well to stakeholders: they are informal, often narrative descriptions (written in natural language) that are expressed in a form that is accessible to stakeholders, technical and non-technical equally. Furthermore, use cases can be used to explicitly express the interests and expectations of the stakeholders and they can be closely tied to goals that users have against the system [Coc97, LX99].

1. The term data-driven was coined by Wirfs-Brock [W-B94b] in relation to approaches that identify and partition objects based purely on data rather than responsibilities [WWW90]. Many of the methods that are based on traditional entity-relationship modelling, e.g., OMT [RBP+91], Schlaer/Mellor [SM92], Coad/Yourdon [CY91], etc., get given this label. The problem with data-driven approaches is not an inherent one, but rather it is a problem that is related to those approaches that claim a seamless mapping from analysis objects to design objects. For a data-driven approach, attempting to seamlessly map analysis objects to design objects will often result in a design that exhibits an uneven distribution of behavior across objects. In the resulting design, controller objects will usually tend to do all the work (making maintenance difficult due to the amount of knowledge that these controllers store about other objects) and other objects usually have few methods and are often equivalent to database tables [SC92]: not reflective therefore of sound object-oriented design [Rie96].

- Use cases are a scenario-based description and they can be used to describe the system as a black-box. Both of these characteristics relate well to the users' view of the system—stakeholders being often users in one way or another. Furthermore, according to Carroll [Car95], results from cognitive psychology indicate that scenario-based techniques offer a middle-ground abstraction between model and reality. Therefore, with use cases, less time is spent convincing stakeholders that the representation is appropriate, compared to model-based techniques for capturing requirements—a limitation of modelling techniques according to [NE00].
- Use cases offer a scalable description that is recursive in nature, allowing specifiers to smoothly scale the description to large and small systems alike. The recursive nature of use cases also promotes refinement and traceability from the description of high-level usage goals right down to the description of low-level usage goals.
- Use cases support a layered style, which allows one to encapsulate the interactions that are perceived to be subject to change in separate use cases (i.e., use cases support abstraction).
- Use cases emphasize the capture of exceptional as well as normal behavior. The use case writing process involves identifying variation points, which naturally require one to brainstorm exceptional as well as normal behavior.
- A complete set of use cases specifies all the different ways to use the system and describes the way the system behaves in the context of its environment. Therefore, they define all external behavior required of the system, bounding the scope of the system [MB99]. This is an important characteristic of use cases, because a system is evaluated with respect to its environment and usage.
- Use cases are sufficiently flexible and lightweight to be able to be used in highly iterative and incremental development environments. This is an important requisite for any software development artifact, since evolutionary software development lifecycles are extremely popular in practice [HU00]. Furthermore, Parnas and Clements [PC86] observed that it is unrealistic to believe that software can be designed from a complete specification, implying that iterations are inevitable. Use cases are fitting for iterative processes, because they offer a formalism that facilitates the refinement and exploration of the system's interactive behavior, where with each iteration the use cases are refined or changed according to the knowledge gained during the iteration.
- Use cases have a wide range of application, i.e., they can be used to describe many different kinds of systems from many different domains. Rather than just attributing the versatile nature of use cases to the expressive power of natural language, it is probably more likely due to imprecision in the definition of a use case. This aspect allows an enormous variability in scope, detail, focus, format, structure, style, and content [CL01].

Each of these points have contributed to the success of use cases in software development practice. However, some points have also a “dark side”. For instance, the imprecise definition of a use case and consequent large variations in scope, detail, focus, etc. can make it difficult for inexperienced users to know which “flavor” of use case is suitable for a particular context [Pol97]. It is therefore important for any approach that applies use cases to take into account the purpose and context of use by clearly defining the focus, format, structure, style, and content of the use cases to be created.

Equally, the first point has a negative aspect, because natural language descriptions are prone to ambiguity and imprecision. Gause and Weinberg [GW89] state that ambiguity is one of the most fundamental problems in defining requirements. Also, as highlighted earlier, automated support for detecting ambiguity, inconsistencies, and incompleteness in natural language descriptions does not exist, beyond the most superficial syntactic level.

On the other hand, a narrative description (natural language) offers maximum expressive power to its writers, allowing them to capture the complexities and subtleties of the informal real world. Also, by providing flexibility and a common ground in terms of comprehension to its writers, natural language makes use cases accessible at the level of the “lowest common denominator”. Furthermore, the narrative description of use cases provides a good basis for documenting and communicating the usage goals on the system among stakeholders.

The ANZAC approach utilizes use cases as the Behavioral Stakeholders Contract. It addresses the above stated “dark side” of use cases in the following ways:

- The problems related to large variations in scope, detail, focus, etc. that is possible with use cases is addressed by providing a use case framework. The ANZAC use case framework establishes the context and purpose of focus and provides a definition of use case that is centered on this context and purpose. Furthermore, it provides usage and style guidelines for the proposed use case “flavor”.
- The problems related to natural language descriptions, i.e., ambiguity, inconsistencies, etc., is reduced by a conscious effort to 1) make use cases concise and 2) ensure that use cases do not contain low-level system detail.

If one makes a parallel between a natural language requirements document and the use case descriptions for a system, then making them concise and with not too much detail is a step in the right direction according to Davis [Dav88]:

> *“The larger the natural language document becomes, the more impossible consistency and maintenance become.” ... “The size of the document rarely has any relationship to the complexity of the problem. The larger documents are usually created by organizations that are attempting more ‘completeness’.”*

With respect to ANZAC, the usage guidelines that are given by the use case framework steer the specifiers toward concise use case descriptions. And the ANZAC mapping technique establishes an explicit relationship between the use case descriptions and the ANZAC specification. This relationship helps to control the level of detail about the system in ANZAC use cases, i.e., ANZAC use case descriptions are only decomposed until they reach a certain level of detail; from this level, the ANZAC specification "takes over", describing the system behavior in more detail (and with more rigor).

2.2.4 Catalysis

The Catalysis approach is a component-oriented development method [DW98]. It offers a very rich set of features for modeling component-based and object-oriented systems. In terms of object-oriented modeling, Catalysis integrates many of the favorable features of both Fusion and Syntropy [CD94]. Catalysis builds upon Syntropy's notions of refinement and types, and the use of OCL for describing pre- and postconditions on operations—Syntropy used the earliest form of OCL. In what follows, I concentrate only on a few of Catalysis' plethora of features—those ones that have direct relevance to this work.

Catalysis defines three levels of modeling[1], of which only the first two I follow with a discussion:

- problem domain or business – captures the relevant concepts in the environment or the concepts of relevance to the stakeholders of the system;
- component specification – captures the external behavior required of the component;
- component implementation – captures the internal workings of the component.

Modeling at the "problem domain or business" level is important for capturing the context of the system. The explicit expression of this development concern as a modeling level is a welcome addition to object-oriented methods that have traditionally paid little attention to this highly important aspect of software development. An explicit description of the environment of the system can lead only to a better understanding of the role that the system plays in that environment, and therefore, the resulting system is more likely to be appropriate.

Leveraging these three modeling levels, Catalysis supports the controlled refinement of actions. Actions can be decomposed into subordinate actions, or composed to form a superordinate action. An action represents work performed by one or more entities, which is described by an action specification, generally using pre- and postconditions written in OCL[2]. Catalysis defines two kinds of actions: localized and joint actions, where actions are categorized as localized or joint according to whether it has one or many participants. More

1. For the purposes of the discussion of Catalysis' levels of modeling, the term "component" can be replaced by the term "system" in the description of the second and third levels so that it is more aligned with the focus of this work. Note that the two concepts are not synonyms in general.
2. Note that Catalysis' authors do mention that descriptions other than pre- and postconditions are possible, but they do not describe these other options in any sufficient detail to be usable for action specifications.

precisely, Catalysis defines joint actions to describe multi-party collaborations, and localized actions to describe strictly the services provided by a particular type.

Catalysis' proposal of actions for refinement through all three modeling levels is a highly attractive feature for rigorous and systematic development. Unfortunately, Catalysis does not go far enough in elaborating how joint actions should be described at the "problem domain or business" level. Clearly, pre- and postconditions are highly inadequate for capturing the complexities and subtleties of a group of business interactions (i.e., as a joint action specification). Also, Catalysis omits any clarification on how actions relate to goals, interests and expectations of stakeholders, which means that justifications of high-level actions are at best implicit. In my opinion, this information is not only important for justifying high-level actions, but it drives the refinement process—refinement steps should only be taken in consideration of the goals, interests and expectations of the stakeholders.

The ANZAC approach makes quite some attempt to provide a description that can be used to clarify and negotiate the system-usage goals of stakeholders and to capture their expectations and interests in the system with respect to these goals (ANZAC use case descriptions). The ANZAC approach, however, makes an explicit separation between the description that captures the Behavioral Stakeholders Contract and the Behavioral Design Contract. As a consequence, a single mechanism is not used throughout the modelling levels.

Rather than emphasizing a difference of modelling level, the ANZAC approach justifies two distinct specifications by the mismatching concerns that need to be addressed in communicating the system in focus to the designers. In short, the ANZAC use cases are targeted towards capturing the concordant goals of stakeholders within the informalities of the real world, and the ANZAC specification is targeted at providing designers with a precise specification of system behavior. Note that the ANZAC approach establishes an explicit connection between the two work artifacts through the ANZAC mapping technique.

2.2.5 Pre- and Postconditions

Before I discuss model-based formal specifications, I believe it necessary to give some background on pre- and postconditions, since they are a predominant part of model-based formal specification languages and the work of this thesis.

Pre- and postconditions come from work on program correctness by Floyd [Flo67], Hoare [Hoa69], and Dijkstra [Dij76]. Originally, they were proposed as a means to prove the correctness of (existing) programs, where a pre- and postcondition pair denotes a correctness assertion on a program fragment. The motivation for pre- and postconditions was to be able to reason about the correctness of the whole program as the combination of the correctness of its parts (aka. divide-and-conquer).

A program fragment is considered totally correct[1] if it can be proved that any execution of it, starting in a state where the precondition is satisfied, terminates in a state where the postcondition is satisfied. Using Dijkstra's weakest precondition approach, it is even possible to calculate the most general precondition (i.e., the weakest one), given the postcondition and the program fragment.

However, it became evident over time that pre- and postconditions could equally be used as a design tool, where an operation (i.e., a coherent program fragment) could be described even before a line of code was written for it [Gri81]. To this end, pre- and postconditions offer a means to constrain the required behavior of an operation. Furthermore, if the assertions are strong enough, they express everything that the caller and the designer needs to know about the operation without disclosing how the operation is or should be designed/implemented. Indeed, pre- and postconditions prove to be one of software engineering's premier tools for combating overspecification.

2.2.6 Model-Based Formal Specifications

Model-based formal specifications define a model of a system using well-understood mathematical entities such as sets, relations and functions. A relation is a set of pairs; a function is a relation in which no pair share the same first element. Operations are specified by defining their effect on the system state; they are described using pre- and postconditions [AP98]. Pre- and postconditions can be seen simply as a binary relation on the state space of the system. The use of a model-based formal specification requires that the mathematical model, which captures both the static and dynamic properties of the system, be refined, formally or otherwise, to an execution model.

In what follows, I briefly discuss three model-based formal specification languages/ approaches: Z, VDM, and Alloy. These three languages have particular relevance to the ANZAC specification, which makes use of pre- and postconditions in its Operation Model and describes an abstract state space with its Concept Model in a similar way to those proposed by Z, VDM and Alloy.

The formal specification language Z [Spi89] is based on set theory and classical first-order predicate logic. An important feature of the Z language is the schema. A schema can be viewed as an encapsulated set of logical formulas associated with some properties, which can be used to model both the static and dynamic properties of systems. Schemas are commonly used in Z to represent types, state invariants and operations. Unfortunately, the schema notation does not distinguish the role played by different schemas, for example, whether a schema represents an invariant or an operation. Furthermore, a schema has only two parts: a declaration and a predicate part, which means that preconditions and postconditions are intermingled in the predicate part of a schema for an operation. In fact, to be more precise, the precondition is not explicitly stated; it has to be calculated from the delta schema definitions.

1. Note that there is also a notion of partial correctness. I point interested readers to [Hoa69] for a description of partial versus total correctness—a good discussion is also given in [Mey90].

Using schemas, it is possible to specify parts of a system separately, and then compose the parts (schemas) to obtain the specification of the whole system. Moreover, Z has a powerful set of operators to manipulate schemas and to support incremental specification. These schema-level operators include: composition, hiding, disjunction, conjunction and negation.

In addition to Z, which is strictly a specification notation, VDM [Jon86] is a formal specification language that supports the modeling of software systems at the imperative level in addition to the assertion level. In VDM, data structures are built from a library of algebraically-defined datatypes. VDM uses a three-valued logic, allowing for the notion of partial functions in modeling programs, which better lends itself to the specification of realistic problems (programs are rarely total). However, with the introduction of “undefined”, the approach is more complicated than the traditional “total function” model.

In VDM, both data and algorithmic abstractions expressed at one level can be refined to a lower level for the purpose of obtaining a more concrete model that is closer to the final implementation of the system. For instance, a VDM specification that is written in an assertional style (pre- and postconditions) can be refined into another VDM specification that is written using (imperative) statements.

Z and VDM are both languages that offer the specifier a toolkit of rich formal notations, which offers more or less[1] an alternative language to UML and OCL in an ANZAC specification. However, they suffer from the problem that they are typically expensive to introduce into software development environments, as is the case with most formal methods, because of their high requirements for mathematical maturity on the user. Also, even if adequate expertise for constructing a formal specification is available, it may not be possible to keep a specification up-to-date in many modern project settings, due to the amount of effort that needs to go into constructing and analyzing a formal specification.

However, the potential benefits of applying formal analysis techniques to a formal specification are attractive. Some of the potential analyses include: removal of ambiguity, an understanding of system properties based on appropriate formal reasoning, detection of (some) incompleteness and inconsistency, and verification of the correspondence between the specification and a subsequent design or implementation [FFM90].

Mindful of both the positive and negative aspects of model-based formal specifications, Jackson proposed a lightweight formal specification language called Alloy [Ja02]. Alloy is inspired by Z, and therefore, models are constructed using types, sets, relations and functions. One principal objective of Jackson’s work on Alloy was to make it more easily used by non-mathematicians than Z. In this direction, Alloy is a small language with a remarkably small kernel language, and it uses a textual notation—in contrast to Z’s dependence on LaTeX. Unlike Z, Alloy uses the dot notation for navigation over the relations of

1. Z and VDM can not describe interactions and therefore they are not able to capture all behavioral aspects of a reactive system.

the model, and it allows operations to have arguments in the traditional programming sense (actuals are substituted for formals at invocation time). Again, in a deviation from Z, Alloy provides a syntax that promotes the construction of expressions that principally manipulate sets rather than relations. To this end, Alloy does not support tuples.

Taking a step closer to object models (à la OMT), Alloy supports object modeling idioms, e.g., it uses multiplicities, and it provides a graphical sublanguage of its textual language for visualizing the static structure aspects of a model. Interestingly, in Alloy, all objects are treated as members of the set of their corresponding type, and it treats all scalars as singleton sets. This means that all navigations over relations result in sets, with no undefined expressions or special null values are required. Furthermore, Alloy allows two kinds of analysis: simulation and model checking, which are supported by a tool prototype.

Considering that Alloy does not address the modeling of interactions, it is not exactly comparable as it stands to the formalism used in ANZAC specifications (i.e., UML and OCL). However, apart from OCL's obvious advantage in its tight coupling to UML (with respect to continuity in an ANZAC specification), Alloy is comparable to OCL.

The creators of Alloy and OCL had similar high-level objectives in developing their languages, and they both claim Z as their initial base. Furthermore, Alloy and OCL have a number of features in common: they both promote navigation over the model, and they provide a syntax that promotes the construction of expressions that manipulate sets instead of relations.

That said, OCL has a more operational style, which, although sometimes verbose, is arguably easier to learn and use for developers that are not versed in classical predicate logic but have a strong background in procedural programming. Taking a purist's stance, one could always argue that an operational style presents a risk in that it may influence the form of the implementation. However, there is some anecdotal evidence [GPZ93, Fuc92] and my own personal experiences that suggests that the advantages gained in terms of usability with an operational style may well outweigh any bias that it may have towards a particular implementation.

In fact, the work on the Operation Model presented in Chapter 6 makes a number of proposals that even extend the operational-style of OCL, giving it a procedural programming language-like flavor, and reducing the strict declarative style of pre- and postconditions written using OCL.

The capability offered by Alloy to perform significant analyses on the model (simulation and model checking) is probably the most noteworthy feature of Alloy that is currently not addressed by OCL—it is still unknown to what extent OCL can be analyzed. In [Ja02], Jackson warns that the difficulty of putting a language in an analyzable form should not be underestimated. Nevertheless, there seems to be little concrete evidence that would suggest that formal analyses could not be performed on OCL, and there have been some positive steps towards clarifying this area and providing tools to support this task [HCH+98, HHK98, RG98, Ric01, RG02, KO01].

2.2.7 Rely- and Guarantee-Conditions

Taking a more focused view of the effect of operations on the state of a system, it is possible for a system to have operations that change the state of the system at the same time. Furthermore, it is possible that operations access common resources of the system. If such operations were to execute in parallel, then interference problems could arise. However, the general assumption in describing pre- and postconditions for operations is that only one operation is executed at a time, where pre- and postconditions can be interpreted simply as a binary relation on the state space of the system.

Rely- and guarantee-conditions were proposed by Jones [Jon83] as an extension to operation specifications that use pre- and postconditions. These additions allow one to cope with the specification of concurrent operations that share resources and may have overlapping executions. A specification that makes use of rely- and guarantee-conditions consists of four assertions, which can be grouped into two logical parts: assumptions and commitments—similar to the concepts originally proposed by Francez and Pnueli [FP78] for proving the correctness of concurrent programs. The assumptions consist of the precondition and rely-condition, and the commitments consist of the guarantee-condition and postcondition.

The meaning of the assumption/commitment specification is the following: if all activities other than the one executing the corresponding operation observe the rely-condition during the execution of the operation and the precondition holds initially (the assumptions), then the operation will terminate in a state that satisfies the postcondition and the guarantee-condition will have been held by the operation throughout its execution (the commitments). Furthermore, if an operation is invoked in a situation when the precondition is false, or if during the execution of the operation, the rely condition is not observed by the other (executing) operations, then the specification does not state what the outcome should be (nor is it obligated to terminate). In other words, any change to the system state by any other operation over the period of execution is supposed to satisfy the rely-condition, while any change to the system state by the operation must satisfy the guarantee-condition.

The advantage of an assertion-oriented approach should not be underestimated, because they offer a specification that does not prescribe how the operation should be realized. Assumption/commitment specifications offer an effective approach for describing concurrent operations, making use of rely-/guarantee-conditions.

Assumption/commitment specifications offer an alternative to approaches that define operation-level exclusion policies, e.g., SCOOP reinterprets and uses the precondition as a wait-condition (i.e., an entry guard) [Mey97]. Clearly, with respect to such an approach, the granularity of concurrency is set at the method level, which (theoretically) restricts designers to also follow suit. Furthermore, such an approach may not even be an optimal choice at design, because, in describing coarse-grained system operations, it may be deemed that it is too inefficient to enforce exclusion at the operation-level.

Also, assumption/commitment specifications offer an alternative to temporal logics [Eme89] for describing safety properties. However, temporal logics usually do not provide descriptions that are functionally complete[1]—they usually describe assertions about the program as a whole, without reference to operations.

A downside of rely-/guarantee-conditions is that they apply to the whole execution period of an operation. In some situations, a constraint that spans the whole operation may be too strong. To illustrate the problem consider a withdraw operation of a bank system, and in addition, an invariant on the bank that states that all accounts should never have balances below zero. In specifying the withdraw operation using the assumption/commitment approach, one would typically write a rely-condition that states that other operations do not put the account below the amount that should be withdrawn. At face value, this constraint seems reasonable, but unfortunately it requires that the balance is held above the amount to be debited (by the other operations) over the whole operation period and not just during the period that the account's balance is debited. For example, say the operation started at time 0, debits the account between times 2 and 3 and the operation terminates at time 5. For this scenario, the rely-condition requires that the balance stays above the amount to be debited between times 0 and 5 inclusive, and not just between times 2 and 3 inclusive. This means that some valid scenarios would be not allowed by the specification.

With the endeavor to address this problem, in this thesis, I propose the rely expression, which has some similarities to rely-conditions, since they both define "during" invariants. In contrast to rely-conditions, rely expressions are used within postconditions and they have scope over only a subset of the operation's effects. Another difference is that a rely expression has a fail predicate that is asserted if the condition that is relied upon can not be held during the period that the scoped effect is being carried out.

Similarly to rely-conditions, rely expressions obligate all other operations (i.e., except the operation being described) to fulfill the condition during the execution period of the scoped effect. Due to the predicate nature of a postcondition, rely expressions impose neither immediate or wait semantics on conditions, and they do not even discount a (running) implementation from making several attempts[2] at carrying out the scoped effect. This descriptive nature of rely expressions provides many possibilities to designers without restricting valid designs. Rely expressions are presented in Chapter 6.

1. Temporal logics were not introduced with the intent to specify the responsibilities of operations as illustrated by pre- and postconditions, but rather as a means to document and eventually prove the satisfaction of safety and liveness properties of a (multi-threaded) program as a whole.
2. The meaning of "attempt" is in the transactional sense of the term, i.e., possible rollbacks and retries.

Chapter 3:
Specifying the Behavioral Stakeholders Contract

This chapter introduces the part of the ANZAC approach that involves defining the Behavioral Stakeholders Contract, called the ANZAC use case framework. It motivates the choice of description, details how the ANZAC use case framework builds upon related work, and provides a definition for an ANZAC use case. It also details the elements, style, and format of ANZAC use case descriptions, explains the kinds of requirements that can be captured with them, explains how other kinds of requirements can be related to them, and defines some techniques for constructing them and dealing with changes to requirements. Finally, it discusses some of the limitations of ANZAC use case descriptions for defining the Behavioral Stakeholders Contract, and it proposes some possible ways to minimize some of these limitations.

3.1 Introduction

The requirements for an application under development are used and even perceived in different ways during different software development activities. Ideally, the requirements should be represented in a way that addresses these different perspectives [Eas91], requiring more than one way to represent them [NKF94, SK98]. In Chapter 1, I introduced two perspectives on the system requirements, which I referred to as the Behavioral Stakeholders Contract and Behavioral Design Contract. The theme of this chapter is the Behavioral Stakeholders Contract.

I define the Behavioral Stakeholders Contract in the following way:

Behavioral Stakeholders Contract – is a specification of the concordant goals that the stakeholders have against the system and that constrain or induce system behavior. It is expressed in a form that is accessible to all stakeholders, encouraging discussion and participation from them, and it allows the capture of the subtleties and complexities of the informal real world.

In the endeavor to provide a concrete means to represent the Behavioral Stakeholders Contract for reactive systems, this chapter presents the ANZAC use case framework, which is used to guide the elicitation, documentation, organization and validation of system-usage goals. The result of applying the framework is a set of ANZAC use case descriptions, which each capture a particular goal that a stakeholder has against the system.

This chapter is structured in the following way. In section 3.2, I provide some background information on use cases. In section 3.3, I highlight some of the different contexts and purposes for which use cases have been successfully used. Furthermore, from the various contexts and purposes provided, I pinpoint the combination of interest to this work, i.e., the combination considered in the definition of the ANZAC use case framework.

In section 3.4, I provide a survey of some of the more popular and well-known definitions of use cases, and motivate from these definitions and my own experience, the definition of an ANZAC use case. In section 3.5, as part of the informal explanation of the ANZAC use case framework, I provide details on use case capture by goal elicitation. In section 3.6, I define the different categories and personalities of actors. In section 3.7, I describe how ANZAC use case descriptions relate to extra-functional requirements and project constraints. In section 3.8, I discuss some issues in managing change in ANZAC use case descriptions. In section 3.9, I describe the proposed form and style of ANZAC use case descriptions.

In section 3.10, I present an overview of the ANZAC use case framework, showing a more consistent view of the framework as a whole. And finally in section 3.11, I describe some of the limitations that ANZAC use case descriptions have for defining the Behavioral Stakeholders Contract, and I propose some possible ways to minimize some of these limitations.

3.2 Background on Use Cases

Before one can critically analyze use cases and introduce or propose a definition and representation for use cases, it is important to define exactly what one means by a use case and what context and purpose they are to be used in.

There is a substantial amount of literature on use cases, which continues to grow in response to the increase of projects that are taking advantage of the many benefits that can be gained from specifying and documenting systems with the aid of use cases. Use cases have been successfully employed in many different types of projects and in many different domains and environments. Mostly based on practical experience gained in industrial projects, the literature on use cases often explains a certain flavor of use case that was applied in a certain context with a particular purpose. Unfortunately, the purpose and context is often unrecognized or left implicit with the resulting approach being purported as universally applicable and suitable, rather than being made explicit. An exception to this observation is Cockburn's, and Constantine and Lockwood's work. Cockburn points out that there are at least 18 different definitions of use case, which differ along 4 dimensions: purpose, contents, plurality, and structure [Coc97]. He offers some insights into ways to classify different approaches by proposing different scopes, abstraction levels, and formalities of a use case [Coc00]. Constantine and Lockwood focus on the deficiencies of the various use cases definitions for the purpose of user interface design [CL01]. They propose a formalism called essential use cases [CL99].

Originally an initiative to unify the various notations used in object-oriented methods and now an initiative to provide a unified software modeling notation in general, UML offers a unique opportunity to classify the different uses, forms, styles, etc. of use cases. However, currently the standard has not taken any such measures. UML instead takes a generic approach to describing use cases, which has lead to an unrevealing definition of a use case, as we will see in section 3.4, and a broad set of guidelines for their use that are only marginally informative.

One way to improve the explanation of use cases to specifiers would be to partition up the different kinds of use cases, and describe each one in detail, i.e., define a family of use case kinds. In this direction, one approach to classification could be to define a separate use case framework for each significant combination of context and purpose. Thus, a use case framework would be focused on a particular purpose, i.e., it can be distinctly placed and its role precisely defined, and focused for a certain set of (related) contexts or environments. A use case framework, for instance, could define a set of building blocks that can be used in the construction of a use case, along with their meaning, rules, and guidelines for use, a set of coherent styles and formats, and general approach for their development.

In this chapter, I take this approach by informally proposing a use case framework for the context and purpose of interest to this work called the ANZAC use case framework. Initially, the description of the ANZAC use case framework is interspersed among the various sections, but towards the end of this chapter, I summarize it bringing together the various pieces. Note that this chapter will describe the ANZAC use case framework informally and it does not address any requirements for a language that is suitable for describing all possible use case frameworks. That is to say, the classification of the different frameworks and the definition of an appropriate language for describing a use case framework is beyond the scope of this work.

3.3 Contexts and Purposes for which Use Cases are Applied

Even though use cases are widely and successfully used in many different software projects, there are large differences of opinion on the best way to apply them and particularly on the most effective form and content a use case should have. Apart from the imprecise definition of use cases [CL01], one reason for the large number of use case variants is that they have been used for different contexts and with different purposes. Consequently, developers have evolved the use case formalism to suit those particular contexts and purposes. Use cases have without a doubt evolved far beyond the original intention of Jacobson, the creator of use cases [Jac87].

To avoid any misconception that the ANZAC use case framework is suitable for all contexts and purposes, I discuss some of the different contexts and purposes for which use cases can be applied in section 3.3.1, and then refine this list of possibilities for the context and purpose for which an ANZAC use case description is designed for, in section 3.3.2.

3.3.1 Possible Contexts and Purposes for Applying Use Cases

Due to its heritage, a use case generally always describes a system that interacts with its environment. Within the bounds of this constraint, there are a number of different things a system described by a use case can be; these include: a computer system (hardware, software, or combination), a component of a computer system, a mechanical system, a group of humans, or a combination of the previous ones (e.g., an organization). Clearly, the number of domains of application that use cases have the potential to touch are enormous, because interactive systems can be found in many different domains. Combine this with the different ways to apply use cases and it would be fair to say that it would not be possible to define a single use case framework that is optimal in all situations.

In software development, project contexts vary according to the resources allocated/available (people (team sizes and competencies), processes, tools, facilities, time-to-delivery constraints, and budget), domains of application, external forces, the types of systems under development, and the quality demands on the system. In the analysis of systems that are not uniquely software, contexts vary according to the domains of focus, type of system, and the constituents of the system being described by use cases: humans, machines (computerized and mechanical), or a combination of the two; although, it is necessary to point out that use cases come from and have predominantly been used for software development.

Independent of these contexts, use cases can be used for the purpose of specifying certain aspects of an already existing system: to document its behavior in terms of what services it offers, or in terms of how it realizes its services; to document the rules for use of the system; or to document the workflow that goes on within the system between its components. And use cases can also be used for the purpose of specifying certain aspects of a system under construction: to informally or formally specify the functional requirements for a system; to specify how the system will realize the desired behavior; to document the perceived risks of using the system; or to specify the envisaged changes to the usage requirements of the system. In summary, a use case describes a certain behavioral aspect of a system already in existence or under construction.

At the description and form level, use cases have large variability in scope, detail, focus, format, structure, style, and content. For instance, the scope of a use case can be a large organization or a small software component. The detail of a use case can vary from describing fine-grained user interaction to coarse-grained interactions between enterprises. The focus of a use case could be on user interface interaction, essential system behavior, internal interaction, system usage rules, etc. The format and structure of use cases can vary from no specified format to a formal "fill in the blanks" template. The style of a use case can vary from causal to formal. And the content of a use case varies according to domain, context, and purpose with which use cases are applied and therefore the variation is large.

3.3.2 The Context and Purpose for Applying Use Cases in this Work

The context of interest for the ANZAC use case framework is software application development. Use cases that conform to this framework have been used in the development of business systems, embedded systems, application tools and utilities, and games. These use cases are suited to the description of reactive systems. In particular, they are not suited to software applications that are transformational in nature where all input is given at one time, i.e., applications that principally involve processing data using possibly large and complex algorithms, for example, compilers, batch processes, stream filters, etc. Also, the form of a use case description that I propose is not suited to the description of applications that are highly state-driven, where the behavior of the system is concisely described by a state-transition machine, e.g., traffic light controllers, because of the scenario-oriented nature of use cases. However, within the bounds of these guidelines, there are many different kinds of software applications that are effectively described by ANZAC use case descriptions.

The type of approach that the ANZAC use case framework targets is one that at least supports some form of requirements engineering activities. This means that the project is driven by at least moderate demands on correct and valid application behavior. ANZAC use case descriptions are employed with the purpose of eliciting and validating the functional requirements for an application under development (not excluding the fact that already existing components may be used to construct the future system). The assumption in applying the ANZAC use case framework is that the need for a software application has already been established but the software system's role in solving or helping to solve the problem has not yet been defined.

The desired result of applying the ANZAC use case framework is to obtain a set of use cases that defines the complete and realistic behavior of the future application without revealing the details of its internal workings, i.e. a black-box description. The targeted scope for ANZAC use case descriptions is any software application that provides services largely to human actors. The reason for the requirement of mostly human actors is due to the informal nature of these use cases, making them suited to the description of user-interactive systems and less suited to the description of machine interactions, where the protocol is often strict and pre-imposed. A consequence of this constraint is that the scope of an ANZAC use case cannot be too fine-grained.

3.4 Defining Use Cases

In this subsection, I present and discuss the definitions of three of the more well-known authorities on use cases, namely, Jacobson, UML, and Cockburn. The aim is to highlight some fundamental aspects of use cases and to point out those aspects that are interesting and less interesting for the use case context and purpose stated in section 3.3.2. The result of this subsection is a definition of a use case that takes into account the critical analysis of the definitions and draws upon some of the knowledge gained from my own experiences in eliciting, analyzing, and validating requirements with use cases. The resulting definition is taken as the reference one for the ANZAC use case framework.

3.4.1 Jacobson’s & UML’s Definition of a Use Case

The original work on use cases comes from Jacobson. He developed use cases while working in the telecommunications domain [Jac87, JCJ92]. Jacobson’s definition of a use case is the following one [JBR99].

> *Definition 3.1: A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor.*

An insight that one can take from this definition of a use case is that it exists for the benefit of one particular actor. Jacobson’s work became the basis for use cases in UML [BRJ99] and therefore it is not a surprise that their definitions are close. UML defines a use case in the following way [Omg01].

> *Definition 3.2: Use case [class] – The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.*

UML’s definition is more general than Jacobson’s one—it does not specifically require that a use case yields an observable result to an actor—no doubt the result of a compromise between the partners that were involved in UML’s definition. However, Jacobson’s definition converges with UML’s one if we assume that a sequence of actions described by a use case always provides value to an actor. UML goes on further to explain the purpose and makeup of a use case [pp. 2-141, Omg01].

"The purpose of a use case is to define a piece of behavior of an entity without revealing the internal structure of the entity."
"A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity."

These extracts imply that use cases are black-box descriptions of an entity, disqualifying the white-box flavors of use case described in section 3.3.1. This is a little surprising for a standard that attempts to cover most of the contexts defined in that subsection. Nevertheless, these extracts are still useful for understanding the use cases defined by the ANZAC use case framework, because the ANZAC use case framework also propose black-box descriptions.

Looking closer at the above extracts, we can see that a use case defines a piece of behavior by an entity related to a series of interactions between it and its users. UML goes on further stating that a use case is directly related to a service offered by the entity [pp. 2-141, Omg01].

"Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again."

The last two sentences of the extract emphasize that a use case is a complete task that is generally context independent (i.e. for it to be possible to initiate it again). Also, UML explains the meaning of "variant" in its definition [pp. 2-141, Omg01].

"A use case also includes possible variants of this sequence (for example, alternative sequences, exceptional behavior, error handling, etc.). The complete set of use cases specifies all different ways to use the entity; that is, all behavior of the entity is expressed by its use cases."

We can derive from this extract that the variants of a use case's "sequence of actions" could be alternative sequences that define normal or exceptional (recoverable or otherwise) behavior. Another point we can derive from this extract is that it is possible to define a set of use cases that completely describes the behavior of the system, which implies that all behavior of an entity can be related to the usage of that entity.

Taking into account UML's definition, the extracts from the UML specification, and Jacobson's definition, we can see that a use case's "sequence of actions" defines a piece of behavior of the entity that results from a series of interactions that take place when the entity is being used by a number of its users.

If we are to better understand these definitions, we should also understand what is meant by the terms *actor* and *action* used in the definitions.

3.4.2 Defining Actor

UML defines an actor in the following way [Omg01]:

Definition 3.3: Actor [class] – A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

And it defines a role in the following way [Omg01]:

Definition 3.4: Role – The named specific behavior of an entity participating in a particular context.

Also, Wirfs-Brock gives some more insight into a role. She defines a role as a set of characteristic needs, interests, expectations, behaviors and responsibilities [W-B94a].

According to the first definition, an actor is a set of roles that are somehow connected, an actor is a participant in possibly many use cases[1], and an actor plays a single role in a use case. This last point is explained further by Stevens [Ste01]:

> *"... it is hard to imagine an example where more than one instance of a given actor communicates with a given instance of a use case... If there are two distinct such actor instances then almost by definition they play different roles in the use case and should therefore be modelled as instances of different actors."*

This would seem to imply that an actor (class) in a use case cannot "contribute" more than one actor instance per use case. For example, a use case that describes a four-player game would require four actor instances of different actors (class). This rule, however, is rather restrictive, because it would seem equally logical to choose *player* as an actor of the use case with four roles, e.g., player yellow, player green, etc., which define the four different roles of the players in the game. A player is also a role itself, which could be "played" by a user, for example, and user could also be seen as a role, and so on. Wegmann and Genilloud observed that an actor is itself a role that can be seen as a composition of roles [WG00].

Two improvements to the definition of actor are perceivable. The first one would be to remove the restriction that an actor can only play a single role in a use case, and the second one, inspired by the work of Wegmann and Genilloud, is to define an actor in terms of role composition. Taking into account these improvements, I propose basically the same definition of actor as the one proposed by Wegmann and Genilloud [WG00][2]:

Definition 3.5: Actor [class] – The specification of a role defined as the composition of the roles that a participant of one or more use cases plays when interacting in these use cases.

This definition would allow both interpretations of actors in the four-player game example from above, and it calls for an explicit means for denoting composition between roles, which I do not address in this work.

To better understand actors, it is also important to know what type of entities could realize an actor, i.e., could play the role. These types of entities include: mechanical sys-

1. derived from "... users of use cases play when interacting with these use cases"
2. I made minor changes to the structure of their definition.

tems, computer systems, hardware devices, software applications, persons, organizations, or some combination of these.

3.4.3 Exploring Actions

Looking into the term “action” used in Jacobson’s and UML’s definition, UML defines an action in the following way [Omg01]—the same definition used by Jacobson.

Definition 3.6: Action – The specification of an executable statement that forms an abstraction of a computational procedure.

UML’s definition of an action is not very specific to use cases. In fact, an action is used in many different areas of UML. In the context of use cases, the only way the system can communicate with its actors and vice-versa is by the sending and receiving of message instances. An action performed by the system is either activated by a message instance from an actor or it is internally generated by the system (i.e., the system is triggered by an internal timer or its scheduler). A message instance sent from an actor to the system represents either a request for a service, a notification, or a reply to a request made earlier by the system. We can therefore relate an action to a use case by associating one or more actions to a service offered by the system, where UML states that a use case encapsulates a service offered by an entity.

3.4.4 Cockburn’s Definition of a Use Case

Cockburn’s definition provides some additional insight into use cases by relating a use case to a goal of a particular actor. Cockburn defines a use case in the following way [Coc00]:

Definition 3.7: A use case expresses the behavioral portion of a contract between the stakeholders of a system. It describes the system’s behavior and interactions under various conditions as it responds to a request on behalf of one of the stakeholders, the primary actor, showing how the primary actor’s goal gets delivered or fails. The use case gathers the scenarios related to the primary actor’s goal.

And he defines a stakeholder in the following way [Coc00]:

Definition 3.8: A stakeholder is an external actor entitled to have its interests protected by the system, and satisfying those interests requires the system to take specific actions.

Cockburn’s definition of a use case provides a number of insights into the nature of a use case. First, it introduces the idea of a use case expressing (the behavioral part of) a contract between stakeholders, which means that a use case represents an agreement between the various stakeholders on the way the system should behave in a certain situation. Second, it reinforces the idea that a use case describes interactions of the system, involving the way the system behaves with its environment and vice-versa. Third, one can derive from the definition that a use case is driven by the goal of a (single) stakeholder, referred to as the primary actor (discussed in section 3.6). Fourth, by relating a use case to the goal of a stakeholder, it highlights that a use case should not only show goal delivery but also goal failure. Finally, it states that a use case is a collection of scenarios that relate to the stakeholder's goal, introducing the concept of scenario and defining its relationship to a use case.

An important insight to take from Cockburn's definition is that the focus of a use case should not just be on describing system services but also on giving the reason for providing those services, and furthermore that a use case represents an agreement that defines the interests of the stakeholders.

3.4.5 Understanding Goals, Interactions and System Responsibilities

In this subsection, we analyze goals, interactions, and system responsibilities and their relation to use cases. The aim is to better understand the driving forces in a use case. Looking further into the meaning of goal, according to Princeton University's WordNet [WN97], a goal is defined in the following way.

Definition 3.9: Goal – the state of affairs that a plan is intended to achieve and that (when achieved) terminates behavior intended to achieve it; "the ends justify the means".

Taking this definition into account, one can derive from Cockburn's definition that a use case describes the desired behavior of the system and interactions it has with its environment that make progress towards the achievement of the goal of a particular stakeholder, where two outcomes are possible for a use case, goal success or goal failure. This point provides a different view of the clause in Jacobson's definition: "... that yields an observable result of value to a particular actor", because one can derive from Cockburn's definition that any actor who receives value from the performance of a use case is in fact the stakeholder who has the goal against the system. Taking into account UML as well, we can relate the three definitions by stating that a use case is a description of a service that provides value to an actor and that, when successfully completed, fulfills the goal of a stakeholder.

3.4.5.1 Relating Goals to Interaction

Looking closer at stakeholder goals, a stakeholder will have a goal against the system because it wants to achieve a personal goal or equally because it is obligated to meet its responsibilities (driven by a goal of one of its own stakeholders). Cockburn states that an action connects one actor's goal to another ones responsibility (where the system can be seen as an actor), i.e., a goal is a request for a service, a responsibility is a promise of service [Coc00].

Concretely, one way that a stakeholder can achieve a goal that it has against the system is to interact directly with it. In this case, the stakeholder is an actor of the system. Taking a UML view of the world and focusing the discussion of use cases on software systems, the only way an actor can interact with the system and vice-versa is by the sending and receiving of message instances. To understand the makeup of a use case, we look at the different ways that a system reacts to message instances sent by an actor. The sending of a message instance to the system can trigger two interesting kinds of reactions from the system.

First kind of reaction: an actor sends a message instance to the system that represents a request. The request places a responsibility on the system to provide the required service under certain circumstances. The reaction can be explained in terms of four stages[1], shown below.

1. An actor sends a request, which includes the relevant data, to the system;
2. The system validates the request by verifying the supplied data and/or querying its own state;
3. The system processes the request by altering its internal state, and/or making one or more requests and/or notifications to actor(s); and
4. The system replies to the actor providing any possible feedback (positive or negative).

The stages occur in the order implied by the numbering, i.e., it starts with stage 1 and finishes with stage 4. Stage 3 may involve a number of interactions between the system and its environment, e.g., the system makes a request to the calling actor, prompting it for more information. Also, stage 3 is not performed if the request fails to be validated.

Second kind of reaction: an actor sends a message to the system that represents a notification, which provides the system with information that can be used to keep it up-to-date with its environment (that part of its environment that it is required to monitor). The reaction can be explained in terms of two stages:

1. An actor sends a notification, which includes the relevant data, to the system; and
2. The system processes the notification by ignoring it, or altering its internal state and/or making one or more requests and/or notifications to actor(s).

There is no explicit validation stage, because validation is not a noteworthy step in the reaction.

One other kind of message instance can be sent to the system: a reply. However, a reply is clearly only part of a complete interaction, and thus we do not detail it as a separate kind of reaction.

The first kind of reaction is started by a request that is driven by the goal of an actor, and the second kind of reaction is started by a notification that is related to some higher-level goal beyond the one that obligates the actor to send the notification. Looking more closely at this second kind of reaction, a notification is in fact driven by a goal that the sys-

1. It is an expansion of Cockburn's explanation of a transaction [Coc00]; a term he uses due to the original terminology used in Jacobson's first definition of a use case [JCJ92].

tem has against the actor. Notifications are responses to implicit requests from the system—information required by the system to be able to fulfill its responsibilities. Thus, ultimately these notifications can be traced to the system needing to correctly fulfill (current or future) requests for services.

In summary, both kinds of reactions can be traced to the request of an actor, where the request is tied to a goal that the actor has on the system. However, stakeholders do not necessarily need to interact with the system directly to achieve their goals against the system. A request for a service can be given to an intermediary, which in turn passes the request on to the system (often in a different form), or to a chain of intermediaries, given that the request eventually gets to the system. For example, a teller could be used to mediate the communication between a client (who has a goal against the system) and the system. Alternatively, a service required by a stakeholder may be automatically generated and processed by the system according to an agreement between stakeholders that the system is required to uphold. For example, the system may be required to send out a monthly magazine at the start of each month to all those people who are subscribed to the magazine. In this case, a subscriber is not required to ask for this service (to get the current issue) each month, rather the system automates the request and consequently upholds the agreement between the magazine company and the subscriber. In this last case, the system must exhibit active behavior to be able to trigger itself.

Thus, a stakeholder's goal may cause the system to perform a task automatically, without the need for a message from the corresponding stakeholder. Note that in most case there would have been an original request, e.g., a subscription request by a customer. Such behavior exhibited by the system can be explained in terms of three stages:

1. The system ascertains that it is time to perform a system-instigated action;
2. The system performs the action by altering its internal state, querying its state, and/or making one or more requests and/or notifications to actor(s); and
3. The system sends the result to the appropriate actor or stores it for later access.

Note that the system may not need to communicate with any stakeholder to fulfill its responsibility. Thus in this situation, the system may not interact with the stakeholder in question, or any other one for that matter.

Taking into account all the different ways that a goal of a stakeholder can drive various kinds of interactions and system-initiated actions, one can describe the system's participation in a use case as a collection of: the first kind of reactions, the second kind of reactions, or the system-initiated actions.

3.4.5.2 Problem-Grounded Goals of Stakeholders

The performance of a use case is driven by the goal of a stakeholder, and it is described from the perspective of this stakeholder. Clearly, it would not be effective to describe all goals of stakeholders using use cases. The goals of interest are only those ones that are directly relevant to the problem that the system will help solve (or improve). Nevertheless, a use case should also describe the (perceived) expectations of the stakeholder with respect to the goal s/he has against the system, offering further insights into the problem itself. One must remember that the problem is defined in the environment of the future system and not just at the interface of the software system and the environment [Jack00]. Ignoring such aspects may have economical and organizational consequences, e.g., the use case delivers a service to users in a way that does not suit their routine or wishes, which may render the system inappropriate or only moderately useful.

To better highlight that a goal is bound to what the stakeholder requires of the system, the term ‘need’ can be used instead of the term ‘goal’. It is a better choice when one wants to emphasize this aspect of a use case, because typical English usage of the word ‘goal’ usually invokes images of possibly unrealistic desires and wishes. However, in general, the term ‘goal’ best summarizes the effect that the stakeholder requires of the system, because it encapsulates its need as well as giving some insight into the stakeholder’s expectations of the system.

3.4.6 ANZAC’s Definition of a Use Case

Drawing on the analysis of use cases given in the previous sections, I propose the following definition of a use case.

Definition 3.10: A use case is the specification of a non-empty, partially ordered collection of system-instigated actions and/or actor-instigated interactions between the system and one or more of its actors. A use case describes, without revealing the details of the system’s internal workings, the system’s responsibilities and its interactions with its environment as it performs work in serving one or more requests that, if successfully completed, satisfies a goal of a particular stakeholder. A use case expresses this stakeholder’s need and its possible expectations with respect to the system, and ideally ensures that all interests of other stakeholders are also protected. A use case is written in the form of a scenario type, which captures a set of scenarios, where each scenario (instance) describes from start to finish a valid and distinct performance of the use case.

The definition is the one used by the ANZAC use case framework. It consists of four sentences, which each describing a different aspect of a use case. The first sentence describes the building blocks of a use case and the rules for putting them together. It takes into account the different ways a goal of a stakeholder can be related to an interaction or a system-initiated action, as stated in section 3.4.5.1. The second sentence states that a use case is a black-box description, and it describes the driving force behind a use case. It highlights that the use case is coupled to the goal of a stakeholder and that it describes the path that leads towards that goal. Furthermore, note that, according to the definition, a stakeholder is not required to interact with the system to achieve its goal. The third sentence states that a use case also makes mention of the different expectations that the stakeholder may have with respect to the goal it has against the system, and it reveals that a use case ideally conforms to the constraints of the interests of other stakeholders. The fourth sentence describes the form of a use case by providing some insight into the meaning of a scenario type.

In summary, a use case describes a collection of potential scenarios that arise under the circumstances in which the stakeholder's goal is reached and not reached. Thus, a use case will describe all possible paths to success and failure. The use case should not only describe the stakeholder's intentions against the system, but also the system's responsibilities with respect to these intentions. And likewise, when the system has a goal against an actor, then the use case is described in terms of the system's intentions with respect to the actor, and the actor's responsibilities with respect to these intentions. Furthermore, the use case should be rationalized with respect to the interests of the stakeholders.

3.5 Construction of Use Cases by Eliciting Actor Goals

Constructing use cases based on stakeholder goals, originally introduced by Kaindl [Kai95] and then Cockburn [Coc97] and later Lee et al. [LX99], has a number of advantages over ad hoc approaches or approaches that concentrate solely on gathering tasks performed by the system. First, a goal-based approach puts users and other stakeholders in the right frame of mind for identifying what services they require of the system, because user goals are what drives the use of the system. Second, explicit description of goals provides a means for justifying and negotiating system functionality, promoting a description of requirements that is likely to be more realistic, verifiable, and non-conflicting. Finally, goals can be defined in a hierarchy such that the goals of a stakeholder can be related in terms of composition/decomposition—a simple mechanism for a scalable description that promotes traceability between use cases. Furthermore, a hierarchy of goals intuitively provides support for evolution management because higher-level goals define more stable concerns [DLF93].

A use case that lies higher in the goal hierarchy provides the context and motivates the need for the use cases lower in the hierarchy, i.e., going up the hierarchy answers the question: "why do I have this goal", and going down the hierarchy answers the question: "how can I accomplish this goal". A use case at a given goal level identifies sub-goals that compose to form its goal. Each lower-level goal can be described by a lower-level use case. Use cases that describe lower-level goals should not be dependent on the higher-level use cases that use them; in that way, (future) reuse is more likely.

The name given to the use case should indicate the respective goal level of the use case. For example, given a use case for an elevator control system that describes the state of affairs surrounding a user taking the lift from source to destination floor, an appropriate name for the use case would be something like "take lift from one floor to another" and not "press floor button", nor "get to work". The goal "press floor button" would be described by a use case that would be lower down the goal hierarchy of the user, and the goal "get to work" would not be an appropriate use case for this system, because it is not grounded in the problem that the system is to help solve. Note that it would nevertheless be important for the higher-level use cases of the system, such as "take lift from one floor to another", to take into account the different expectations and interests that the users of the lift may have, e.g., for getting to work. Analysis of these expectations and interests could lead to special demands on the lift that take into account users coming and going to work at certain hours of the day, e.g., on weekdays, from 8-9 am lifts should be on the basement floor, if they have no requests to service, and from 5-6 pm they should be at the main office floor.

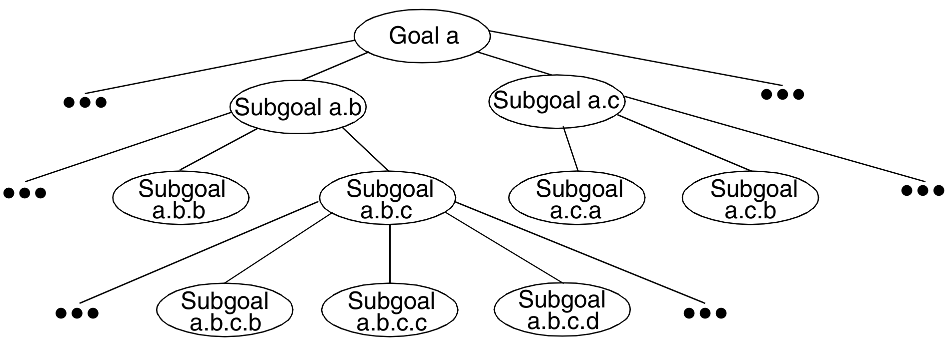

Figure 3.1: Goal Hierarchy

A goal hierarchy is built for each type of stakeholder (a tree with possibly common sub-trees, i.e., sub-goals can be reused between higher-level goals). Intuitively, a use case can be defined for each goal in the hierarchy. Use cases defined according to a goal hierarchy can, therefore, be related to other use cases in terms of their positions in the hierarchy. Figure 3.1 displays a sketch of an example goal hierarchy. A goal is achieved by the completion of all its sub-goals. Thus in a similar way, a performance of a use case is successful if each sub-goal is successfully accomplished. Unfortunately, the goal hierarchy shown in figure 3.1 does not express the variational aspect of use cases, i.e., it does not state the effect that sub-goal failure has on the goal, whether there are alternative sub-goals, and the order sub-goals are performed.

3.5.1 Representing Use Case Relationships

To show the variational aspects of a use case, it is necessary to show which sub-goals (i.e., sub-use cases) are alternatives, optional or repeated, the order sub-goals are performed, and which sub-goals can be performed in parallel to other ones. Ryser and Glinz [RG01] propose a diagram called a dependency chart, which graphically depicts such relationships between use cases. UML also provides a diagram for visualizing relationships between use cases, and between use cases and actors [Omg01]. Unfortunately, the relationships defined by UML are quite restrictive, which comes from the legacy of Jacobson's approach, where he was of the opinion that any kind of composition association between use cases would "support functional decomposition, which would lead easily to a functional rather than object-oriented structure" [JC95]. These relationships prove to be one of the most controversial parts of the UML specification, evidenced by the numerous publications on the subject [Gra94, Hen98, HK98, Fir99, Lil99, RS99, SG99, Sim99, VS99, Gli00, WG00, MOW01, Ste01].

Based on Cockburn's work, I define, as part of the ANZAC use case framework, an informal, textual calculus that is used within use case descriptions to define these rules/relationships; it is detailed in section 3.9. Thus, rather than showing these relationships graphically, I prefer to provide this information textually in use case descriptions, because one can provide a common notation for relationships between use cases and within use cases. It may however be interesting to generate diagrams, such as dependency charts, from the textual descriptions; this would allow one to get the benefit of a focused overview that visual representations are able to offer. However, the details of such a tool are out of the scope of this work.

3.5.2 Stakeholder Goal Levels

By conceptually defining a goal hierarchy for use cases, different use cases will have different goal granularities. Cockburn identified three principal goal levels: summary, user-goal, sub-function [Coc00]. An ANZAC use case can be classified as one of these three goal levels.

- **Summary level** use cases describe a high-level goal of the stakeholder that unifies lower-level goals and puts them in context. It usually describes a goal that takes multiple sessions to go from start to completion, each one taking place at different points in time.
- **User-goal level** use cases describe a goal that the stakeholder has in trying to get work done in using the system. A user-goal level use case is a single, discrete, complete, meaningful, and well-defined task of interest to the driving actor. The use case is usually situated in one place and performed in a single session.
- **Sub-function level** use cases describe sub-goals of the driving actor that are required to carry out its user goals (the level above). They are low-level and need to be justified from an efficiency point of view, either for reasons of reuse and/or necessary detail.

Clearly, three goal levels may be insufficient to describe certain software systems. In such cases, summary level use cases may be sub-goals of a coarser-grained summary level use case. And at the other end of the spectrum, a sub-function level use case may be a super-goal (the inverse of a sub-goal) of finer-grained sub-function level use cases. User-goal level use cases define the baseline for development, because goals at this level generally define a discrete, complete, and meaningful unit of system capability, which can be prioritized and delivered incrementally. Furthermore, the informal style of writing ANZAC use case descriptions is best utilized at a high-level of abstraction, where the ANZAC approach judges that it is usually not economical to push use cases far beyond the user-goal level (since the ANZAC specification is used to go into the “details”).

Even in the worst case, i.e., decomposing use cases as far as possible, it is important to note that a use case can not be smaller grained than a single interaction or system-instigated action, and a use case must always be triggered by some kind of (external or internal) stimulus. In other words, one cannot functionally decompose use cases past the boundaries of the finest-grained interaction or system-instigated action. Therefore, by focusing use cases on stakeholder goals rather than on tasks performed by the system, one can avoid, to a certain extent, the effects of endless functional decomposition, and the infamous maintenance nightmare.

3.5.3 Difficulties in Goal Elicitation

There may be functional requirements that are at least difficult to associate to the goal of a stakeholder, i.e., it may be difficult to pinpoint which stakeholder has the goal that justifies a particular functional requirement. For example, a functional requirement for a banking system is to send out monthly statements. So a question one could ask is whose goal drives the use case for sending out monthly statements? Is it the goal of the client to receive a monthly statement or is it the goal of the bank? Probably the most appropriate answer to this question is to say that it is the goal of the bank, because the bank must fulfill the account contract that was agreed upon at the time of opening the account (i.e., assuming the account contract states that the client will receive monthly statements), and since it is not necessarily the goal of a client to receive monthly statements.

Such situations can arise in the development of enterprise systems, where business policies are either outdated or they are clouded by intricate rules. In general, once requirements can not be easily traced back to stakeholder goals then usually the business process of the enterprise should be re-evaluated. This can lead to re-engineering the business process of the enterprise in question[1] [HC01]. The implications and activities involved in business process reengineering are beyond the scope of this work.

3.5.4 Defensive Goals

Usually, if one thinks of a goal that a stakeholder has against the system, what comes to mind is a goal that adds value to the stakeholder in some way. However, some goals can be defensive in nature, by protecting the interest of a stakeholder. For example, in a university book borrowing system there may be a requirement for warning letters to be sent at the start of each week for all books that are overdue. The goal that drives this functional requirement

1. I acknowledge Gil Regev and Alain Wegmann for helping me to clarify this point.

is the university's one; it has two interests that it would like to protect: avoid lost books, and enforce book borrowing privileges. Note that in this case the stakeholder (university) with the goal against the system (warn members with overdue books) does not explicitly trigger the use case. Rather the use case would describe a system-instigated action.

3.6 Categories and Personalities of Actors and Stakeholders

Jacobson originally categorized actors into two kinds: *primary* and *secondary* [JCJ92]. He defined a primary actor as the one who receives value from the use case; in Cockburn's terminology this is the actor who has the goal against the system. Furthermore, it is possible to have a use case where the primary actor is not involved in any interactions of the use case. Providing another example (in addition to the "send out warning letters" use case), consider an enterprise system that is required to perform surveys of the workers at random moments throughout the year. At the end of each year, it is required to create a report, by compiling all the surveys, and storing it. With regard to the driving goal, it is the goal of management to obtain a yearly report that details whether the workers are satisfied with their jobs. Grounding this goal in the context of the system, one could imagine three sub-goals (user-goal level): take worker survey, produce end of year report, and print report. Focusing on the first two use cases, we can see that management does not even figure as an actor, because they are both system-instigated actions that deliver their result to (probably) a database. Clearly management is the stakeholder with the goal, but it is an *off-stage actor*[1]. Therefore, a primary actor may well be an off-stage actor.

In summary, a primary actor is an actor whose goal is the focus of the use case. The primary actor may initiate the interaction with the system, it may have an intermediary interact with the system on its behalf, or may have the system trigger an action automatically. Furthermore, the primary actor has the leading role in the use case. In contrast, a secondary actor is defined as an actor that plays a supporting role in a use case—support required by the system to fulfill its responsibilities. Note that it is possible to be a primary actor in one use case and a secondary actor in another, i.e., the scope of the primary/secondary category is that of the use case in question.

Depending on the kind of interaction that an actor has with the system in a use case, actors can have four different personalities: *initiator*, *server*, *receiver*, and *facilitator* [AM01]. An actor with an initiator personality is defined as an actor that initiates an interaction with the system. An initiator is usually a primary actor in the use case[2]. An actor with a server personality is defined as an actor to which the system makes an explicit request, i.e., it provides a service to the system (within the context of a use case). An actor with a receiver personality is an actor that receives a notification from the system (in a use case).

1. A term coined by Cockburn [Coc00].
2. It may also be a secondary actor that sends a notification (implicit request).

An actor with a facilitator personality is an actor that supports another actor's interaction with the system, or that supports the system's interaction with another actor (in a use case). A facilitator is usually a secondary actor. Note that actor personalities may even have multiple personalities in a single use case.

3.6.1 Scoping Secondary Actors

Secondary actors play a supporting role in a use case and without them the system would not be able to fulfill its responsibilities. However, secondary actors that have facilitator and server personalities will often not appear in use cases until the lower goal level use cases are elaborated.

Most summary level use cases and some user-goal level use cases do not mention secondary actors with facilitator or server personalities, because the interactions with these actors are abstracted away. Instead, the use case concentrates on the intention of the primary actor, relating this intention to some responsibility of the system. For example in an ATM system, we could imagine that all PIN validations are to be performed by a centralized authentication authority. Looking at the use case that describes the user's goal to withdraw money using the ATM, we would probably see something like, "System validates PIN". In this case, the interaction between the system and the secondary actor (server personality) would be transparent. It would require a lower goal level use case to make it clear that the system delegates PIN validation to the centralized authentication authority.

It may also be the case that the nature or type of facilitator is unimportant. In this case, the facilitator actor would be given a role name that generalizes all possible concrete realizations. For instance in the same ATM system, there is a facilitator between a customer and the system that allows customers to identify themselves with the system. Given that there are many possible ways that customers could identify themselves with the system, an appropriate actor type for the secondary actor would be something like identification facilitator. This is in contrast to a specific actor type like "retinal scanner", "keypad", or "fingerprint recognizer". Remember, according to the definition of actor, an actor is just a composition of roles, thus "identification facilitator" could be seen as the composition of the roles: "retinal scanner", "keypad", or "fingerprint recognizer", etc. Note that such an approach would only be appropriate if a particular realization has not been mandated as a project constraint, e.g., an ATM built specifically for James Bond would require strictly retinal scan identification.

3.7 Tying Use Cases to Extra-Functional Requirements and Project Constraints

Use cases describe the functional requirements of a system under development by illustrating all the different possible ways that the system could perform work. They do not however capture all extra-functional requirements.

Extra-functional requirements, also known as non-functional requirements, capture required properties or qualities of the system. They differ from functional requirements because they usually state how services have to be provided, rather than which ones. Failure to meet them can make the system unusable, where a missing function may just degrade the system. Furthermore, some extra-functional requirements can have a large effect on determining the software architecture of the system: two systems with the same use cases but very different extra-functional requirements may need very different solution architectures.

Extra-functional requirements often relate to the system as a whole rather than to a single feature, and therefore they can be difficult to associate to any particular use case or even groups of use cases. For instance, a system may have the following reliability requirement: the system must function normally, without a reboot, for at least 3 working weeks. Clearly, this requirement would be difficult to associate to any particular use case. However, there are other extra-functional requirements that can be directly related to a use case. For instance, a system may be required to respond to a particular kind of user request within a time limit while concurrently processing up to a certain threshold of other user requests. In this case, the extra-functional requirement could be placed within the use case that describes the corresponding user request. This highlights the fact that it is possible to describe some timing constraints and some aspects of concurrent behavior in use cases. In section 3.9, I explain how these aspects may be expressed in use cases.

Sometimes extra-functional requirements require the system to react to certain situations. For example, in an enterprise system, there may be a security requirement that requires any user that performs a suspicious task to be automatically logged off. This requirement may touch many different use cases (all those use cases that could involve suspicious tasks performed by a user), requiring each use case to log off a user under those circumstances.

Project constraints are also important because they define the infrastructure that is already in place for a project or the rules that must be obeyed by the project. These constraints do not depend on the system, but they often clarify the extremities of the system, i.e., they define those things that the future system will interface with. Project constraints along with domain properties, design constraints, project issues, etc. are at best indirectly described by use cases. This information is not usually documented by use cases; nevertheless the relevant information can be referenced by the use cases, e.g., business rules can be centralized in another document [Ros97] and referenced from the relevant use cases [CL01].

The point to make is that use cases are an excellent tool for eliciting functional requirements, and rationalizing them with respect to stakeholder interests and concerns, but use cases themselves do not capture all information needed to define the problem. The complete Stakeholders Contract for a system would consist of the combination of all these requirements and constraints. However, it is often difficult to get a homogenous description of all the kinds of requirements and constraints.

3.8 Managing Change in Use Cases

Change is always difficult to predict and use cases are subject to change like any other work artifact. Changes to requirements may or may not affect use cases. Roughly speaking, requirements can be categorized into different groups, where each group is subject to a different amount of change over time. At one end of the "change scale" are those requirements that are mandated and perceived to be stable, and at the other end of the scale are those requirements that are stakeholder preferences and may conflict with the interests of other stakeholders. It is important to take into account the (perceived) stability of requirements when writing use cases. Recognizing whether a requirement is a preference or a mandated constraint requires some skill in interviewing stakeholders. For instance, using a non-software example, a client could state that he requires a purple sports car to get him from Sydney to Brisbane, but if you were to push down to what is really necessary then it may become clear that the essential requirement is to travel from Sydney to Brisbane on ground transportation.

3.8.1 Reducing the Effects of Change

Abstraction is a useful tool for reducing the effects of change. Thus, use cases that are described at a high-level of abstraction are theoretically less subject to change compared to use cases at a lower-level of abstraction. Change can also be managed by the layered hierarchy of use cases. Lower-level use cases should not depend on higher-level use cases, and higher-level use cases can abstract away those interactions that are perceived to be subject to change by encapsulating them in separate use cases—in this way the higher-level use case does not depend on the details of the "included" lower-level use case.

3.8.1.1 Relationship of Use Cases to User Interface and Design Details

One commonly cited observation is that the user interface is, in general, more volatile (i.e., subject to change) compared to the application logic of the system. For this reason, and the added complexity of describing user interface interactions, I propose to avoid the description of those aspects that capture the requirements for the user interface in ANZAC use case descriptions. This point is aligned to the suggestions of Cockburn [Coc00], Constantine and Lockwood [CL99], and Anderson and Fertig [AF00].

For instance, a business for whom an internet ordering system (see section 3.9.2) is being built may have a screen layout and navigation style that is required to be observed by the development team, e.g., check-out modes are shown as radio buttons and mode confirmation is linked to an "okay" button. Even though these are valid requirements, it would still not be recommendable to write the following phrase in a use case: "user clicks radio button to make choice of check-out mode and clicks on okay button to confirm choice". Constantine and Lockwood have further arguments for avoiding all discussion of the user interface in a use case [CL99, CL01]. Note that there are effective variants of use cases that can be used to describe user interface details, see [WM99, Red96].

All non-mandated technology and design constraints are also excluded for similar reasons, and only those mandated constraints that respect the abstraction level of a use case should be included. For example, a use case that details identification for an ATM system will no doubt be dependent on concepts like ATM cards and PINs, but a use case at a high goal level will probably not mention ATM cards and PINs. The reasoning behind including mandated constraints is that they are generally more stable than non-mandated ones and thus are less likely to change. They are nevertheless important for defining the constraints on the possible solutions.

3.9 Use Case Descriptions

The form of a document does make a difference to the readability and general enthusiasm that people have towards using a particular work artifact of software development. Therefore, it is important to choose an appropriate presentation structure, style, and level of detail.

The success of use cases is arguably founded on their informal, scenario-oriented form that is accessible to non-technical as well as technical members of a project. That is, due to their simplistic appearance and their informal style, use cases are not a daunting work artifact even for project members that have little experience with software requirements specification. Furthermore, use cases are often written in natural language, which proves to be extremely expressive and malleable to the description of all possible contexts, yet accessible to all.

The representation that is proposed by the ANZAC use case framework is informal narrative that is structured with a number of clauses. The various clauses of the use case act as a template that is filled in, and the structure of the scenario description makes use of a calculus to define the rules that need to be observed.

Moreover, the structure of the description of the scenarios is flattened into one "main" scenario that has variation points. This flattened-scenario style of description is preferred over a branching condition description, e.g., using if-then-else like statements, following the hypothesis that non-technical readers of use cases have more difficulties with the branching style (and particularly nesting of branches) than with the flattened scenario style [Jac95, Coc00, Lil99]. This follows from the requisite that the style of a use case makes as few assumptions about the background of a potential reader of a use case as possible. Also, since, with the flattened-scenario style, the main scenario (should) expresses stable requirements, only variants are added or changed, whereas changes to a branching style description will unavoidably alter the main flow of the use case.

3.9.1 Representation for Use Case Descriptions

In practice, use case descriptions can have varying degrees of formality in content and form. Some use case descriptions will be just a casual story-telling description, others will be full-fledged descriptions that include an assortment of secondary information. The type of template and style one chooses reflects the nature of the project and/or the preferences of the development team.

The style and template of use case description that I propose as part of the ANZAC use case framework is a slightly more formal version of Cockburn's fully dressed use case template [Coc98]. This means that the framework defines only a single style and use cases produced in accordance with it are used in a reasonably formal setting. Note that the formality of the style does not affect the fact that use cases are refined by iteration.

This style of description is written in a single column, and it shows a series of numbered scenario steps, where a step defines a significant stage in a use case. A step has zero or more variation steps. A step is written in natural language with a simple grammar: articles and adjectives are excluded (when reasonable to do so), and each step is an active-verb phrase. Informal comments can be placed between steps to provide additional information, commonly used to indicate repetition and optional steps.

The set of scenarios defined by an ANZAC use case are described in two sections of the use case description called the *main success scenario* clause and the *extensions* clause. The main success scenario clause defines the most probable and stable (with respect to change) scenario that leads to a successful performance of the use case. The extensions clause defines all other possible scenarios that could occur, both where the use case is successfully performed and otherwise. It defines the other scenarios by the use of variation points that indicate additional and alternatives flows to the main (success) scenario. Thus, an extension step always refers to a step or a number of steps in the main scenario.

An extension step might correspond to regular behavior, exceptional behavior that is recoverable, or unrecoverable erroneous behavior. If an extension step takes place in addition to the respective main step, then the following notation is used:

```
<step no. range>||
```

For instance, 2|| would indicate that it occurs in addition (possibly concurrently) to step 2 of the main scenario under certain conditions. If an extension step occurs as an alternative, then the following notation is used:

<step no. range> <alphabet character>

For instance, 2a would mean an alternative to step 2 in certain situations. For both parallel and alternative extension steps, the point at which the extension step rejoins the main scenario needs to be clearly stated, so as to guarantee that all possible scenarios of the use case are complete and well-defined. There are four possible ways that an extension step can finish:

... use case ends in failure;

... use case ends in success;

... use case continues at step x;

... use case joins main scenario.

Note that the last option is used for extension steps that are performed in addition to a main scenario step (i.e., "||"). Also, note that the success of a use case is measured in terms of the primary actor reaching its intended goal; if the goal is not reached, then the performance of the use case fails.

Care needs to be taken that the third option (of the four ways to finish an extension step) is not misused, making a use case a labyrinth of jumps between the main scenario and the extension steps. For a justification, one only needs to make the analogy to the discouraged use of "goto" statements in programming languages [Dij68]. This possible problem area is a result of the choice for a flattened scenario style.

Extension steps define (implicitly or explicitly) the condition under which it is taken in addition or alternatively to the corresponding main scenario step(s). If the condition is explicit then the following notation is used:

<step no. range> <alphabet character> condition ':' step(s)

Steps in the extensions clause can be nested. In such cases, Dewey decimal numbering is used. For instance, 3a.2. would mean the second step of the extension step 3a.

The form of a use case in terms of the clauses that it contains and the contents of those clauses is described in figure 3.2.

Use Case:
The name of the use case. This is the goal stated by a short active verb phrase.

System:
The name of the software application under development.

Level:
The goal level at which the use case is being described: Summary, User-Goal, or Sub-Function

Intention in Context:
A statement of the intention of the primary actor or the reason for performing the use case, and the context within which the use case is performed.

Primary Actor:
The primary actor of the use case.

Main Success Scenario:
The numbered steps of the scenario, from trigger to completion.

Extensions:
The alternatives steps, and steps that occur in addition to the main success scenario.

Figure 3.2: Use Case Format

These clauses define a minimum template for constructing use cases. However, other clauses can be used. Here is a non-exhaustive list of other clauses that could be added:

Stakeholders' Interests - A list of stakeholders and their key interests that relate to the use case.
Precondition - What we can assume about the state of the system and environment before a performance of the use case.
Trigger - What event starts the use case.
Notes - Any additional noteworthy information.
Timing Requirements - A description of the frequency at which this use case performs and any performance requirements.
Other Concurrent Use Cases - A description stating which other use cases can perform concurrently to this one.
UI Links - Comments and cross-references to UI design guidelines and manuals.
Open Issues - A description of some unresolved issues that relate to this use case.
Version - Information that includes: use case number, date, author, reviewers, etc.
Priority - Information that includes the following ratings: business need, design difficulty, overall priority.
Secondary Actors - A list of all secondary actors for the use case.

3.9.2 An Example of a Use Case

Figure 3.3 shows a user-goal level use case called Buy Items—named after the goal of the primary actor, Shopper. The system of focus in the use case is an online ordering application. The main scenario defines 3 steps, each one a sub-use case, where the underline indicates a hyperlink to the corresponding sub-use case description (albeit symbolic in this example). The main success scenario clause starts with an introductory commentary on the use case, stating that the shopper can navigate back to previous shopping cart inventories. Iteration of steps in the main scenario is also described informally as a comment before step 1. Note that it is possible to describe iteration by using an extension step that joins the main scenario at an earlier step; however it is important to not over-use "jumps" between extension steps and the main scenario.

A consequence of describing iteration as part of the main scenario is that the main scenario defines not a single scenario but many possible ones. In this example, it covers all scenarios where a successful purchase was made without any sub-goal failures or any modifications to the shopping cart. The extension steps state all variations to the use case's main scenario. Each extension step is interpreted in the following way.

Step 1a is taken instead of step 1 (an alternative) if the shopper fails to find an item. The details of how the sub-goal "find an item" can fail are encapsulated by the corresponding sub-use case (hyperlinked). Thus, a consistent level of abstraction within a use case can be kept by using lower goal level use cases to elaborate the details. The body of the extension step (step 1a.1) states that if the sub-goal fails, the shopper is given the option to try again. The extension step joins the main scenario at step 1.

Step (1-2)||a means that the shopper may modify the contents of his/her shopping cart at any time that he/she is in the process of finding products to buy. In other words, the sub-use case Modify Contents Of Shopping Cart can be triggered and take place in addition to steps 1 or 2. The Modify Contents Of Shopping Cart use case describes the sub-goal of modifying the contents of the shopping cart. This extension step does not explicitly state the condition under which the extension step occurs; the implication is that the extension step is taken if the sub-use case is triggered, where the condition for triggering is defined within the sub-use case. The extension step rejoins the main scenario when it completes.

Step (1-3)a is taken instead of step 1, 2, or 3 (an alternative to any of the three) if the system determines that the shopper is no longer shopping with it. In the case that this scenario arises (i.e., this extension is taken), the system cancels the pending purchases, contained in the shopping cart. This extension step explicitly states the condition (the phrase before the ':') that defines the circumstance in which the main scenario between 1 and 3 inclusive is "overridden". It also defines the action that would result (step (1-3)a.1). The extension step results in use case failure, which means goal failure for the primary actor.

Step 2a is taken instead of step 2 (an alternative) if the shopper fails to add an item to his/her shopping cart. The details of how the sub-goal could fail is described in the sub-use case Add Product To Shopping Cart. As a result of failing to add an item, the system prompts the shopper to try again (step 2a.1). The shopper may choose to try again (step 2a.2) in which case the use case rejoins the main scenario at step 2, or abandon the item (step 2a.2a) in which it rejoins at step 1. These two options are themselves defined using the same principle (and lettering scheme) that is used to show an alternative to the main scenario, i.e., a main path is chosen (2a.2) and an alternative is defined using the extension step lettering scheme (2a.2a).

Step 3a is taken instead of step 3 (an alternative) if the shopper fails to check-out his/her shopping cart for a reason detailed in the sub-use case, Check-Out Shopping Cart. This extension step results in use case failure.

Use Case: Buy Items
System: Acme Online Ordering Application
Level: User-Goal
Intention in Context: The intention of the Shopper is to buy a certain quantity of item(s) on Acme's online shopping site. Acme requires that a Shopper place items in a (virtual) shopping cart before the items can be purchased. There are possibly many Shoppers ordering items at any moment in time.
Primary Actor: Shopper
Main Success Scenario:
At any time the Shopper may navigate to an earlier shopping cart configuration (wrt. items in cart) and continue shopping from that point.

The Shopper repeats steps 1-2, as many times as desired.

1. Shopper <u>finds an item</u> of interest using the system's search and browsing capabilities

2. Shopper <u>adds (a certain quantity of) a product to shopping cart</u>.

3. Shopper <u>check-outs shopping cart</u>.

Extensions:
1a. Shopper fails to <u>find an item</u> of interest:

  1a.1. System prompts Shopper to try again; use case continues at step 1.

(1-2)||a. Shopper <u>modifies contents of shopping cart</u>; use case joins main scenario.

(1-3)a. System determines that Shopper has left:

  (1-3)a.1. System cancels the pending purchase; use case ends in failure.

2a. Shopper fails to <u>add product to shopping cart</u>:

  2a.1. System prompts Shopper to try again.

  2a.2. Shopper tries again: use case continues at step 2.

    2a.2a. Shopper abandons the product: use case continues at step 1.

3a. Shopper fails to <u>check-out shopping cart</u>: use case ends in failure.

Notes:

All || steps are interleaved (i.e., not performed in parallel)

Other Concurrent Use Cases:

This use case can perform concurrently to possibly many other "Buy Items" use cases.

Figure 3.3: Buy Items Use Case

The "Other Concurrent Use Cases" clause highlights that a performance of this use case can overlap with possibly many other performances of the same use case. That is, it is possible that many shoppers can buy items at the same time.

I now highlight the flexibility that is possible with natural language descriptions by revisiting the introductory commentary in the Buy Items use case, which states that the shopper can navigate back to previous shopping cart inventories. This commentary is quite complicated to describe using the proposed extension step calculus. Clearly, natural language allows one to describe such behavior succinctly without the need for a language extension. The ability of use cases to succinctly capture "unusual" behavior also makes them more readable to stakeholders. It is often the case in writing use cases that one comes across new situations that may be difficult to encode using a certain calculus.

Another point to note about the Buy Items use case shown in figure 3.3 is its name. One could imagine that the name "shop for items" would also be suitable, because in fact the shopper's intention would most likely be to look around different stores finding the items that are best suited to his/her buying criteria, such as, best quality for price ratio, best features, best price, etc. There are two reasons why "buy items" is a better choice for the use case name. 1) It defines a stronger goal that is clear and straight-forward to describe, because it is clear what success and failure means if one has the goal of buying items, which is not the case with the weaker goal "shop for items". Also, from the viewpoint of the shopper (and Acme for that matter) progress is made by achieving the stronger goal, but it is not clear if progress is made in the case of the weaker goal. The definition of a user-goal level use case, see section 3.5.2, implies that the use case defines a meaningful and complete task. 2) The use case is only concerned with one online ordering system, i.e., the Acme Online Ordering System, and the name of the use case must state the goal relative to this system. The goal "buy items" is clearer than "shop for items", because shopping entertains the idea of visiting many stores.

3.9.3 Granularity of Use Case Steps

Ideally, each step of a use case is a sub-goal of the primary actor, which is expressed as a sub-use case. This would typically mean that each step would have an alternative, in the case that the sub-goal is not reached, and the alternative would be described by a step in the extensions clause. The Buy Items use case (figure 3.3) is an example of a use case where steps are defined in terms of sub-goals, and these sub-goals are described as separate use cases. However, there comes a point in the decomposition activity when a step is too fine-grained to be described as a separate use case. At this point, the sub-goal is described "inline". This often results in an interaction being broken up into multiple steps in a use case.

The sub-function goal level Check-Out Shopping Cart use case is an example of a use case that breaks up interactions into separate steps; it is shown in figure 3.4. It is referenced in step 3 of the Buy Items use case (figure 3.3) as a sub-use case, elaborating that single step by five main scenario steps. Similarly to the Buy Items use case, the system and primary actor are the same.

Use Case: Check-Out Shopping Cart
System: Acme Online Ordering Application
Level: Sub-function
Intention in Context: The intention of the Shopper is to finalize the purchase of the items within his/her shopping cart. All purchases at Acme require payment via a credit card.
Primary Actor: Shopper
Main Success Scenario:

The Shopper may change, by navigation, the mode of check-out or any of his personal, payment, or delivery details so long as step 5 has not occurred, otherwise it is too late.

1. Shopper requests System to check-out his/her shopping cart; System prompts Shopper to choose from two ways to check-out[1]: express or regular.

2. Shopper requests System to perform regular check-out; System prompts Shopper for personal, payment, and delivery details[2] and prompts Shopper to confirm purchase once details have been provided.

3. Shopper provides System with information and confirms purchase.

4. System validates that it has sufficient information on Shopper to perform check-out.

5. System debits Shopper’s credit card with total purchase price, it passes order on to Warehouse for delivery to Shopper, and it informs Shopper of successful completion.

Extensions:
(2-3)a. Shopper requests System to cancel the shopping cart order.

(2-3)a.1. System cancels the pending purchase; use case ends in failure.

(2-3)b. System determines that Shopper has left:

(2-3)b.1. System cancels the pending purchase; use case ends in failure.

2a. Shopper requests System to perform express check-out; use case continues at step 4.

4a. System determines that it does not have sufficient details about Shopper to perform the check-out:

4a.1. System prompts Shopper to provide the missing information; use case continues at step 3.

5a. System determines that credit card is invalid:

5a.1. System prompts Shopper to provide details of another card; use case continues at step 3.

5b. System determines that it is not possible to debit Shopper’s credit card at this moment:

5b.1. System informs Shopper of problem and to try again later.

5b.2. System cancels the pending purchase; use case ends in failure.

Notes:
[1] more information on check-out options is given in document xxx

[2] data formats and representation styles are detailed in document yyy

Timing Requirements:
Step 5 must be performed within 2 seconds.

Figure 3.4: Check-Out Shopping Cart Use Case

The shopper's goal described by the Buy Items use case is to finalize the purchase of goods with the system by checking out the items within the shopper's shopping cart. The use case describes three interactions between the shopper and the system (each driven by a sub-goal of the shopper): shopper initiates check-out, shopper indicates check-out mode, and shopper finalizes order.

In the main scenario, the first interaction is described by step 1, the second one by step 2, and the third one by steps 3-5. The reason the third interaction is broken into three steps is because 1) the system must validate that it has sufficient information on the shopper and this is a point that can fail, 2) the debiting of the credit card is a failure point, and 3) the shopper providing his/her details is used by an extension step (4a) to rejoin the main scenario. Each failure point in the main scenario is partitioned into a separate step to make the use case easier to read and follow. If this third interaction was to get any more complicated, it would become a good candidate for a sub-use case; this would seem reasonable also if one wanted to describe which secondary actor the system interacts with to debit the shopper's credit card and the protocol used between them.

Finally, the "Timing Requirements" clause was added to highlight the performance requirement that the debiting of the credit card be performed and the shopper informed of the result within 2 seconds.

3.10 Summary of the ANZAC Use Case Framework

In the previous sections, I provided an informal description of the ANZAC use case framework. In this section, I attempt to summarize this framework, bringing together the parts of the various sections to give a more holistic view of this framework.

Context and Scope:
The ANZAC use case framework is targeted for the development of reactive software systems. The targeted scope for the ANZAC use case framework is any software application that provides services largely to human actors.

Purpose:
The purpose of the ANZAC use case framework is to facilitate the job of stakeholders in understanding, negotiating and agreeing upon the (software) system requirements. The concordant goals of the stakeholders are documented by ANZAC use case descriptions. The aim is to obtain a set of use cases that defines the complete behavior of the system, without detailing the internal workings of the system.

Definition of Use Case:
An ANZAC use case is the specification of a non-empty, partially ordered collection of system-instigated actions and/or actor-instigated interactions between the system and one or more of its actors. An ANZAC use case describes, without revealing the details of the sys-

tem's internal workings, the system's responsibilities and its interactions with its environment as it performs work in serving one or more requests that, if successfully completed, satisfies a goal of a particular stakeholder. A use case expresses this stakeholder's need and its possible expectations with respect to the system, and ideally ensures that all interests of other stakeholders are also protected. A use case is written in the form of a scenario type, which captures a set of scenarios, where each scenario (instance) describes from start to finish a valid and distinct performance of the use case.

Approach:
The ANZAC use case framework encourages one to develop use cases by focusing on the goals that stakeholders have against the system. The result of this process is a goal hierarchy, where a use case defines each goal in the hierarchy. Following Cockburn [Coc00], each use case is classified as one of the three goal levels: summary, user-goal, sub-function.

Actor Classifications:
To better understand the context of an ANZAC use case, the ANZAC use case framework defines a number of ways to classify actors. Following Jacobson [JCJ92], actors are partitioned into two categories: primary and secondary, in a use case. A primary actor is the one whose goal is the focus of the use case. The primary actor may initiate the interaction with the system, it may have an intermediary initiate the interaction with the system on its behalf, or it may have the system trigger an action automatically. On the other hand, a secondary actor is defined as an actor that plays a supporting role in a use case—support required by the system to fulfill its responsibilities.

Following Armour et al. [AM01], actors can have four different personalities: initiator, server, receiver, and facilitator. An actor with an initiator personality is defined as an actor that initiates an interaction with the system. An actor with a server personality is defined as an actor with which the system makes an explicit request, i.e., it provides a service to the system. An actor with a receiver personality is an actor that receives a notification from the system. An actor with a facilitator personality is an actor that supports another actor's interaction with the system, or that supports the system's interaction with another actor. Note that actors may have multiple personalities, even within the same use case.

Form:
The representation of an ANZAC use case description is informal narrative (natural language) that is structured into a number of clauses. The various clauses of the use case act as a template that is filled in. Following Cockburn [Coc00], a use case is written in a single column and it shows a series of numbered scenario steps contained within the *main success scenario* clause and the *extensions* clause. A step is written in natural language with a simple grammar: active-verb phrase, where articles and adjectives are excluded (when reasonable to do so).

Style:
An ANZAC use case defines a set of scenarios. Following Cockburn [Coc00], these scenarios are represented using a flattened-scenario style. More precisely, the main success scenario clause describes the most probable scenario that leads to a successful performance of the use case, and the extensions clause defines all other possible scenarios that could occur, both where the use case is successfully performed and otherwise. An extension step might correspond to regular behavior, exceptional behavior that is recoverable, or unrecoverable erroneous behavior.

A step in the main scenario defines a significant stage in the performance of the use case, which has possibly many variation points. All extension steps define the place in the main scenario where they interrupt or occur in addition to the main scenario and also where they rejoin the main scenario. This information is described using a simple calculus that defines the ordering of steps. In addition, informal comments can be placed between steps to provide additional information, commonly used to indicate repetition and optional steps that are difficult to describe using the calculus.

Ideally, each step of a use case is a sub-goal of the primary actor, which is expressed as a sub-use case. This would typically mean that each step would have an alternative, in the case that the sub-goal is not achieved, and the alternative would be described by a step in the extensions clause.

3.11 Deficiencies of ANZAC Use Case Descriptions

In this section, I describe some of the limitations that ANZAC use case descriptions have for defining the Behavioral Stakeholders Contract, and I propose some possible ways to minimize some of these limitations. I also explain the traditional role that use cases can play in other development activities.

Limitation 1
One shortcoming of the ANZAC use case description for the purposes of defining the Behavioral Stakeholders Contract is that it restricts the specifier to existential rather than universal descriptions of properties and constraints of the system [DH99], i.e., they define scenarios not closed-ended behavior. This means that it is not possible to describe system-wide properties and constraints, or it is at least difficult to define them in a single place. For example, consider a system that is required to manage all pottery sales for a certain business. If there existed a rule stating that all pots made at Bishopdale are terracotta, then it would not be possible to define such a constraint with use cases. In some cases, it may be possible for the specifier to code such notions in each and every relevant use case, but clearly this is laborious, reduces the visibility of the invariant and can lead to inconsistencies, particularly in the face of changes to use cases.

Furthermore, a use case gives the developer only a partial view of the problem domain. Consequently, if care is not taken, inconsistencies between the domain vocabularies used in different use cases may result; for instance, different names could be used for the same concept or the same name for different concepts.

These limitations of ANZAC use case descriptions can be resolved by the introduction of a complementary and separate description that records all terminology, properties, and invariants of the domain that are deemed relevant to the problem in hand. Such a description of the problem domain would act as a reference for the use cases, where all the use cases for a particular system must be shown to conform to the corresponding description. Note that such a description specifies the pertinent aspects of the problem domain and is preferably devoid of any notion of the software application under development. In [SS99b], a proposal for such a description, called the Domain Class Model, is made. It uses a UML class diagram as notation and is derived from the object model of the Fusion object-oriented software development method [CAB+94]. Similar models can be found in a number of different development methods, e.g., [SM89, CY91, HC91, RBP+91, MO92, Dou99].

Limitation 2

Another shortcoming is that the style of writing alternatives in ANZAC use case descriptions is clumsy for systems that are highly state dependent, i.e., systems that have many alternatives for each state that it is in [Gli01]. For such systems, following Glinz's recommendations, a description that succinctly expresses state-dependent system behavior can be used to express those aspects that prove to be cumbersome to describe with a use case, e.g., using a UML state diagram [Omg01]. The combination of the two descriptions would provide a better means for documenting the functional requirements of this kind of system. However, care should be taken that one does not start specifying solutions rather than requirements [Pol97], and also because state machines are not necessarily accessible to non-technical stakeholders. Thus, it may be worth keeping a "hazy" but usable description, and clarifying these issues in the construction of the Behavioral Design Contract, i.e., an ANZAC specification.

Limitation 3

Another shortcoming of the ANZAC use case description for the purposes of defining the Behavioral Stakeholders Contract is that it is not possible to define the effects of composing features using ANZAC use case descriptions. Different features of the system may interact in unexpected ways, termed "feature interaction" in telecommunication literature [KK98]. An example of a undesirable feature interaction (taken from [Zav01]) could arise in a telecommunications system where the feature "music on hold" interacts with the feature "phone conference". For instance, three people are having a phone conference and one of the members puts his phone on hold, but he has the "music on hold" feature enabled, and consequently disrupts the other two members from communicating because of music being played on all three phones.

Due to their original definition, use cases are unable to describe feature interaction, because they are assumed to perform in isolation [Jac95]. This means that all forms of interference, including that due to concurrent behavior, can not be constrained nor specified by use cases. Again, injecting this information into the use case descriptions will undoubtedly make them complicated and overly technical. Thus, it may be worth deferring the description of this information, and clarifying these issues in the construction of the Behavioral Design Contract.

Limitation 4

Another shortcoming of the ANZAC use case description for the purposes of defining the Behavioral Stakeholders Contract is that it is not possible to capture all requirements of a system using ANZAC use case descriptions (as discussed in section 3.7). Consequently, to get the overall view, stakeholders are conceptually required to merge the various descriptions that document the different parts of the requirements and project constraints.

3.11.1 Making use of Use Cases in other Development Activities

I have described a kind of use case that can be effectively used to define the Behavioral Stakeholders Contract. In the context of software development, however, it is insufficient to just define the requirements; it is necessary to communicate these requirements to developers that are involved in design, implementation, testing, and maintenance. In practice, use cases are often used[1] for activities such as architectural design, component design, user interface analysis and design, testing, and maintenance activities.

Different development activities have different concerns and therefore will make use of different aspects of the use cases. In this chapter, I have been careful to define use cases with those characteristics that are suited and advantageous for defining the Behavioral Stakeholders Contract for software systems, and I have tried not to compromise those characteristics by biasing use cases towards other activities where use cases could be used, particularly design.

1. I try to avoid the term ‘input’ since it gives one the impression that one should produce the work product(s) for the phase and then “throw it over the wall” to the team that is occupied with the next phase.

Chapter 4:

Specifying the Behavioral Design Contract

"In the realm of dynamic behavior, there is a particularly dire need for approaches that are sufficiently clear and well-structured to enable designers to capture their thinking in a coherent and comprehensive fashion."
David Harel [pp. 11, Har92].

This chapter introduces the part of the ANZAC approach that involves defining the Behavioral Design Contract, called the ANZAC specification. It provides an overview of the three different models of the system that form an ANZAC specification: Concept Model, Operation Model and Protocol Model, and it discusses their relationship to ANZAC use case descriptions.

4.1 Introduction

One aspect of reactive systems that makes them particularly difficult to describe is their ability to behave differently to the same stimuli at different points in time. A reactive system reacts to stimuli from its environment according to its current situation, and therefore, in describing the Behavioral Design Contract for a reactive system, one needs to carefully consider this aspect. This chapter involves introducing the part of the ANZAC approach that deals with defining the Behavioral Design Contract.

I define the Behavioral Design Contract in the following way:

> *Behavioral Design Contract – is a specification of the system responsibilities in servicing all possible requests that it can receive over its lifetime, where each responsibility can be justified against the interests of the stakeholders. It is expressed in a form that is accessible to designers and provides the development team with sufficient information about the software system in focus without prohibiting plausible solutions.*

This chapter presents the ANZAC specification, which is ANZAC's realization of the Behavioral Design Contract, establishing the second (and last) work artifact of the ANZAC approach. An ANZAC specification is used to precisely describe the system's behavior using three distinct but complementary models. The ANZAC approach ensures that the system responsibilities described by an ANZAC specification have a correlation to the interests of stakeholders, due to refinement relationships that are formed between it and ANZAC use cases.

This chapter is structured in the following way. In section 4.2, I describe some limitations of ANZAC use case descriptions if they were to be used to define the Behavioral Design Contract in addition to the Behavioral Stakeholders Contract. This discussion is used to motivate the need for a different work artifact for the specific purpose of defining the Behavioral Design Contract: the ANZAC specification. In section 4.3, I provide an overview of the ANZAC specification. In section 4.4, I briefly discuss the ANZAC mapping technique, which is used to map ANZAC use case descriptions to an ANZAC specification.

4.2 Wanted: Behavioral Design Contract

The allure of a single formalism for describing both the Behavioral Stakeholders Contract and the Behavioral Design Contract is strong. A single formalism above all offers homogeneity that when combined with refinement offers seamless development over the two contracts. However, it is important to keep in mind that the Behavioral Stakeholders Contract and the Behavioral Design Contract address quite different concerns, and therefore a single formalism may not be ideal for both purposes.

In this section, I highlight five characteristics of ANZAC use case descriptions that make them less suited for capturing the Behavioral Design Contract, and I point out the key concerns that need to be addressed with respect to the Behavioral Design Contract. For each issue, I discuss some of the merits of the ANZAC specification for addressing that particular concern, and hence expose the ANZAC specification as an appropriate means for defining the Behavioral Design Contract. The general conclusion that can be gained from the discussion is that there is unlikely to exist a formalism that could be optimal for defining both the Behavioral Stakeholders Contract and the Behavioral Design Contract.

Issue 1: The Behavioral Design Contract should describe the system responsibilities in servicing all possible requests over time, whereas the Behavioral Stakeholders Contract should describe the goals of stakeholders that constrain or induce system behavior.

ANZAC use case descriptions focus on the state of affairs that surround the accomplishment of a particular goal of a stakeholder, and they highlight the different expectations and interests that the stakeholders have with respect to this goal. Although this point of view gives context to readers, a description that is focused on these aspects will often provide superfluous information about situations that cannot be detected or acted upon by the system, which can cause confusion during design.

Thus, a description that expresses strictly only the behavior of the system in terms of verifiable responsibilities would allow developers to concentrate on those aspects of the problem that are pertinent to the design of the system, and hence it would avoid possible confusion and reduce "noise" in the description.

However, if one would rework use cases with this need in mind, then a crucial (and hard-fought) product of requirements elicitation would be lost, i.e., the justification for the system actions and the stakeholders' expectations and interests with respect to these actions. Furthermore, the inclusion of such information allows one to have greater confidence that the resulting system will be appropriate and adequate, because (reified) actions can be directly justified.

Realistically if use case descriptions were to be also used to define the Behavioral Design Contract as well as the Behavioral Stakeholders Contract, then it would be necessary to separate the use case descriptions for the two contracts. This would mean two separate work artifacts and the additional maintenance overhead of two artifacts compared to one.

An ANZAC specification, on the other hand, is targeted specifically at the description of the system responsibilities in servicing all possible requests that it can receive over its lifetime. It offers a precise, operational view of the system that is suited to the specific concerns of the development team in designing the system.

Issue 2: The Behavioral Design Contract should be in a form that facilitates design activities and provides a precise and unambiguous description, whereas the Behavioral Stakeholders Contract should be in a form that is accessible to stakeholders and facilitates the capture of informal, real world needs.

The fact that one can make no assumptions about the backgrounds and technical competencies of the various stakeholders makes it very difficult to go past natural language for defining the Behavioral Stakeholders Contract, because natural language is "the lowest common denominator" and is familiar to everyone involved. Also, natural language has a distinct advantage over formal languages for this purpose in that it offers maximum expressive power to its writers, allowing them to capture the complexities and subtleties of the informal real world. The importance of this last point should not be underestimated, as demonstrated in Chapter 3 (ANZAC use case descriptions).

In design activities, the concern is different. What is needed is a precise, concise, and unambiguous description of system behavior. For this purpose, the natural language descriptions of ANZAC use cases have a number of disadvantages. The expressive power of natural language makes it difficult to impose a consistent level of detail and precision across use cases and even within use cases, e.g., one step can be vague and figurative while another one can be exact and literal. Also, natural language is prone to ambiguity, which leads to uncertainty in design activities. And, there are only limited forms of analysis available for natural language, which means that tools for use case descriptions are limited to approximately the same analysis capabilities as word processors, i.e., spelling and basic grammar.

Furthermore, it can sometimes be difficult to determine whether the behavior of the system described in two separate use cases constitutes the same or slightly different functions of the system. This is due to the loose nature of natural language description, i.e., one use case could describe the behavior of the system in one way and another use case could describe the same behavior in another way. This is a consequence of the fact that different goals may have common sub-goals. Clearly, such a situation can lead to uncertainties in design and result in redundancies and inconsistencies.

Thus, a description that uses a language that is less prone to ambiguity, that facilitates a consistent level of detail and precision, and that is amenable to formal analyses would better facilitate design activities, assuming the formalism chosen was accessible to developers.

To this end, an ANZAC specification offers a language that has a well-defined syntax and semantics, which means that missing, ambiguous or inconsistent information can be found more easily. It uses a mixture of visual and textual notations, which are targeted at being comprehensible to developers who do not necessarily have a strong background in mathematics. And, it is amenable to tool support.

Issue 3: The Behavioral Design Contract should offer the designer all information that can facilitate the design of a concurrent system (reactive systems being often concurrent), whereas the Behavioral Stakeholders Contract should avoid any details that could complicate the specification, and risk exclusion of some stakeholders.

Concurrent behavior is at best only superficially described in an ANZAC use case description. More precisely, a use case can include information on which other use cases can perform concurrently to it. However, it is not possible to state the effects or consequences of performing use cases concurrently. This is because use cases are assumed to be free from interference—according to Jacobson “use cases are atomic, serialized slices of a system” [JC95]. Nevertheless, this level of expression means that use cases do not become too technical and complicated, which would make them less accessible to some stakeholders.

However, from a designer’s point of view, use cases do not provide a means to describe correct system behavior in the face of concurrency. And, taken literally, the design of a concurrent system that ignores conflicts among performances (i.e., executions) of use cases would almost undoubtedly lead to a flawed system. Thus, a description that explicitly describes which system features may perform in parallel to others and the necessary synchronization requirements for avoiding undesirable interference between these features would greatly assist the design of a concurrent system.

As an historical note: policies for dealing with interference are traditionally left implicit in requirements documents, often perceived as design decisions—use cases follow this tradition. Notwithstanding, the need to avoid certain situations that could result from interference are often constraints on the problem in hand. For example, consider a system that is used to manage ticket sales and that allows the concurrent sale of tickets for given spectacles. Typically, a specification of this system would make no mention of what should happen if two requests for a single ticket are made at the same time. However, there would probably be a constraint stating that a ticket can only ever be sold to one buyer, which would implicitly provide some insight into resolving what should happen when two requests are made for a single ticket. Providing such explicit concurrency and synchronization rules would assist developers in design activities, and if care is taken, such information would not lead to overspecification.

To this end, an ANZAC specification offers explicit support for specifying (inherent or required) concurrent behavior of the system, and for specifying synchronization constraints between concurrent operations. Also, care has been taken with the design of the ANZAC specification that the language and guidelines help specifiers avoid overspecifying concurrency constraints.

Issue 4: The Behavioral Design Contract should facilitate the scheduling of (incremental or parallel) work tasks.

In a use case, often the goal of the primary actor is delivered for a set of features and often features will cross multiple use cases. However, design is normally scheduled in terms of features. Therefore, in using ANZAC use case descriptions in design activities, it may be unnecessarily complicated for designers to keep track of features. Note that it would nevertheless be possible to decompose use cases until the level of a single system feature. However, this requirement may be too restrictive or too expensive for all development efforts.

An ANZAC specification, on the other hand, explicitly describes system features (aka. system operations), offering a direct basis from which work can be scheduled and progress tracked.

Issue 5: The Behavioral Design Contract should not bias possible solutions, particularly ones related to functional decomposition.

It has been well documented that use cases are constructed in a manner that matches closely to functional decomposition [Gra94, CF98, Kor98, Hen98, Fir99, Kov99, Sim99], where a use case decomposition hierarchy has many similarities to a functional decomposition hierarchy[1]. The problem therefore is that in the hands of inexperienced developers, use cases can lead to naïve object-oriented designs that match the functional decomposition hierarchy of use cases (in the case that object-orientation is used). This is really not an inherent problem of use cases as the documented functional requirements, but rather a problem that has come about because use cases have been championed as object-oriented [Jac95], where many people are misled by the numerous claims of seamless object-oriented development with use cases, e.g., [JCJ92, RS99].

On the other hand, an ANZAC specification is not described in a manner that can be easily associated to functional decomposition. In fact, it expresses the functionality of the system with a flat list of feature descriptions, which matches nicely with some current work on feature-driven design, spearheaded by Coad [CLD99, PF02].

4.3 ANZAC Specification

Having motivated the need for a specification that better matches the concerns of the Behavioral Design Contract, in this section, I give an overview of the ANZAC specification. I introduce the notion of view, and I explain the three models that make up an ANZAC specification, which each capture a different view of the system. I then provide some details on how an ANZAC specification models communication between the system and the actors in its environment, and describe the way that the system processes incoming communications, according to an ANZAC specification.

4.3.1 Three Views of the System

In communicating to the designers the way that the future system is required to behave, it is important that the specification is not only precise and described at an appropriate level of detail, but that it is expressed in a way that clearly presents the pertinent behavioral aspects of the system.

Projection allows one to partition the details of the system into views according to the aspects that are deemed to be important and that are reasonably orthogonal. Projection when combined with abstraction allows one to focus a specification on describing only the details of the system that are judged important. Views in a specification describe a particular aspect of the system in isolation, where the complete model of the system is defined by the combination of the views. An advantage of using views in a specification is that each aspect captured by the view can be constructed and managed separately. Also, views allow a more lucid specification due to a separation of concerns. However, if views are not sufficiently orthogonal, then the resulting specification may present unnecessary redundancies and lead

1. Note that a use case can not be decomposed beyond a simple interaction. Thus, a more accurate term would probably be “interaction decomposition” in the case of use cases.

to difficulties in composing the views during the formation of the overall model of the system.

In describing the behavior of a reactive system, it is important to capture the way that the system reacts to stimuli over time. Note that stimuli received by the system represent requests, notifications and replies made by the system's environment, or internal triggers that are generated by the system.

With describing the system's behavior over time, it is necessary to capture the protocol that the system has with its environment and to capture the system's responsibilities in each kind of interaction. By encapsulating in an operation each responsibility that the system has in reacting to each stimuli, it is possible to describe the system's behavior by two separate views; one that describes the effect of each operation, and one that describes the temporal ordering of those operations, i.e., the input protocol that the system has with its environment, where input stimuli trigger operations on the system.

A reactive system may behave differently to the same stimuli at different points in time. Its reaction depends on its current situation (or state). Thus, to be able to predict the effect that an operation has on the system and its environment, it is necessary to have an explicit notion of system state. The state space of the system describes all possible states that the system can be in over its lifetime. Thus, a description of the state space of the system offers another view of the system that is orthogonal but complementary to the operation and input protocol views (mentioned above).

An ANZAC specification describes the behavior of a (reactive) system along these three lines of projection. The resulting views of the projections are described by the Concept Model, the Operation Model and the Protocol Model, which capture the (abstract) state space of the system, the operations of the system, and the input protocol of the system, respectively. In the following subsections, I provide an overview of each model of an ANZAC specification.

4.3.2 Concept Model

The Concept Model defines an abstract state space of the system. The system's state space is the set of all possible populations of entities and relations between those entities that are needed by the system over its lifetime. These entities and relations represent concepts of the problem domain that can be classed as shared phenomena[1]. A system state is defined by a snapshot of the system's state population at a given moment, i.e., the population of all entities and relations contained by the system.

The Concept Model describes the system's state space as the cross-product of all instances of entity types and relation types. An entity type or relation type is only included in the state space of the system if one or more of its instances are needed to describe the

1. The term "shared phenomena" was coined by Jackson [Jack95]. A shared phenomenon is a concept that is visible to both the system and the environment [GGJ+00].

responsibilities that the system has in fulfilling requests from its environment over its lifetime.

The cross-product of all instances of entity types and relation types usually allows for more system states than are possible. As a consequence, invariants are defined to exclude invalid system states from the state space. Unfortunately, it is often impractical to exclude all invalid states from the Concept Model's description of the system's state space. Consequently, often only the obvious invalid states or the clearly unsafe states are removed by invariants, leaving the Operation Model to describe how each operation can correctly change the state of the system.

It is important to emphasize that the Concept Model exists for the sole purpose of providing a means to describe the effect of operations on the system. This is why it is part of the behavioral specification of the system and it is defined as describing an *abstract* state space of the system. Moreover, the Concept Model is not a description of the static structure of the system, since the software does not exist yet, i.e., the description is not captured by abstracting away from the implementation representation details of the software.

Concepts

The Concept Model defines the entities and relations that can conceptually populate the system over its lifetime. These entities and relationships represent concepts of the problem domain. According to Odell and Ramackers [OR97], a concept is “an idea or notion that we apply to classify those things around us”. The kind of concepts that are represented by a Concept Model depends on the problem domain, and on the services that are offered by the system. Thus, they are pertinent abstractions of the problem domain.

In this context, a concept may be something in the system's environment that is monitored or controlled by the system. Equally, it may be a non-physical phenomenon or theory of the problem domain. For example, in an embedded system, pertinent concepts may include things in the environment that the system monitors or controls. However, in a business system, pertinent concepts may include business-related phenomenon, e.g. account, order, goods, etc., facts on people and organizations that interact with the business and facts that relate to the history of their dealings with the business.

Definition

The definition of the Concept Model is the following:

> *Concept Model – defines an abstract state space of the system in terms of entities and relations, which represent concepts of the problem domain, that are needed in the description of system responsibilities.*

Language

In terms of the language used, a Concept Model is described using the language of a restricted UML class diagram with some small specializations for the purpose of describing the state space of the system. This language offers a number of useful qualities for this purpose:

- It offers a rich set of constructs for describing and structuring entities and relations.
- It supports the precise expression of system invariants for constraining the (abstract) state space of the system.
- It offers a visual form that arguably has cognitive advantages for comprehension.

The Concept Model is explained in detail in Chapter 5.

4.3.3 Operation Model

The Operation Model specifies the responsibilities that the system has in performing operations. An operation is a discrete action (or unit of work) that is performed by the system in response to a stimulus—a stimulus may either come from the system's environment or have been internally generated. An operation is a service that is offered to the outside world.

Objectives

There are two principal objectives in describing an Operation Model for a system:

1. to define the responsibilities that the system is obligated to fulfill in performing each operation, and
2. to define the assumptions that are made about the context of execution for each operation.

This last objective may seem a surprising objective, but it is important to clearly state under what circumstances an operation can expect to execute in. This is because bounding the calling context of the operation makes the job easier for designers, i.e., it may not be necessary (nor effective) to consider all possible contexts, and it offers insight for callers into what context the operation can be used.

Definitions

A system operation is defined in the following way:

System operation – encapsulates a discrete and coherent unit of work that corresponds to the system's reaction and complete handling of a stimulus.

Note that a system operation may be performed many times and triggered by different kinds of stimuli.

The definition of the Operation Model is the following:

Operation Model – defines the required (functional) effect of each system operation on the state of the system. For each operation, it expresses the assumptions about the circumstances under which the operation is called, and the obligations placed on the system in offering the operation.

Operation Schemas

The Operation Model describes each operation by way of an Operation Schema. From a notational point of view, an Operation Model is nothing more than a collection of Operation Schemas for a particular system. An Operation Schema expresses the assumptions and obligations, described above, in the form of assertions: pre- and postconditions. An Operation Schema's precondition describes the assumptions about the state of the system before execution of the operation, and also the assumptions about the acceptable values of the operation's parameters. An Operation Schema's postcondition describes the result of the execution of the operation in terms of the effect that the operation has on the state of the system and the stimuli that are sent to the system's environment.

A significant benefit that can be gained by using pre- and postconditions is their ability to abstract away from the details of how the operation is realized. This gives designers the freedom to realize the operation in any way they see fit, so long as they keep within the bounds of the constraint implied by a pre-/postcondition pair. Pre- and postconditions are an important tool for combating over-specification, and they offer a concrete means for abstracting above the algorithmic complexities of operations. Also, it is often easier to describe the result of an operation than it is to describe how it might be computed.

The pre- and postcondition pair defines a correctness rule on the design of the corresponding operation: the operation must terminate in a state that satisfies the postcondition given that the precondition holds. If the precondition is not satisfied, then there is no obligation on the operation to fulfill the postcondition (nor terminate). Pre- and postconditions are generally used in the description of sequential operations, i.e., it is assumed that the operation is the only one executing in the system[1], because a postcondition defines a relation between the before- and after-execution states of the system. Nevertheless with a number of enhancements made to pre- and postconditions, such as those ones proposed by this work, it is possible to describe concurrent operations that can share resources using Operation Schemas.

1. To be more precise, the assumption can be reduced to just the subset of the entities and relations that are accessed by the operation (read/write), which would allow other operations to execute at the same time so long as they have disjoint subsets.

Language

In terms of the language used, each Operation Schema in the Operation Model is described using UML's Object Constraint Language with some enhancements in the following areas: an explicit frame assumption, a notion of creation and destruction of objects, mechanisms for structuring pre- and postconditions, assertions for the sending of stimuli to the system's environment, the ability to specify concurrent operations. This language offers a number of useful qualities for describing operations; these include:

- It is a mathematically founded language, whose principles are based on set theory and first-order predicate logic, and it is amenable to automated checking. These facts mean that this language is less likely to lead to descriptions that are ambiguous, inelegant, and noisy.
- It is made for the explicit purpose of defining constraints on UML models, particularly UML class diagrams (the base formalism for the description of the Concept Model). Therefore, this language can be used uniformly across all three models in an ANZAC specification: pre- and postconditions in the Operation Model, invariants in the Concept Model, and guards in the Protocol Model.
- It has a simple navigation style for constructing constraints—everything, more or less, is achieved by working with and manipulating sets, bags and sequences.
- It has an operational style, which makes it more accessible to users that are more familiar with imperative languages (as opposed to declarative ones).

The Operation Model is explained in detail in Chapter 6.

4.3.4 Protocol Model

The Protocol Model specifies which operations are performed by the system in reaction to stimuli received over time. It combines with the Operation Model to describe the way that the system reacts to all possible stimuli that may be received by the system during its lifetime. In this way, the Protocol Model can be seen as the correctness contract on the system's input protocol. Since a reactive system can react to a certain stimulus in different ways over time, a Protocol Model must take into account that the system is necessarily state-sensitive.

Objectives

There are three principal objectives in describing a Protocol Model for a system:

1. to define the input communication protocol that the system has with the (external) actors in its environment,
2. to define which (system) operations are triggered by stimuli from external actors or internally triggered, and
3. to describe the activities that the system operates.

Definition

The definition of the Protocol Model is the following:

> *Protocol Model – specifies which operations are performed by the system in reaction to stimuli received over time. The choice of operation is determined by the kind of stimulus received, and the current state of the system. It defines the input protocol of the system as a partial order of operations, where operations are structured into activities and temporally related to one another by sequencing, interleaving or concurrent ordering constraints.*

Activities

The Protocol Model is structured in terms of activities, where each activity consists of a sequence of operations that correspond to a coherent work task of the system. In the Protocol Model, each activity is temporally related to other activities using a similar means to the one used to temporally relate operations, i.e., sequence, interleaving and concurrent ordering constraints. Furthermore, activities can be composed to form larger-grained activities, and therefore hierarchies of activities can be defined.

Language

A Protocol Model is described using the language of a restricted UML protocol state machine with some small enhancements for addressing some issues in describing concurrent system behavior and timing constraints. This language offers a number of useful qualities for describing the input protocol of the system; these include:

- The notion of state and transitions between states is explicit in the language, which naturally matches the reactive model of system behavior.
- Ordering constraints on operations and activities can be expressed in terms of sequence, sequential interleaving, or concurrent execution.
- Conditional reactions can be expressed using guards, which covers those cases when the operation chosen or the effect of the operation depends on the value of information passed with the stimuli rather than the state of the system.
- Time-triggered and condition-triggered operations can be expressed directly in the language (without the need to fabricate stimuli that "come" from the system's environment).
- Time-based guards and invariants that define timing constraints can be formulated.
- Sequences of operations can be structured into activities and sequences of activities can be structured into coarser-grained activities, which allows for a hierarchical description of system behavior.

The Protocol Model is explained in detail in Chapter 7.

4.3.5 Communications between System and Actors

Following UML, all communications between the system and the actors in its environment are achieved by message passing (in either direction). According to UML [pp. B-11, Omg01], a message is:

"A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation."

A message represents a request on the target to perform some service or a notification informing the target of some situation or information. Messages may have synchronous or asynchronous semantics. A synchronous message means that the source of the message (i.e., the sender) will wait for the target to process the message, with the target possibly returning a result to the source (depending on the type of request). An asynchronous message means that as soon as the source has sent the message it will continue to perform other tasks. In other words, the source of the message will not wait for the target to finish processing the message.

In an ANZAC specification, a message is an abstraction of the actual means that is conveyed from source to target, where a message may represent a number of (concrete) messages in the actual communication protocol between the system and actors. This abstraction level is the result of a number of omissions concerning the interaction between the system and actors. These omissions are the following:

Abstraction 1: In an ANZAC specification, all interactions required to obtain the necessary and correct information from the user to perform an application-logic operation are abstracted away, i.e., all interactions related to the presentation-logic of the system are omitted. For instance, if a user wishes to place an order with the system, s/he first identifies the products and quantities, etc., which may even require several attempts in the case that a user inputs a wrong identification number for a product, for example. Once the details of the order have been established, the user can then confirm the order with the system (which triggers the application-logic action). This whole series of interactions would be generalized as a single message sent from the user to the system in an ANZAC specification, e.g., the "place order" message.

Abstraction 2: In an ANZAC specification, all interactions needed to ensure secure communication, e.g., encryption, or needed to ensure that an actor is a trusted source, e.g., certificate exchanges, are abstracted away.

Abstraction 3: In an ANZAC specification, all interactions that are needed to ensure reliable communication are abstracted away, e.g., resends, handshakes, etc., where it is assumed communication is reliable.

As well as abstract messages, in an ANZAC specification, the parameters of messages are equally abstractions of the actual information that is conveyed from source to target. A useful rule for finding the right level of abstraction for parameters is the following: all parameters of messages that stand for entities in the system, as described by the Concept Model, should be replaced as parameters by those entities. For example, rather than passing a customer identification string as a parameter in a message, the corresponding customer object would be "passed" instead. This rule promotes parameter types of messages that are less likely to change, because the entities defined by the Concept Model are usually more stable than the data structures that are used to convey the information at the realization level. Note that this rule is applicable to both input and output messages (relative to the system).

4.3.6 System Reaction

There are two ways that the system can be triggered into action, i.e., execute an operation; either by the reception of an incoming message or the generation of an internal trigger (time-out or condition-change), which were referred to as stimuli in the previous sections.

On the reception of a message (from an external actor) or an internal trigger, the system's communication infrastructure generates an event occurrence and places it in the system's input event queue, ready for processing [Omg01]. In an ANZAC specification, a single event occurrence is generated for each message instance received, and the event occurrence that is generated has the same signature (name and parameters) as the message instance received.

At an appropriate time, the system consumes the input event occurrence and according to the rules defined by the Protocol Model, an operation may be triggered. The execution of a system operation will (usually) cause the system to change its state and/or send one or more messages to actors. Similarly to incoming messages, messages that are output by the system correspond to requests that the system makes on actors, or to notifications that inform the actors of some situation or information.

4.4 ANZAC Mapping Technique

In selecting and establishing a formalism to define the Behavioral Design Contract, it is not only necessary to ensure that the formalism is "designer-friendly" but that it is compatible with the formalism that is used in the description of the Behavioral Stakeholders Contract. This need for compatibility is further brought about due to the fact that the Behavioral Design Contract is refined from the Behavioral Stakeholders Contract. With respect to the ANZAC approach, this means that it should be possible to smoothly refine ANZAC use case descriptions to an ANZAC specification.

An ANZAC use case description provides a plan of the possible interactions between the system and the actors in its environment in the form of a set of scenarios. These interactions can be refined into an informal statement of the system's interface to its environment and the responsibilities that the system has in reacting to requests and notifications made by actors. This statement provides the basis for describing the Protocol and Operation Models of an ANZAC specification, where the existential description of scenarios is transformed into a universal one (Protocol Model) and the system responsibilities are gathered to form the descriptions of operations (Operation Model).

The ANZAC approach provides explicit support for this refinement activity. Chapter 8 describes the ANZAC mapping technique, which defines a step-by-step process for refining ANZAC use case descriptions into an ANZAC specification and establishing a traceability relationship between them.

Chapter 5:
Concept Model

This chapter presents the Concept Model, which is one of the three models that form an ANZAC specification. It describes the purpose of the Concept Model, provides some guidelines for constructing one, explains the notation used to represent one, discusses ways to define additional constraints to capture system invariants, and provides an example of a Concept Model for a game system. Finally, it discusses the relationship between entities in a Concept Model and design classes, and thus it defines the difference between classes that are part of the Concept Model and those that could be found in a design.

5.1 Introduction

The way that a reactive system behaves to a particular request depends on the state that it is in when it receives the request. This means that in the description of reactive systems, it is necessary to be able to differentiate one state of the system from another one. For the purpose of describing the state space of the system, an ANZAC specification employs the Concept Model.

The Concept Model defines the (abstract) state space of the system as the set of all possible populations of entities and relations between those entities that are needed by the system over its lifetime for the purpose of fulfilling its responsibilities. These entities and relations represent concepts of the problem domain. The state of the system at any one time is defined by a snapshot of the system's state population at that given moment. A system state is consistent if it is reachable by a valid series of system operations from the initial system state, and if it is a stable state, i.e., it is a snapshot of the system's population of entities and relations with no operations executing.

The kind of concepts that are represented by a Concept Model depends on the problem domain, and on the services that are offered by the system. Thus, they are pertinent abstractions of the problem domain. In this context, a concept may be something in the system's environment that is monitored or controlled by the system. Equally, it may be a non-physical phenomenon or theory of the problem domain.

This chapter is structured in the following way. In section 5.2, I present the notation used to represent a Concept Model. In section 5.3, I briefly describe how one might go about describing a Concept Model. In section 5.4, I present an example of a Concept Model for a game system called Strategic Conquest. Finally in section 5.5, I discuss the relationship between classes in a Concept Model and design classes.

5.2 The Notation of Concept Models

In this section, I present the notation that is used to represent Concept Models. In particular, I describe the graphical model elements that are used and their meaning in the context of a Concept Model.

A restricted form of UML class diagram is used to describe a Concept Model. UML offers class diagrams to graphically depict structural aspects of a system[1]. For the purposes of describing a Concept Model, class diagrams offer a means to describe the entity types and relation types, and to define constraints on them, which allows one to remove invalid system state's from the state space.

5.2.1 Describing the Concept Model with UML

For the purpose of representing a Concept Model in a UML class diagram, the concepts that are deemed pertinent for describing the state space of the system are divided into and described by three groups: classes, relationships between those classes, and constraints on those classes and relationships.

Classes

Concepts of the problem domain can be categorized according to common properties and represented as UML classes. Classes define entity types, using the terminology mentioned earlier.

It is important to note that the notion of class used in a Concept Model is a restricted form of UML class. This is to say that classes in the Concept Model can only contain value

1. Note that according to UML, a class diagram need not contain only classes [pp. 3-34, Omg01].

attributes and their objects do not possess any behavioral capability, because they represent static concepts of the problem domain and not software components.

Class properties are denoted as attributes; an attribute may have class or object scope, and it may vary or stay constant after the point of initialization—in UML, this is termed changeable or frozen, respectively. Other relationships in a Concept Model can be expressed using associations and generalizations. Data types can also be represented in a Concept Model.

Associations

An association is used to denote that there is a relationship between two or more classes, and they define relation types, using the terminology mentioned earlier. An association signifies a fact of the problem domain or of the system.

There are three specific kinds of association: normal, aggregation and composition. An association is defined between two or more classes; it is a type, and it allows any object of a participating class, playing one role in the association, to be connected with another object, who is playing another role in the association (and is an object of a participating class). This connection between instances, called a link in UML, is often dynamic, i.e., it may exist at one moment in time and not at another moment. In a Concept Model, a association is interpreted in much the same way as a mathematical relation.

An aggregate association is used to denote an acyclic, containment relationship between two classes, where one class can have instances, the composites, that contain instances of the other class, the components; however, a component can exist without the composite and it can be shared by different composites. A composition association is an aggregate association where the component can not exist outside of a composite and the component is only ever part of one composite at any one time. Consequently, a component always belongs to exactly one composite. Also, all of the three kinds of associations can be qualified. A qualified association indicates that the links found from navigating from a given end of the association relate directly to a separate key or set of keys. If an association has value attributes that belong to the association and not to either of the classes that participate in the association, then an association class is used to denote the attribute(s).

Generalization

A generalization models a static relationship between two classes, which denotes that one class, the child, can be classified as a variety of the other class, the parent. Practically speaking in the context of the Concept Model, the meaning is that the child has at least all the properties as the parent (attributes and associations).

Rep classes

Apart from classes that represent the concepts of the problem domain, a Concept Model may contain rep classes. A rep[1] class is a special class whose instances represent the identity of an actor instance. Instances of rep classes are used to indicate the address of an actor instance for communication purposes. A rep class is used in a Concept Model only if it is necessary for the system to identify actor instances as individuals. One constraint on a rep class is that its name must match the name of the actor whose identity it denotes. A rep class signifies that a mechanism, beyond a callback to the sender, will be needed to send a message to certain actors. E.g., a directory service, a name server, etc. could be used to realize such a mechanism in the system design.

5.2.2 Graphical Notation for Concept Models

Graphically in a Concept Model, a class is depicted by a box with two compartments, one that shows the name and the other one that lists its attributes. Figure 5.1 shows a class, far left, called ClassA that has six attributes: attrA1, attrA2, attrA3, attrA4, attrA5, and attrA6, of type String, EnumType1, DataType1, Real, Integer, and Boolean, respectively. All four predefined OCL datatypes are used by ClassA, namely Real, Integer, String and Boolean.

All attributes of ClassA, except the third one, indicate they are object-scope attributes (the default interpretation), i.e., each object has a separate value for each attribute. The underline on the third attribute indicates that it is a class-scope attribute, i.e., there is a single value for all objects of the class. The fourth attribute has the {frozen} property, which means that its value can not be changed once it has been initialized; note that the default is changeable. The fifth attribute has a default value (i.e., 0). The attribute is given this value when an object of the class is created but no initial value is explicitly given to the attribute (see Chapter 6 for a further discussion). The sixth attribute is a derived attribute (denoted by the '/'), which means that this attribute can be computed from other elements in the model. Even though it adds no semantic information, it can be particularly useful for denoting shorthands or for enhancing readability.

Second from the left, figure 5.1 shows a class called *ClassB*, whose name is written in italics. The italic font indicates that it is an abstract class, which means that there are no direct objects of that type. It defines a placeholder type for classes that specialize it. Its single attribute attrB1 will be inherited by all classes that specialize from *ClassB*.

Data types, which include enumerated types, are also depicted in a similar way to classes. They have an additional stereotype <<datatype>> and <<enumeration>>, respectively. Figure 5.1 shows an enumerated type, second from right, called EnumType1 that has two enumeration literals, enumLiteral1 and enumLiteral2. It also shows a data type, far right, called DataType1, which has attributes: val1 and val2, of type Boolean and EnumType1, respectively.

1. "rep" is short for "representative"

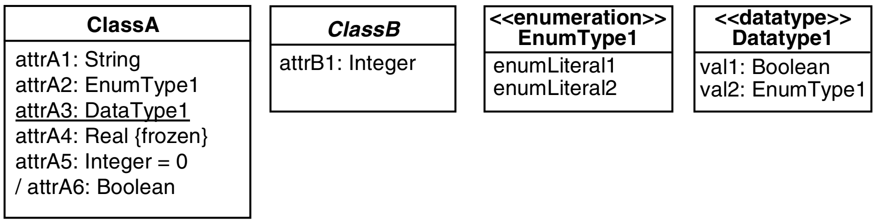

Figure 5.1: Graphic Notation for Class, Enumeration, and Data Type

(Binary[1]) Associations

An association is depicted by an unbroken line drawn between the boxes of all the classes involved in the association, where an aggregation and composition association differs from a (normal) association by the addition of a diamond at the end of the composite. An association optionally has a name, usually located adjacent to the line in the center, and it has zero or more role names, one for each end of the association, placed at the corresponding end. Aggregation and composition associations are differentiated by the color of the diamond: aggregation is white, composition is black. Generally, aggregation and composition associations are usually left unnamed, because the name is predictably always a variation of 'Contains' or 'Has'. An alternative to a line between boxes for a composition association is visual containment, i.e., a class box that lies inside a compartment of the box of another class, where the outer class denotes the composite and the inner class denotes the component of the composition association. Also, an association class joins a class to an association. This is shown graphically by drawing a dotted line between the box of the association class and the line of the association.

Figure 5.2 displays the different kinds of UML associations that are used in a Concept Model. (a) shows an association, named Association1, between Class1 and Class2 with the respective association roles named end1 and end2, each having a multiplicity of zero or more. (b) shows an aggregation association that is similar in form to (a); the diamond indicates that Class3 is the composite and Class4 is the component. (c) shows a composition association that is similar in form to (b), except the multiplicity of end5 is one. In fact, all composition associations have a multiplicity of one on the side of the composite, because the component is never shared or in existence without a composite. (d) shows an association with an association class attached that is similar in meaning to (a), except the association has an attribute attr1 of type DataType1. (e) shows a composition association that has the same meaning as (c), the only difference between the two notations being that (e) does not show role names. Note that it impossible to show explicit role names using visual containment. Also, note that the multiplicity in the top right-hand corner of Class6 is a shorthand for the multiplicity at the component end of the association.

1. Higher-order associations (also known as n-ary associations) are also possible. They are not discussed here, but interested readers are referred to [GLM01] for an in depth discussion of these associations.

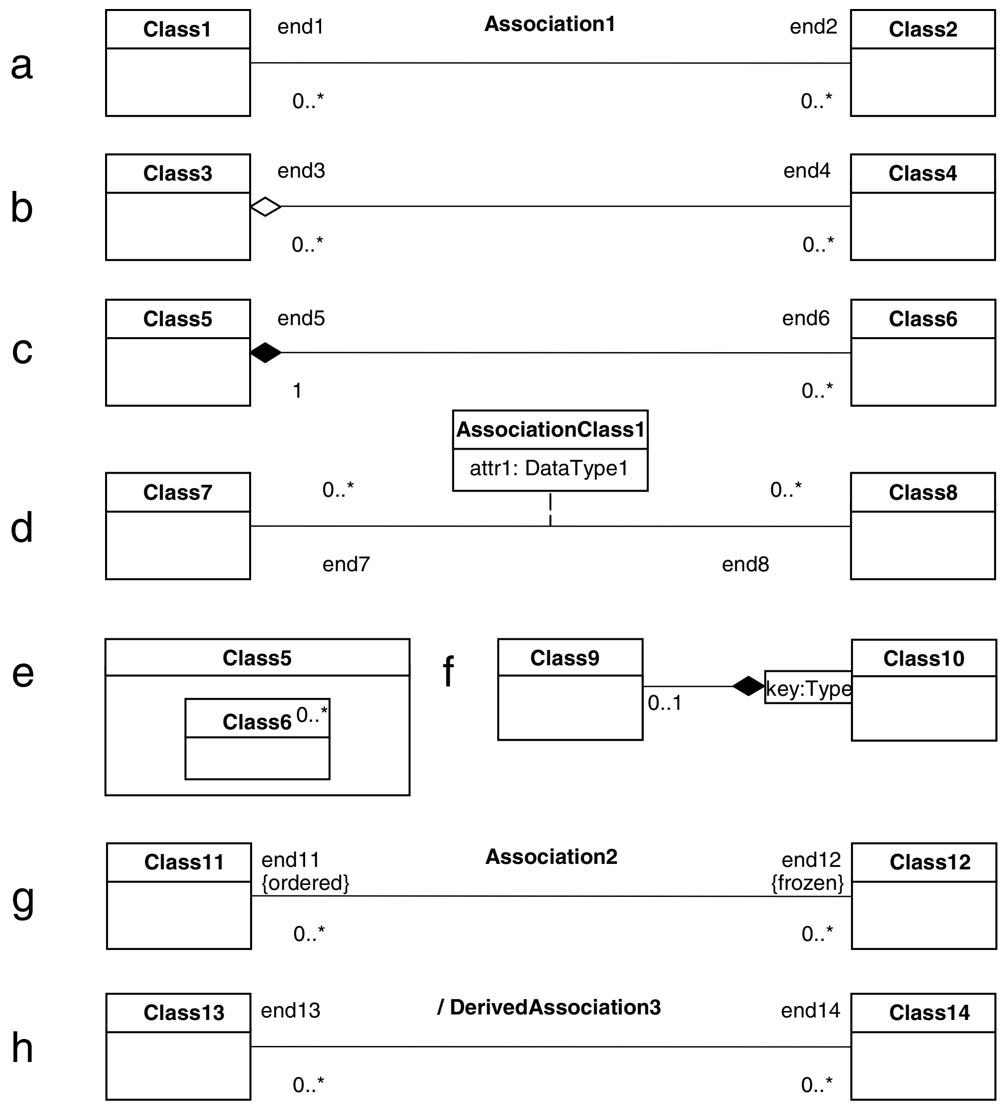

Figure 5.2: Diagrammatic Notation for Associations in the Concept Model

(f) shows a qualified composition association between Class9 and Class10. The association acts as a map, i.e., given an object of Class10 and a key of type Type, one can find zero or one associated objects of Class9. (g) shows an association between Class11 and Class12 with the respective association roles named end11 and end12, each having a multiplicity of zero or more. On association end end11, there is the constraint {ordered}; it means that all links when navigating from an object of class Class12 are in a particular order (i.e., would result in a sequence for an OCL navigation from an object of Class12). Equally, on association end end11, there is the constraint {frozen}; it means that all links when navigating from an object of class Class11 can not be changed after initialization[1] (similar to the constraint on

1. Note, however it is possible to add new links between objects of Class11 and Class12, as long as they do not include objects of Class11 that are already participating in the association.

attributes). (h) shows a derived association between Class13 and Class14. Similarly to a derived attribute, it can be computed from other elements in the model. Even though it adds no semantic information, it can be particularly useful for denoting shorthands or for enhancing readability.

Generalizations

A generalization relationship is depicted by an unbroken line drawn between the two class boxes. A white triangle is placed on the parent end of the line, and the arrow of the triangle points towards the parent and its tip touches the parent class box. Figure 5.3 shows a generalization relationship between Class21 and Class22.

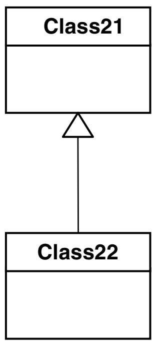

Figure 5.3: Diagrammatic Notation for Generalization in the Concept Model

System

In a Concept Model, all classes belong to the system class and can therefore be shown as being graphically contained. The system class is depicted as a box with two or three compartments: the first one shows the system's name and the <<system>> stereotype, the second one is an attribute compartment, which lists all attributes of the system and is optional if their is none, and the last one shows the components of the system. There is exactly one system class per Concept Model, and the system class is the composite of all other classes in the model, i.e., all classes in a Concept Model model are components of the system, either directly or indirectly (i.e. by transitivity). Figure 5.4 shows a system class called System1. It contains three classes: ClassX, ClassY, and ClassZ, and two associations: AssociationA and AssociationB. Note that the class components have a multiplicity, i.e., their class boxes have a number range in the top right-hand corner; it denotes that the system can contain as many objects as indicated by the number range.

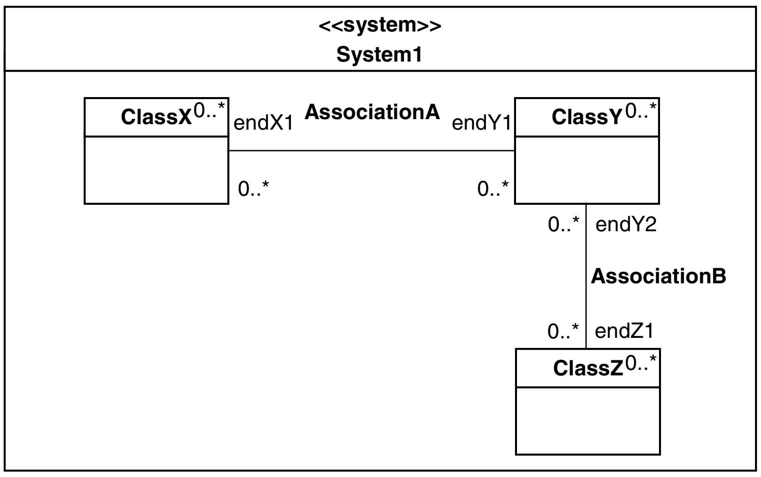

Figure 5.4: Diagrammatic Notation for the System Class in the Concept Model

Rep classes

Zero or more components of the system class may contain a <<rep>> stereotype in the name compartment of their class box; these classes are referred to as rep classes. A rep class can only contain a single compartment, which is used for the name: no attributes are allowed. As a rule, a rep class cannot be associated to another rep class. Usually, a rep class participates in a single (normal) association within the system. If a rep class participates in an association with a class of the system, then the rep class represents the identity of the actor that is corresponds to the class. For example, figure 5.5 shows an association between Customer and Client (a rep class). This association allows one to navigate to the identity of the client (actor instance) that corresponds to a given customer. The customer is the representation of the concept inside the system and the client is the address of the client actor (used for the purpose of communication).

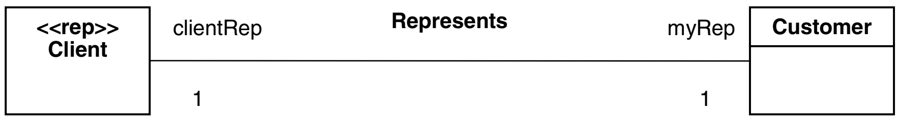

Figure 5.5: An Association between Customer and Client (a rep class)

5.2.3 Constraints

The state of the system is a snapshot of all the objects in the system and the association links between them at a given instant in time. The Concept Model defines the classes of those objects and the associations of those links. Clearly, it would be unusual for every element of the power set of objects and association links of the system to be a consistent system state. Constraints can, therefore, be placed on the classes and associations of the system to restrict the number of allowable states for the system. These constraints are called invariants and they must hold true for all consistent system states. Invariants can be added to a UML class diagram by placing multiplicity constraints on the ends of associations and by attaching constraints written in OCL to elements of the model.

5.2.3.1 Multiplicities

The multiplicity on one end of a (binary) association is a non-negative integer or an (increasing) range of non-negative integers that specifies the number of objects at the target end of the association that may be associated with a single object at the opposite end of the given association. By placing multiplicities on association ends, one can restrict the number of association links between objects, and thus reduce the number of permitted system states.

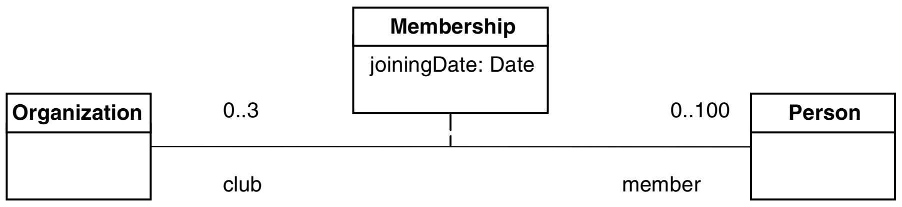

Figure 5.6: Association between the classes Organization and Person

Figure 5.6 shows an example of an association between the classes Organization and Person. The name of the association is Membership and it has a role name for each end of the association: club and member. The association defines a (association) class that holds the date at which a person became a member of the club. It is a rule of associations that there are no duplicate pairs[1], therefore in this example there can be only one date for any (organization object, person object) pair. The multiplicity on the club side of the association indicates that a person, over its lifetime, can have zero, one, two, or three clubs. The multiplicity on the member side of the association indicates that an organization, over its lifetime, can have zero, one, ... up to one hundred members.

5.2.3.1 OCL Invariants

Multiplicities are limited in their capacity to define invariants. For the purpose of defining additional constraints on UML class diagrams, UML provides the Object Constraint Language (OCL) for expressing more complicated constraints.

A small tutorial on OCL is given in Appendix A, nevertheless I provide here a description of some of the important aspects of OCL. OCL invariants are always written with respect to a context, usually a class. Thus, constraints are defined in terms of navigation from a representative object of the context class. This representative object is referred to as self, and the expression stated in the invariant holds true for all objects of the class. The

1. If one requires duplicates (for history purposes, e.g.) then one could refactor the association class into a separate class and define separate associations between it and the two other classes.

dot notation of OCL indicates the traversal of an association, in which case the result is a collection of objects, or the traversal to a property, in which case the result is the value of the property. When navigating on association links, the dot notation is used together with the role name, e.g., p.club (assuming p is of type Person). If there is no explicit role name, then the name of the target class is used as an implicit role name.

Some examples of invariants written in OCL are given in section 5.4.1.

5.3 Describing Concept Models

In this section, I briefly discuss some activities involved in elaborating a Concept Model. Also, I discuss the mutual dependency between the Concept Model and the Operation Model (described in Chapter 6) in the context of defining an ANZAC specification.

The Concept Model represents all concepts of the problem domain that are pertinent to the description of the system's responsibilities and can be classed as shared phenomena. Generally, an entity or relation is only allowed to be part of the Concept Model if it appears in an Operation Schema of the system. (Operation Schemas are discussed in Chapter 6).

It is important to point out that the Concept Model is something that is constructed, not some universal reality that just pops out of the problem domain. Furthermore, a Concept Model is limited to the expressiveness of UML class diagrams and to the specifier's perception of the problem domain.

Arguably, the best way to start a Concept Model is from a description of the domain. From this description, it should be possible to make a "first cut" of the Concept Model. Usually, concept classes are brainstormed first, with properties and relationships being added thereafter, and then constraints and system invariants being added at the end. It is, however, pointless to attempt to come up with a "complete" Concept Model before defining the Operation Schemas, because the scope of the system only becomes clear with the description of the system's responsibilities, i.e., it is necessary to know what state changes are required before one can build a description of the state space. In describing an ANZAC specification, usually the Concept Model is developed along with the development of the Operation Model, where entities and relations, which represent concepts, are added and removed according to the needs of Operation Schemas. In general, this suggests that the Concept Model is not finished until all the responsibilities of the system have been expressed by Operation Schemas.

5.4 An Example of a Concept Model

To demonstrate the various concepts and notations introduced in this chapter, in this section I present a Concept Model for a game system called Strategic Conquest. Strategic Conquest is a two player game of strategy where the ultimate goal is world domination. The game is fought on land and sea and in the air with different types of military units (e.g. tank, fighter planes, etc.) produced by cities when they are under the control of a player. Figure 5.7 shows the Concept Model for the Strategic Conquest Application.

The diagram is divided into two parts: the system class, and the generalization hierarchies and datatypes. Note that classes with the same names refer to the same class. The two generalization hierarchies are drawn outside of the system box to avoid clutter inside the system box.

Describing the system at the top level, it contains zero or one games, zero to two players, and zero to two user rep classes. Within a game there is one map, which contains two or more sectors (links that are frozen, i.e., unchangeable); a game also contains zero or more military units, and an attribute: numDays, signifying the number of days that have passed in the game. Within a sector there is zero or one cities contained (composition links that are frozen).

According to the generalization hierarchy, sectors (abstract) are generalizations of land, sea and port sectors. Military units (abstract) are generalizations of land, sea and air units. Land units (abstract) are generalizations of tank and artillery units. Sea units (abstract) are generalizations of transporter and battleship units. Air units (abstract) are generalizations of fighter planes.

There are a number of associations between the classes in the system. A player is the current player for zero or one game (HasTurn), it can own zero or more cities (Conquered) and zero or more military units (Owns), and it is associated to a particular user rep class. A military unit is always present on a sector (IsPresentAt), it may have a destination sector that is used to store a multi-turn move of the unit (DestinedFor), or it may be under construction by a city (Production, which has an attribute duration). A transporter can have zero to six land units on board. A city can have zero or more military units (/Upholds). This last association is derived from the association IsPresentAt and the composition relationship between Sector and City. The meaning of the derived association /Upholds is given by the following two invariants.

context: MilitaryUnit
inv: self.secures = self.currentSquare.city;
context: City
inv: self.holds = self.sector.contains;

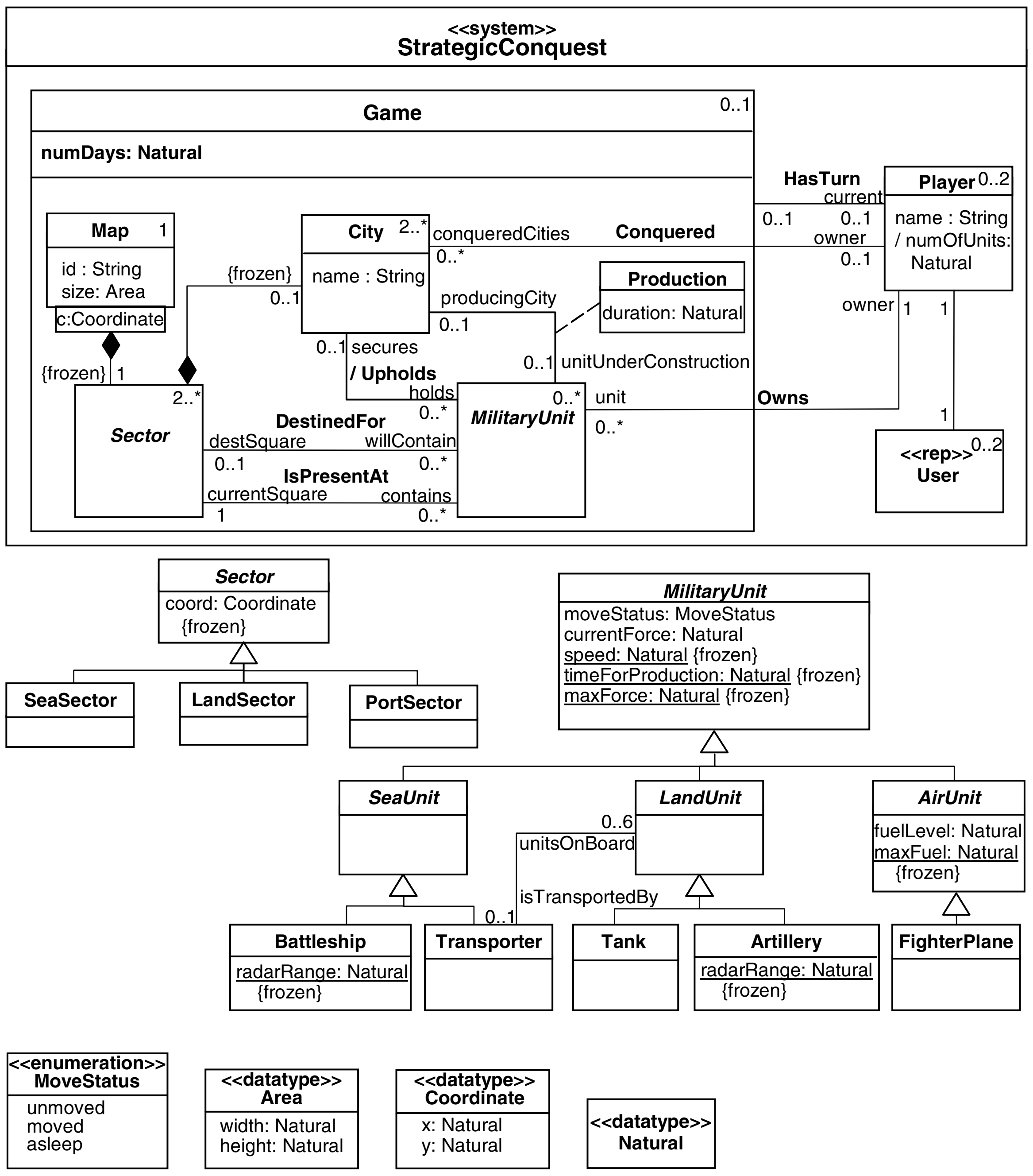

Figure 5.7: Concept Model for Strategic Conquest Application

There are a number of attributes contained in the different classes. Military units have two attributes: moveStatus indicates whether the unit has moved for this round or whether it is asleep, and currentForce indicates the amount of force that the unit currently has. Military units also contain three class-scope immutable attributes: speed indicates the maximum number of sectors that the unit can traverse in one day, timeForProduction indicates the number of days that it takes a city to produce the unit, and maximumForce indicates the maximum amount of force that the unit may possess. Sectors have a immutable attribute coord. The map has two attributes: id and size. Air units have an attribute fuelLevel, and a class-scope immutable attribute maximumFuel. Battleship and Artillery units have a class-scope immutable attribute radarRange. Cities have an attribute name. Players have an attribute name and a derived attribute numOfUnits, which represents the number of units that the player owns. The meaning of this derived attribute is given by the following invariant.

```
  context: Player
  inv: self.numOfUnits = self.unit->size ();
```

5.4.1 Invariants for the Strategic Conquest System

Below, I provide a number of different invariants that could be placed on the Concept Model for the Strategic Conquest system. I first provide a textual description of the invariant, which is followed by the OCL description.

1) All land units are on a land or port sector except those that are on a transporter, which are on sea or port sectors.

```
context: LandUnit
inv:
  if self.isTransportedBy->notEmpty () then -- if land unit is being transported (on a Transporter sea unit)
    not self.currentSquare.oclIsTypeOf (LandSector) -- then land unit is not on a land sector
  else
    not self.currentSquare.oclIsTypeOf (SeaSector) -- otherwise land unit is not on a sea sector
  endif;
```

2) All sea units are on sea or port sectors

```
context: SeaUnit
inv: not self.currentSquare.oclIsTypeOf (LandSector) -- literally: sea unit never has a link to a land sector
                                                     -- (wrt. IsPresentAt association)
```

3) All sea sectors contain only a single sea unit

```
context: SeaSector
inv: self.contains->select (su | su.oclIsKindOf (SeaUnit))->size () = 1 -- sea sector has only one sea unit
```

4) All units on a sector are owned by a single player

```
context: Sector
inv: self.contains->forall (u1, u2 | u1.owner = u2.owner) -- all units on a sea sector have the same owner
```

5) All cities are on land or port sectors

```
context: City
inv: not self.sector.oclIsTypeOf (SeaSector) -- city is not on a sea sector
```

6) All air units do not have more fuel than their maximum fuel capacity

```
context: AirUnit
inv: self.fuelLevel <= AirUnit.maxFuel -- an air unit's fuel level value is less than or equal to its class' max. fuel value
```

7) All military units do not have more force than their maximum force capacity and they cannot be in production with a duration greater than their standard production duration.

```
context: MilitaryUnit
inv: self.force <= MilitaryUnit.maxForce and self.production <= MilitaryUnit.timeForProduction
-- a miltary unit’s force value is less than or equal to it class’ max. force value
-- and its elapsed production time value is less than or equal to it class’ production period value
```

Note that the invariant 3 makes use of the property oclIsKindOf. The meaning of oclIsKindOf is the following: x.oclIsKindOf (atype) is true if atype is the type of object x, or one of its supertypes (by transitivity). Thus, invariant 3 states that a sea sector only has one military unit that is either a battleship or a transporter (since a sea unit is abstract).

5.5 Relating the Concept Model to Design Artifacts

In a Concept Model, the classes and associations represent concepts of the problem domain. An important advantage of using the vocabulary of the problem domain is that it is more stable to changes compared to descriptions that use the vocabulary of the software domain.

There is nevertheless a correlation between the entities defined in the Concept Model and the classes that form part of the application-logic layer of a system. To this end, the design of the system must at least encapsulate the state space defined by the Concept Model in some form[1]. With respect to object-oriented design, a class defines a coherent unit of data together with related services that manipulate the data or provide access to it; hence classes of the Concept Model are candidates for design classes, because they encapsulate a coherent unit of data, being one prerequisite for a design class[2]. However, care should be taken that the object identification activity is not solely data-driven [WWW90, W-B94b] (see Chapter 2 for a further discussion on this point). Also, there will often be additional classes that are needed to realize an object-oriented design of a system. The roles that these additional classes play include: controllers, coordinators, structurers, servicers, and interfacers [W-B92].

That said, the information represented by several classes in a Concept Model may be folded into a single design class, and vice versa. The reason for folding several Concept Model classes into one design class and vice versa are often driven by performance, maintenance, scalability, etc. requirements.

One conclusion that one can draw is that the transition from the Concept Model to a design class model is far from seamless [Kai99]. It requires consideration beyond the scope of a Concept Model: how does one distribute the required behavior of the system among objects, while taking into account reuse of already existing software components, resilience to expected changes, performance constraints, etc. These significant decisions require one

1. Catalysis [DW98] takes an even stricter view on this relationship by defining refinement relationships between the classes in the specification (which it calls types) and the classes in the design.
2. Another important property of an (design) object is information hiding. This means that other objects rely on the services offered by the object and not on its data, allowing developers to change at least the form of the data without effecting the service it offers to the “outside world”.

to at least consider the other models of the ANZAC specification, and probably also the description of other extra-functional requirements.

Chapter 6:
Operation Model

This chapter presents the Operation Model, which is one of the three models that form an ANZAC specification. The Operation Model is described by a collection of Operation Schemas, where each Operation Schema describes a system operation by pre- and postconditions. This chapter focuses on explaining and detailing Operation Schemas. It proposes a language and form for Operation Schemas; introduces a number of constructs, rules, assumptions, and interpretations that apply to them; describes how they can be used to express that messages are sent to actors during the execution of the operation; and proposes some enhancements to them for the purpose of specifying the behavior of concurrent operations.

6.1 Introduction

It is important in the development of a reactive system that its responsibilities in serving requests are clearly described. For this purpose, an ANZAC specification employs the Operation Model. The Operation Model provides a description for each operation offered by the system using an Operation Schema. Therefore, from a notational point of view, an Operation Model is nothing more than a collection of Operation Schemas for a particular system. The focus of this chapter is on the Operation Schema.

An Operation Schema describes an operation provided by the system in terms of the assumptions about the circumstances under which the operation is called, and the obligations placed on the system in offering the operation, given that the assumptions are respected by the requester. The assumptions and obligations are stated in terms of pre- and postconditions for the operation, respectively.

This chapter is structured as follows: In section 6.2, I discuss the meaning of pre- and postconditions with respect to Operation Schemas. In section 6.3, I introduce the language that is used to describe a pre- and postcondition pair in an Operation Schema. In section 6.4, I describe the form of an Operation Schema and explain the consistency rules for the various clauses of Operation Schemas. In section 6.5, I present a number of constructs, rules, assumptions, and interpretations that apply to Operation Schemas. In section 6.6, I introduce two mechanisms that are used to structure Operation Schemas. In section 6.7, I describe how Operation Schemas can assert that messages are sent to actors with the execution of the operation, and I define a number of message kinds. In section 6.8, I explain some additional features of Operation Schemas for expressing synchronous calls. In section 6.9, I describe some enhancements to Operation Schemas for the purpose of specifying concurrent operations that share resources. Finally, in section 6.10, I present an Operation Schema for an operation of an auctioning system, which illustrates many of the aspects introduced in the previous sections.

6.2 Pre- and Postconditions in Operation Schemas

In this section, I discuss the meaning of pre- and postconditions within the context of their use in Operation Schemas. In particular, I propose a correctness rule, which defines the contract that the Operation Schema places on the design of the operation.

An Operation Schema declaratively describes the assumptions and obligations that any realization of the corresponding system operation must observe. It expresses these assumptions and obligations by pre- and postconditions that must hold true immediately[1] before and after the execution of the operation, respectively. An Operation Schema's precondition describes the assumptions about the state of the system before execution of the operation, and also the assumptions about the acceptable values of the operation's parameters. An Operation Schema's postcondition describes the result of the execution of the operation in terms of the effect that the operation has on the state of the system and the message instances that are output.

Pre- and postconditions are evaluated in a consistent system state, i.e., no operations are executing. Also, it does not make sense for an Operation Schema to have a precondition and not a postcondition, and vice-versa—they work together as a couple. In an Operation Schema, if a pre- or postcondition is missing then the meaning is that the missing condition is true by default.

1. The meaning of "immediately" is that the pre- and postconditions are evaluated without the system changing state between the time of evaluation and the start and end of the operation, respectively.

The correctness interpretation of an operation with respect to an Operation Schema's pre- and postcondition is similar to the interpretation used by the Larch family of specification languages [GHW85]. Before I define it, I first give a definition of the terms assumption and obligation that are used by it.

Definition 6.1:Assumption – the precondition and all system invariants are satisfied immediately before the operation is executed.

Definition 6.2:Obligation – the operation must terminate in a state that satisfies the postcondition and all system invariants.

Definition 6.3:Correctness Rule – If the assumptions are satisfied, then the obligations are satisfied. If any of the assumptions fail, then the Operation Schema places no obligation on the system to behave in a particular way, and no obligation on termination.

The correctness interpretation for pre- and postconditions, given by definition 6.3, ensures termination under "contractually" normal circumstances; this is an important property of discrete operations for reactive systems.

Explaining the notion of contract in this context, Meyer [Mey97] proposes that the precondition can be seen as a contract on the client, i.e., the requester, and the postcondition as a contract on the server, i.e., the one performing the request. So contractually, if the client cannot fulfill its part of the contract, i.e., the precondition, then the service provider (i.e., the system) is under no obligation to meet its part of the contract, i.e., the postcondition. This means that in the case that the operation is called and the precondition does not hold, the operation can literally do anything, i.e., it is under no obligation to be "well behaved". However, from a "caller-friendly" point of view, it would seem appropriate to propagate a special exception to the caller in such situations[1], if it is possible to do so (i.e., if the check on the precondition terminates). It is important to emphasize "special exception", because calling an operation whose precondition fails is incorrect client behavior, i.e., a bug in the client realization. This circumstance should be differentiated from the raising of an exception that relates to a valid call on an exceptional system state in the context of the operation execution, e.g., customer has insufficient funds. Although, it is important to note that it is only possible to separate pre-condition exceptions from other "normal" exceptions in the context of an operation. For example, "customer has insufficient funds" could equally be a precondition for a different operation.

Pre- and postconditions are declarative in nature, therefore they are side-effect free, i.e., the evaluation of a postcondition does not change the state of the system nor generate any message instances. A postcondition can rather be seen from an observational point of

1. Note that Eiffel's implementation of design-by-contract [Mey92a] (spear-headed by Meyer) raises a special exception each time a pre- or postcondition fails [Mey92b].

view: “a team of observers is asked to say what they saw happen to the system with the execution of the operation, with the results being compared to the postcondition”.

The vocabulary used in the description of pre- and postconditions in Operation Schemas comes from the Concept Model. Consequently, pre- and postconditions of Operation Schemas must conform to the constraints imposed by the Concept Model. Furthermore, according to definition 6.3, system invariants must also be obeyed as part of the contract. In logical terms, this means that the invariants are implicitly conjoined with both the pre- and postconditions of an Operation Schema. Intuitively, Operation Schemas only have to show that they do not contradict invariants, not that they logically conform to them (because of the implicit inclusion). This notion of invariant comes from Hoare’s work on invariants [Hoa72].

Within the constraints of the Concept Model, a postcondition of an Operation Schema can assert that objects are newly part or no longer part of the system’s state population, that attribute values of objects are changed, that association links are added or removed, and that certain message instances are output.

6.3 Language for Writing Pre- and Postconditions in Operation Schemas

In this section, I introduce the language that is used to describe a pre- and postcondition pair in an Operation Schema.

UML’s Object Constraint Language (OCL) was chosen as the language for defining pre- and postconditions in Operation Schemas. Clearly, the choice of defining ANZAC specifications in UML makes OCL a primary candidate for defining the pre- and postconditions of Operation Schemas. What makes OCL a good choice as constraint language for writing Operation Schemas is that it is targeted at bridging the gap between natural language descriptions, which are prone to ambiguities and imprecision, and traditional formal languages, which often have the disadvantage of requiring a strong mathematical background. Furthermore, the audience targeted by the (language) designers of OCL meets our targeted audience, i.e., developers who do not necessarily have a strong mathematical background [CKM+99b, WK98]. Nevertheless, OCL is a mathematically founded language, which has a formal semantics, where its principles are based on set theory and first-order predicate logic.

OCL is reasonably easy to learn and use, particularly if the writer/reader is already familiar with UML’s class diagram notation, even though one should admit that it is sometimes verbose, due to its simple navigation style for constructing constraints—everything, more or less, is achieved by working with and manipulating sets, bags and sequences. The downside that one faces with taking OCL on board is that it is still a young language, and consequently there is a lack of “industrial-strength” tool support (see [KO01] for a full list of tools) and certainly nothing that could compare to some of the powerful analysis tools that have come out of work on formal methods, such as, B [Abr88], PVS [OSR99], HOL [GM93], Isabelle [Pau94], Z/Eves [Saa97], COOPN tools [BBC00], etc.

In [SS02], Strohmeier and I made the observation that the target audience, i.e., developers, are in general more familiar with procedural programming languages compared to declarative ones. We concluded that most developers would be more inspired with a style that was familiar to them. This motivated us to make some changes to OCL, giving it a procedural programming language-like flavor, and reducing the strict declarative style. Furthermore, we called for explicit guidelines for those people who would write and/or read Operation Schemas, which would shed some light on some of the grey areas of pre- and postcondition descriptions, such as, the frame problem, deletion and creation of objects, interference in the face of concurrency, exceptions, etc.

6.3.1 Writing Pre- and Postconditions in OCL

Each pre- and postcondition of an Operation Schema is a boolean expression written in OCL, i.e., they can evaluate to true or false. The Concept Model is used as a plan of navigation for writing OCL expressions, i.e., navigation expressions are constructed by traversing associations from objects in one class to other objects in other classes. In fact, association links between objects act like a network, allowing one to navigate to any state information that is used by an Operation Schema. Most expressions involve navigation, and navigation usually entails one to work with and manipulate sets, bags and sequences.

To illustrate how OCL can be used, figure 6.1 shows a Concept Model for LotteryTicketManager—a system that manages lottery tickets. LotteryTicketManager consists of four (normal) classes: Customer, Branch, Ticket and Draw, two rep classes: Client and Manager, and one association class: Bought. In this system, customers are members of a particular branch (association IsMemberOf), they possibly own many tickets (association Bought), and each customer has a set of favorite numbers (attribute favoriteNums). A ticket is always sold by a particular branch (association Sells), it is part of a particular draw (association Has), it is owned by a particular customer (association Bought), it holds a particular set of chosen numbers (attribute chosenNums), it has a certain cost (attribute cost), and it has a certain winning value (attribute value). Finally, draws have a date when they are drawn (attribute drawDate), a draw number (attribute drawNum) and a set of winning numbers (attribute winningNums). The LotteryTicketManager system is used as an ongoing example throughout this chapter.

In OCL, navigation is achieved with the dot operator. The use of the dot operator indicates either the traversal of an association, in which case the result is a collection of those objects that have a link to the source object (i.e., the one being navigated from), or the access of a property of the respective objects, in which case the result is the value of that property.

When navigating from an object to its attribute, the dot notation together with the name of the attribute is used. For instance, given figure 6.1 and cust: Customer, the expression: cust.name, results in the name of the customer, e.g., 'Richie McCaw'. When navigating on association links, the dot notation is used together with the role name. For instance, given figure 6.1 and burleigh: Branch, the expression: burleigh.members, results in the set of all members (objects of type Customer) of the branch burleigh, i.e., it results in the set of objects of type Person that have a link with the object burleigh.

If there is no explicit role name, then the name of the target class is used as an implicit role name, e.g., given figure 6.1 and draw123: Draw, the expression: draw123.ticket, results in the set of all tickets that are in the draw 123 i.e., it results in the set of objects of type Ticket that have a link with the object draw123; note that there is a maximum of 1000 tickets associated to any particular draw, which bounds the possible number of elements in the set. Composition associations can also be navigated upon. For instance, given figure 6.1 and ltm: LotteryTicketManager, the expression: ltm.customer, results in all customers that are contained within the ltm system.

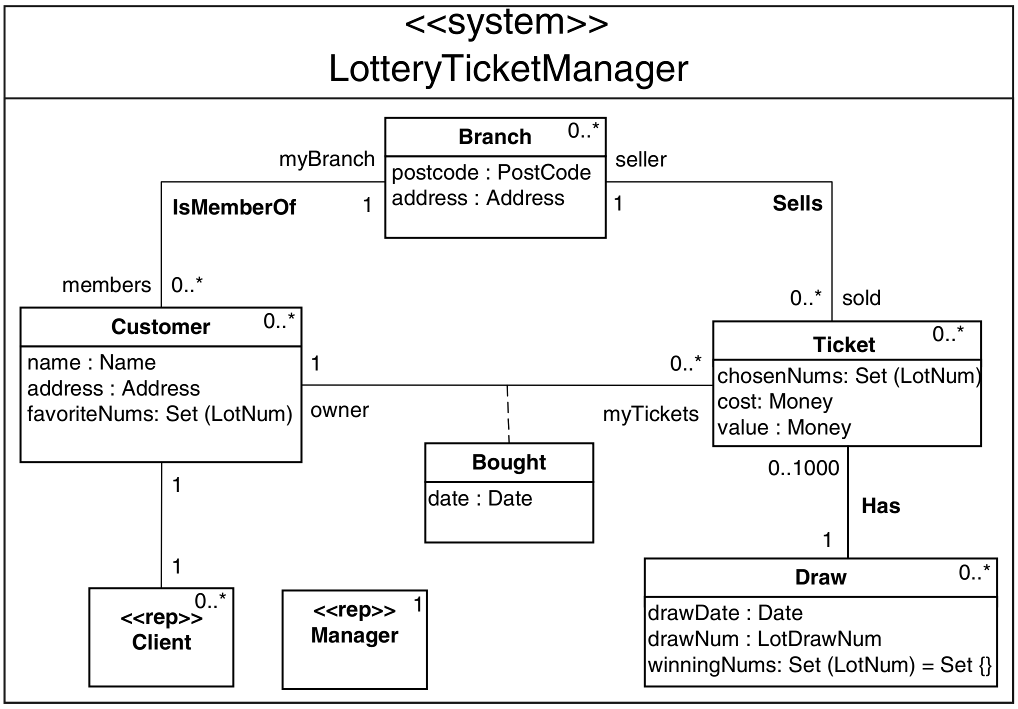

Figure 6.1: Concept Model for LotteryTicketManager

Each OCL expression is written in the context of an instance of a specific type. The predefined identifier self refers to this instance. In OCL, the context is explicitly declared. In the case of an Operation Schema, the context is implicit: it is an instance of the system that owns the operation.

OCL defines a number of predefined properties for objects. One of these is the oclIs-New property, which is used to denote object creation in a postcondition. For example, as part of a postcondition, we could write the following expression:

```
draw123.oclIsNew ()
```

which results in true if the object is newly created during the execution of the corresponding operation. Note that use of the property only makes sense in a postcondition.

A more comprehensive summary of OCL is given in Appendix A.

6.4 The Form of an Operation Schema

In this section, I explain in detail the form of an Operation Schema. In section 6.4.1, I describe the various clauses that make up an Operation Schema. In section 6.4.2, I explain the rules for consistency between and within clauses of Operation Schemas.

6.4.1 Operation Schema Form and Structure

An Operation Schema consists of a number of different clauses. All clauses of the Operation Schema except the first are optional. They are listed below:

Operation:
Description:
Notes:
Use Cases:
Scope:
Messages:
New:
Aliases:
Pre:
Post:

In the following subsections, under the heading of the corresponding clause name, I give a brief explanation of each clause and an informal BNF description. The examples used throughout this subsection are based on the Concept Model of figure 6.1. The full grammar of an Operation Schema is given in Appendix B. Further details on the form and usage of Operation Schemas can be found in [StS01].

6.4.1.1 Operation Clause

The *Operation* clause declares the signature of the operation, which consists of the name of the system followed by the name of the operation, and an optional comma-separated parameter list and return type. The system defines the context of the schema, i.e., OCL's self keyword refers to the corresponding system instance. All parameters are of mode "in", which does not mean that the state of an object that is a parameter cannot be changed. Also, all parameters have defined values, i.e., object references point to objects in the system—no null references. The Operation clause has the following form:

“**Operation**” “:” SystemTypeName “::” OperationName “(” [ParameterList] ”)” [“:” ReturnType] “;”

The following example of an Operation clause declares the informWinners operation of LotteryTicketManager, which has two parameters.

Operation: LotteryTicketManager::informWinners (draw: Draw, winNums: Set (LotNum));

6.4.1.2 Description Clause

The *Description* clause provides a concise natural language description of the purpose and effects of the operation. The Description clause has the following form:

“**Description**” “:” Text “;”

The following example of a Description clause describes the informWinners operation.

Description: This operation informs all winning lottery ticket holders that they won, where a ticket is classed as a winning one if it matches the winning numbers and is part of the given draw;

6.4.1.3 Notes Clause

The *Notes* clause provides additional comments about the respective operation. The comments are written in natural language and are deemed noteworthy. The Notes clause has the following form:

“**Notes**” “:” Text “;”

The following example of a Notes clause is given for the informWinners operation.

Notes: An informWinners operation can only be invoked by the Head Officer of the lottery;

6.4.1.4 Use Cases Clause

The *Use Cases* clause declares all use cases (0 or more) that have a (traceability) relationship with this operation. The Use Cases clause consists of a list of use cases (and use case step numbers in curly brackets), each one terminated by a semicolon:

“**Use Cases**” “:” (UseCaseName [“::” “{” (StepNum “;”)+ “}”] “;”)+

The following example of a Use Cases clause is given for the informWinners operation.

Use Cases: DrawWinningLotteryTicket::{2; 3a;};

This clause would mean that the operation was derived from, or related to, steps 2 and 3a in the Draw Winning Lottery Ticket use case.

6.4.1.5 Scope Clause

The *Scope* clause declares all those classes and associations from the Concept Model that define the name space of the operation. Note that messages are not included in this clause (they are shown in the Messages clause). The Scope clause is a list of classes and associations, each one terminated by a semicolon:

“**Scope**” “:” (ClassOrAssociationName “;”)+

An omitted Scope clause defaults to the entire namespace of the system. Also, for reasons of convenience, rep classes are not included in the Scope clause.

To illustrate the Scope clause, the following example is given for the informWinners operation.

Scope: Customer; Ticket; Draw; Bought; Sells; Has;

The first three items are classes, the fourth one is an association class, and the last two are associations.

6.4.1.6 Messages Clause

The *Messages* clause declares the possible messages that can be output with the execution of the operation. It has the following form:

“**Messages**” “:” (ActorWithMessages “;”)+

This clause declares the type of messages that can be sent by the operation together with their destinations, i.e. the receiving actor classes. ActorWithMessages shows the type of message that can be sent to a given actor class.

ActorWithMessages ::=
  ActorClassName “::” “{” (MessageTypeName [“**Throws**” ExceptionMessages] “;”)+ “}”
ExceptionMessages ::= MessageNameList
MessageNameList ::= MessageName (“,” MessageName)*

ExceptionMessages defines the exceptions that might be thrown by a call denoted by the message preceding the **Throws** keyword. The **Throws** keyword and its usage is described in section 6.8.2.

The following example of a Messages subclause is given for the informWinners operation.

Messages: Client::{WonLottery;};
Manager::{NumOfWinnersNotification; WinnerStatistics;};

The first line states that actors of type Client may be sent messages of type WonLottery. The second line states that actors of type Manager may be sent messages of types NumOfWinners-Notification and WinnerStatistics. Note that it does *not* indicate the number of messages that may be sent to the actors, and equally it does *not* indicate the number of actor instances that may receive a message.

6.4.1.7 New Clause

The *New* clause provides a declaration of all those names in an Operation Schema that refer to new objects or (new) messages. All names declared in this clause are local to the schema. The New clause has the following form:

“**New**” “:” (ObjectOrCollectionDecl | MessageInstanceOrCollectionDecl)+

A name given in an ObjectOrCollectionDecl designates the name of an object or collection of objects that is used in the Post clause to indicate that an object or collection of objects have newly become part of the system state. Association objects can also be declared (i.e., instances of association classes). Each name ideally declares a distinct object or collection of objects, and it must be a unique name in the Operation Schema (i.e., no clashes with other identifiers). ObjectOrCollectionDecl has the following form:

```
ObjectOrCollectionDecl ::= ObjectDecl | ObjectCollectionDecl
ObjectDecl ::= ObjectName (“,” ObjectName)* “:” ClassName “;”
ObjectCollectionDecl ::=
   CollectionName (“,” CollectiontName)* “:” CollectionLiteral “(” ClassName “)” “;”
```

Any object declared in this clause is newly created with the execution of the operation. Therefore, these objects can not be used to match already existing objects in the typical declarative matching style. For example, the following Operation Schema fragment is NOT well-formed because minPayOut is not defined before the operation, i.e., minPayOut does not exist at precondition time and therefore should not be used in the Pre clause.

New: minPayOut: Ticket;
Pre: winningTickets->includes (minPayOut)
Post: winningTickets->forall (t | minPayOut.value <= t.value) ...

The New clause can also declare message instances and collections of message instances. The types of the message instances must be also mentioned in the Messages clause, otherwise this clause is not well-formed. New message instance declarations have the following form:

```
MessageInstanceOrCollectionDecl ::= MessageInstanceDecl | MessageInstanceCollectionDecl
MessageInstanceDecl ::= MessageNameList “:” MessageTypeName “;”
MessageInstanceCollectionDecl ::=
   MessageNameList “:” CollectionLiteral “(” MessageTypeName “)” “;”
```

Illustrating a New clause with object and message declarations, consider the following example of a New clause for the informWinners operation (which respects the Messages clause example, shown earlier):

```
New:      youWon: WonLottery;
          numWinners: NumOfWinnersNotification; stats: Set (WinnerStatistics);
          newTicket: Ticket; -- new object (the others are messages)
```

The first line declares a message instance of type WonLottery called youWon. The second line declares a message instance of type NumOfWinnersNotification called numWinners and declares a set of message instances of type WinnerStatistics called stats. The third line declares a new ticket object of type Ticket.

6.4.1.8 Aliases Clause

The *Aliases* clause provides a declaration of all those names in an Operation Schema that refer to expression-substitutions. Everything declared in this clause is local to the schema. The Aliases clause has the following form:

“**Aliases**” “:” ExpressionSubstitutionDecl+

An alias designates a name for an expression, which is written in OCL and is well-formed, i.e., the expression resolves to a valid type. This name must be unique in the Operation Schema (i.e., no clashes with other identifiers). ExpressionSubstitutionDecl has the following form:

```
ExpressionSubstitutionDecl ::= Name ":" TypeName "Is" Expression ";"
```

Expression must conform to the type of TypeName. To illustrate an expression-substitution, one could declare some expression-substitutions for the informWinners operation, such as the following ones:

```
winningTickets: Set (Ticket) Is draw.ticket->select (t | t.chosenNums = winNums));
winners: Set (Customer) Is self.customer -> select (p |
                              p.myTickets->intersection (winningTickets)->notEmpty ());
thereIsAtLeastAWinner: Boolean Is winners->notEmpty ();
```

winningTickets captures the set of all tickets associated to the draw that match the winning numbers (where draw and winNums are parameters of the operation). winners captures the set of all customers that own one or more winning tickets. thereIsAtLeastAWinner evaluates to true if there is at least a winner, otherwise it is false.

The name of the expression-substitution, e.g., winningTickets, can be used throughout the Operation Schema in all situations where the expression itself could be used correctly. An expression-substitution is similar to a macro, where the expression is substituted for the name in all places of use. However, there are two additional constraints on the substitution. First, the substitution is treated as an atom with respect to the context of substitution, overriding precedence rules, i.e., one can think of the substituted expression as having parentheses placed around it.

Expression-substitutions can also use decorators (i.e., @pre) within Expression. In fact, the expression that is substituted is interpreted in that context, and therefore care should be taken to use the correct decorator so that the interpretation is the intended one.

The Aliases clause allows schema-wide declaration of expressions to be made in a distinct and single place, which is in line with the proposal of Cook et al. [CKM+99a].

6.4.1.9 Pre Clause

The *Pre* clause defines a condition that represents the assumption on the state of the system and/or parameters before the execution of the operation. It is a predicate written in OCL. An omitted Pre clause means the condition defaults to true.

```
"Pre" ":" Condition ";"
Condition ::= BooleanOCLExpression
```

To illustrate the Pre clause, the following example is given for the informWinners operation.

```
Pre:  thereIsAtLeastAWinner and winNums->size () = 5; -- LHS of 'and' is an expression-substitution
```

The Pre clause asserts that the operation is performed under the assumption that there is at least a winner (note that thereIsAtLeastAWinner is an expression-substitution declared in the

Aliases clause example from above) and that there are exactly 5 winning numbers (winNums is a parameter of the operation).

Note that the Pre clause may make use of Parameterized Predicates and Functions (described in section 6.6).

6.4.1.10 Post Clause

The *Post* clause defines a condition that represents the required state of the system after the execution of the operation. It is a predicate written in OCL. An omitted Post clause means the condition defaults to true.

```
"Post" ":" Condition ";"
```

A Post clause defines changes to system resources by way of assertions on the value that they have after the execution of the operation. In a Post clause, the variables that are used in the description of the postcondition have two snapshots: one taken before the operation, and one taken after the operation. The snapshot taken before the operation is denoted by postfixing the decorator @pre to the expression that represents the entity, and the snapshot taken after the operation is denoted simply by the expression that represents the entity, i.e., without decorator. Having these two values for an entity allows one to define the effect of the operation on this entity relative to its value before the operation's execution (@pre).

As a rule, any navigation to a frozen attribute or frozen association end (i.e., the corresponding Concept Model shows the property {frozen} for the item) does not require the addition of the @pre suffix. Furthermore, for reasons of conciseness, it is often more convenient to drop the @pre suffix for all entities that are not changed by the operation (but are not frozen). However, care needs to be taken in maintenance activities that expressions, which rely on an entity to stay unchanged, are not invalidated, due to a change made to the Operation Schema.

To illustrate a Post clause, the following example is given.

```
Post:
  draw.winningNums = winNums &
  draw.drawNum = draw.drawNum@pre + 25;
```

The first line of the Post clause asserts that after the execution of the operation the winningNums attribute of draw is equivalent to winNums (both draw and winNums are both parameters of the operation). The second line asserts that the drawNum attribute of draw is equal to its previous value plus 25.

A Post clause may make use of Parameterized Predicates and Functions (described in section 6.6), but it is not possible to refer to another Operation Schema within the postcondition of a schema. When such situations are deemed necessary, Parameterized Predicates should be used to describe the commonality and then they can be "called on" in the respective schemas.

6.4.2 Rules Governing Consistency Between Clauses

In the activity of checking Operation Schemas, it should be ensured that the syntax and types of each clause of an Operation Schema are correct. Type correctness can only be checked if all type and message information is available. Also, type checking an Operation Schema requires that all of its clauses are type consistent.

Operation Schemas should also be internally consistent, consistent with each other, and all declarations should be used at least once (no floating declarations) and should not re-declare any parameters or predefined identifiers. The consistency rules for each clause, assuming each clause is syntax and type correct, are the following:

- The Operation clause should be internally consistent: no repeated formal parameter names.
- The Description and Notes clause should not contradict the other clauses in the commentary given.
- The Use Cases clause should be internally consistent: no repeated use case names, and only valid step numbers.
- The Scope clause should be consistent with the other clauses: it contains all classes and associations used in the other clauses. And, it should be internally consistent: no repeated classes and association names.
- The Messages clause should be consistent with the other clauses: it contains all names of message types used in the New clause.
- The New clause should be consistent with the other clauses: it contains all names of objects and message instances used in the Operation Schema, and it should declare only objects that have a class shown in the Scope clause and declare message instances that have a type shown in the Messages clause.
- The Aliases clause should be consistent with the other clauses: all the classes and associations mentioned, and those inferred by the expressions (found on the RHS of an expression-substitution declaration) are included in the Scope clause. And, it should be internally consistent: no repeated name declarations.
- The Pre clause should be consistent with the other clauses: it does not refer to any objects that are declared in the New clause, all the classes and associations inferred by the condition are included in the Scope clause, and the condition refers only to entities declared as expression-substitution in the Aliases clause, to parameters of the operation, to the predefined identifiers: self and sender, or to entities navigated to from any of the previous ones.
- The Post clause should be consistent with the other clauses: all the classes and associations inferred by the condition are included in the Scope clause, and the condition refers only to entities declared in the Aliases and New clauses, to parameters of the operation, to the predefined identifiers: self, result (if it returns a result), exception (if it

can raise an exception) and sender, to message instances, or to entities navigated to from any of the previous ones.

Well-formedness rules for the expressions used in Operation Schemas are given in [StS01].

6.5 Operation Schemas: Constructs, Rules, Assumptions, and Interpretations

In this section, I introduce a number of constructs that are used to make specifying operations with Operation Schemas less cumbersome and more intuitive. I also propose a number of rules, assumptions, and interpretations that apply to Operation Schemas, and explain their meaning and utility in the context of specifying operations with them.

To be able to write and read Operation Schemas, it is necessary to understand the meaning of the various constructs used and the rules and assumptions that the model imposes. Before I explain the various rules, assumptions, and interpretations that apply to Operation Schemas, I first introduce an additional language construct that is used when explaining these aspects of Operation Schemas.

Additional Language Construct: commercial-and

Non-trivial pre- and postconditions will consist of multiple constraints on the system, where each constraint is described by a boolean expression that asserts a property that must hold true at the respective moment (i.e., before or after the operation). The condition is therefore defined by the conjunction of each of these *constraint atoms*. For reasons of readability, it is advantageous to highlight these constraint atoms and minimize the visibility of the conjunction operator that glues these atoms together to form the condition. Standard practice in OCL is to use the keywords “pre” and “post” to separate each constraint atom [Omg01]. However, this approach can become quite heavy and disruptive for the reader of large specifications.

An alternative is to use a special character to separate the constraint atoms, where the use of a single character offers minimal visual clutter. In this direction, commercial-and “&” is a good candidate, having an intuitive relationship to conjunction. Consequently, commercial-and is used as a constraint atom separator in Operation Schemas.

The commercial-and and the logical-and operators are logically equivalent, except commercial-and has the lowest precedence of all operators. Note that in OCL logical-and has a higher precedence than logical-implies, and therefore parentheses are not needed in the use of logical-implies with commercial-and as atom separator.

Contrasting the two styles, the following two postconditions are logically equivalent: the one on the left-hand side is written using the multiple pre/post keyword style of OCL, and the one on the right-hand side is written using the proposed commercial-and style.

```
post:  effectA                       | Post:  effectA &
post:  if condA and condB then       |        if condA and condB then
          effectB and                |           effectB &
          effectC                    |           effectC
       else                          |        else
          effectD                    |           effectD
       endif                         |        endif &
post:  effectE                       |        effectE
```

6.5.1 Asserting Creation and Deletion in Operation Schemas

The context in an Operation Schema is the system. The system consists of a (state) population of objects (i.e., instances of classes) and association links at any point in time. With respect to the system, an object exists only if it is part of the system's state population. In Operation Schemas, instead of explicitly asserting that an object is created or destroyed with the execution of an operation, one rather defines creation and destruction in terms of what is part of state population of the system and what ceases to be part of it. One can judge an object as part of the system state if it has a composition link either directly or transitively with the system, e.g., an object that is a component of a component of the system is also part of the state population by transitivity. Thus, "creation" of an object is asserted by stating that a composition association link between the object and the system (direct or transitive) was formed, and "destruction" of an object is asserted by stating that the composition association link between the object and the system was removed. Taking into account the addition and removal of objects from the state population of the system, Rule 1 is as follows:

Rule 1: An object is part of the state population of the system if and only if it has a direct or transitive composition association link with the system.

For example, given the Concept Model in figure 6.1, one can express the postcondition of an operation that adds a new customer to the lottery ticket manager in the following way:

Operation: LotteryTicketManager::addNewCustomer (...);
Aliases: newCust: Customer;
Post: self.customer = self.customer@pre->including (newCust) ...

Note that it is also possible to use the oclIsNew property with the same meaning, i.e., newCust.oclIsNew ().

Equally, one can express the postcondition of an operation that removes a customer from the lottery ticket manager in the following way (note that OCL does not have a property like oclIsRemoved):

Operation: LotteryTicketManager::removeCustomer (c: Customer);
Post: self.customer = self.customer@pre->excluding (c);

Note that it is not necessary to assert that an association link is "created". Creation of association links is implicit in Operation Schemas, i.e., it is only necessary to assert that navigation between two objects is possible for the creation of the link to be implied. Similarly to link “creation”, link “destruction” is also implicit, i.e., it is only necessary to assert that navigation between two objects is no longer possible for the destruction of the link to be implied. However, an association object is a special case (i.e., objects of association classes, e.g., Bought in figure 6.1). According to UML [pp. B-4, Omg01] an association class is:

“A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.”

I propose that one explicitly asserts that an association object becomes part of the system’s state population with the operation’s execution by asserting the oclIsNew property of the object, for example. This means an association object’s existence in the system is defined by its composition association link with the system. Also, the composition link can be used in navigation expressions to collect all association objects of an association class, for example. However, with this approach, care must be taken that the association object is also made part of the corresponding association, otherwise the Post clause would be incomplete, i.e., the object does not make sense if it is not a member of the corresponding association.

In the case of removing an association object from the system, it is necessary to either remove the link that it represents from the association or remove the composition link it has with the system. The two-way approach in “destruction” is taken purely for reasons of convenience.

6.5.2 Consistency of Associations

Revisiting the Post clause for the removeCustomer operation (shown above), what happens if the object c is also involved in other associations at the time when the composition link between c and the system is removed?

Answering this question directly: all association links of the system only make sense if they relate existing objects. It is a well known consistency constraint for object models that when an object is removed from the state population of the system that all association links connected to it have to be removed too. Although it would be possible to explicitly state all association links that must be destroyed, this is quite cumbersome in the presence of numerous associations. In light of this fact, it is implicit in the use of Operation Schemas that association links of no longer existing objects (wrt. the system) are removed. A consequence of this approach is that all associations with instances that are implicitly changed according to this rule also need to be mentioned in the Scope clause of the respective Operation Schema.

With respect to the consistency of associations constraint placed on Operation Schemas, Rule 2 is as follows:

Rule 2: Removal of an object from the system implies implicitly that all association links in the system that included the destroyed object are also destroyed.

For example, given the Concept Model in figure 6.1, the removal of a ticket from the lottery ticket manager requires only that the composition link is asserted to be removed:

```
Operation: LotteryTicketManager::removeTicket (t: Ticket);
Post:   self.ticket = self.ticket@pre->excluding (t);
```

Thus, it would be *unnecessary* to write the following, because it is implied by Rule 2:

```
Post:   self.ticket = self.ticket@pre->excluding (t) &
        t.seller->isEmpty () & -- unnecessary due to Rule 2
        t.owner->isEmpty () & -- unnecessary due to Rule 2
        t.draw->isEmpty (); -- unnecessary due to Rule 2
```

6.5.3 Frame Assumption

Looking at the postcondition as a whole, it describes the effect of the system operation on the system state. Each assertion describes a constraint on the change to an object, an association link or a collection of these ones. However, a question that can be posed is: What happens to those objects/links that are not mentioned by the postcondition? Do they stay unchanged? Addressing this problem, I now look at the frame assumption for Operation Schemas.

According to [Mor94], the frame of an operation specification is the list of all variables that can be changed by the operation. In an Operation Schema, this list is a subset of all those objects and association links that have their type included in the Scope clause. The postcondition of a specification describes all the changes to the frame variables, and since the specification is declarative, it is assumed that the postcondition must also state all the frame variables that stay unchanged. The reason is simple: if the unchanged frame variables were left unmentioned, they would be free to be given any value, and the result would still conform to the specification.

In writing pre- and postconditions for Operation Schemas, it is at least convenient to change the above "open world" assumption to a "closed world" one, because the goal is to write complete functional descriptions of operations (with respect to the state space of the system). Note that it is also possible, and in some cases even desirable, to write partial descriptions; however, this aspect is not addressed in this work. Interested readers are referred to [Ja95] for a discussion.

Currently, many formal approaches, such as, Z [Spi89], VDM [Jon86], Larch [GHW85], etc. explicitly state what happens to each frame variable—even for those variables that stay unchanged. This approach soon becomes cumbersome to write and error-prone, particularly for specifications that have complex case distinctions (where the complete frame is the combination of all the variables read/changed in each different case). One approach that avoids this extra work is to imply a "... and nothing else changes" rule when dealing with these types of declarative specifications [BMR93, Lan95, BMR95]. This means that the specification implies that the frame variables are changed according to the postcondition, but that all other (unmentioned) frame variables are left unchanged. This approach reduces the size of the specification, thus increases its readability, and makes the activity of writing specifications less error prone. This convention is therefore adhered to with Operation Schemas.

However, there is a slight problem with this assumption in the case of implicit removal—a consequence of the association consistency assumption (Rule 2). For example, let us reconsider the Operation Schema for the removeTicket system operation (of the lottery ticket manager).

```
Operation: LotteryTicketManager::removeTicket (t: Ticket);
Post:   self.ticket = self.ticket@pre->excluding (t);
```

If one strictly applies the frame assumption "... and nothing else changes", the associations Bought, Sells, and Has would stay unchanged because they are left unmentioned. Taken literally, this would lead to an inconsistent system state. Thus, it is necessary to take into account Rule 2, stated in section 6.5.2, for the frame assumption of Operation Schemas.

In addition, there is another case to consider: What happens to attributes of new objects that are not mentioned in the postcondition? One could imagine four different (but non-exclusive) approaches for this situation[1]: 1) attributes of new objects that are not mentioned in the postcondition can take any consistent value; 2) the unmentioned attributes have unknown values, where any expressions that attempt to navigate to their values are undefined; 3) the unmentioned attributes get predefined default values; or 4) the specification is incorrect if a value is not given to the respective attributes. The first approach is not sufficiently restrictive, because the value given to the attribute may be unknowingly used as if it contains a credible value. The second one takes a direct approach to unmentioned attributes. And, the third and fourth approaches require the specifier to constrain the value of the attribute to one that indicates "I am uninitialized". Using these approaches, the datatypes would need to be chosen carefully. For example, given a numOfFish attribute of type Natural, which is part of an object of class Lake, it is difficult to know what value one can give to this attribute to indicate that it is "uninitialized", because the top-end limit is uncertain, i.e., a large, nutrient-rich lake could have an enormous number of fish, and the bottom-end limit could be zero, i.e., clearly a lake that is overly acidic would have zero fish. In such cases, it would be necessary to extend the datatype, e.g., NumFish, by including some value which could stand for uninitialized, e.g., -1. However, such workarounds are not in the spirit of conceptual descriptions.

The approach chosen is the second one, which also allows predefined default values (third approach), and it is stated as Rule 3:

Rule 3: All unmentioned attributes of a newly created object that have predefined default values are given those default values, but if there exists an unmentioned attribute that does not have a default value then its value is unknown, where any expression that attempts to navigate to its value is undefined.

1. Recall that all attributes of objects are datatypes.

Taking into account Rule 2 and Rule 3, the revised frame assumption is given by Rule 4, below.

Rule 4: No frame variables of the system are changed with the execution of the operation other than those that are explicitly mentioned to be changed by the post-condition, the associations that are implicitly modified as defined by Rule 2, and the unmentioned attributes of new objects in the system that are treated according to Rule 3.

For example, let us consider the Operation Schema for the makeDrawAndRemoveOldest system operation (of the lottery ticket manager), which creates a new draw and removes the oldest one along with all the tickets that were part of that draw.

```
Operation: LotteryTicketManager::makeDrawAndRemoveOldest (num: LotDrawNum,
                                                    drawnOn: Date);
Aliases:    newDraw: Draw;
            oldDraw: Draw Is self.draw->any (old | self.draw->forall (d | old.date <= d.date));
Post:   newDraw.oclIsNew () & -- newDraw has become a new member of the system
        newDraw.drawDate = drawnOn &
        newDraw.drawNum = num &
        self.draw = self.draw@pre->excluding (oldDraw)->including (newDraw) &
                                        -- old one removed from and new one added to system
        self.ticket = self.ticket@pre - oldDraw.ticket; -- all tickets of that draw are removed from system
```

With respect to this postcondition, the frame assumption states that the only changes made to the state population of the system were: newDraw was added to the system and its attributes drawDate, drawNum and winningNums were (implicitly) given values, oldDraw was removed from the system, all ticket objects associated to oldDraw were removed from the system, and all association links that the removed objects were involved in no longer exist (Rule 2). It is important to note that winningNums is constrained to be given its default value, which is the empty set (shown in the Concept Model).

Looking closer at the Post clause, the fourth line of the Post clause states that the set of draw objects in the system (immediately) after the operation is equivalent to the set of draw objects in the system before the operation minus oldDraw but plus newDraw. However, looking at the first line of the Post clause, one can see that newDraw has already been asserted as a new member of the system state population. In an attempt to remove the redundancy, it would *not* be a good idea to remove the postfixed expression ->including (newDraw) from line four of the Post clause, because this would create a contradiction with line one of the Post clause. Note that removing line one of the Post clause would eliminate the redundancy.

6.5.4 Incremental Descriptions

In describing the effects of an operation, it can be useful to construct a postcondition in an incremental fashion, i.e., a particular effect of an operation may be defined by a combination of constraints that are defined at different places throughout the postcondition[1]. However, due to the declarative nature of the description, operators that can be used in an incremental fashion, e.g., the collection operator includes, do not relate the post-operation state of the variable to its pre-operation state. Consequently, the use of such operators is not sufficiently restrictive. To illustrate the issue, consider an operation (of the lottery ticket manager) called applyCriteria1ToFavoriteNums that adds numbers to a customer's favorite numbers attribute, according to the following criteria: add 10 to the customer's favorite numbers if his/her branch postcode is 1000, add 12 to the customer's favorite numbers if s/he bought a ticket in December, and add 99 to the customer's favorite numbers if s/he has bought more than 50 tickets. Using an incremental approach, the Post clause of the Operation Schema for the applyCriteria1ToFavoriteNums operation could be described in the following way:

```
Operation: LotteryTicketManager::applyCriteria1ToFavoriteNums (cust: Customer);
Post:  cust.myBranch.postcode = 1000 implies cust.favoriteNums->includes (10) &
       cust.bought->exists (b | b.date.month = 12) implies cust.favoriteNums->includes (12) &
       cust.myTickets->size () > 50 implies cust.favoriteNums->includes (99);
```

The postcondition satisfies the demands of the criteria, i.e., in the appropriate cases, the correct numbers were constrained to be members of the customer's favorite numbers. However, the condition is too weak, because it leaves cust.favoriteNums completely open to take any other numbers as well, according to the "open world" nature of declarative specifications.

As a consequence, one is forced to construct such constraints by defining the change that is made to the variable in terms of it pre-operation state. Nevertheless, it would not be correct to write the following Post clause for the applyCriteria1ToFavoriteNums operation.

```
Operation: LotteryTicketManager::applyCriteria1ToFavoriteNums (cust: Customer);
Post:  cust.myBranch.postcode = 1000 implies
              cust.favoriteNums = cust.favoriteNums@pre->including (10) &
       cust.bought->exists (b | b.date.month = 12) implies
              cust.favoriteNums = cust.favoriteNums@pre->including (12) & --a possible contradiction
       cust.myTickets->size () > 50 implies
              cust.favoriteNums = cust.favoriteNums@pre->including (99); -- a possible contradiction
```

If cust.favoriteNums is empty then the postcondition is inconsistent if two or more of the implies-conditions evaluate to true (i.e., the expressions on the left-hand-side of the implies expressions).

Currently, the only way to state the postcondition for the applyCriteriaToFavoriteNums operation, so that it ensures that only the intended changes are made, is the following.

```
Post:
   cust.myBranch.postcode = 1000 and cust.bought->exists (b | b.date.month = 12) and
      cust.myTickets->size () > 50 implies
         cust.favoriteNums = cust.favoriteNums@pre->union (Set {10, 12, 99}) &
   not cust.myBranch.postcode = 1000 and cust.bought->exists (b | b.date.month = 12) and
```

1. In section 6.6, I further motivate the need for using an incremental approach.

```
        cust.myTickets->size () > 50 implies
            cust.favoriteNums = cust.favoriteNums@pre->union (Set {12, 99}) &
    cust.myBranch.postcode = 1000 and not cust.bought->exists (b | b.date.month = 12) and
        cust.myTickets->size () > 50 implies
            cust.favoriteNums = cust.favoriteNums@pre->union (Set {10, 99}) &
    cust.myBranch.postcode = 1000 and cust.bought->exists (b | b.date.month = 12) and
        not cust.myTickets->size () > 50 implies
            cust.favoriteNums = cust.favoriteNums@pre->union (Set {10, 12}) &
    not cust.myBranch.postcode = 1000 and not cust.bought->exists (b | b.date.month = 12) and
        cust.myTickets->size () > 50 implies
            cust.favoriteNums = cust.favoriteNums@pre->including (99) &
    not cust.myBranch.postcode = 1000 and cust.bought->exists (b | b.date.month = 12) and
        not cust.myTickets->size () > 50 implies
            cust.favoriteNums = cust.favoriteNums@pre->including (12) &
    cust.myBranch.postcode = 1000 and not cust.bought->exists (b | b.date.month = 12) and
        not cust.myTickets->size () > 50 implies
            cust.favoriteNums = cust.favoriteNums@pre->including (10) &
--note that one does not need to state the case when no change to the set is required (according to the frame assumption)
```

Clearly, if I defined an expression-substitution (Aliases clause) for each of the three conditions, the postcondition would be smaller. The same number of cases would nevertheless be needed.

For the purposes of facilitating incremental description of sets in an Operation Schema, the minimum set principle is introduced.

6.5.4.1 Minimum Set Principle

The minimum set principle allows one to express incrementally the elements that are added and/or removed from a set with the execution of an operation. Unless asserted otherwise, all sets that can be constructed by navigation are subject to the minimum set principle in the postcondition (Post clause) of an Operation Schema. The minimum set principle is defined in the following way.

Definition 6.4: The minimum set principle states that each set referred to in a postcondition is the smallest possible set that fulfills the postcondition and contains all the elements of the original set (i.e., the contents of the set before the operation) minus the elements whose removal is asserted in the postcondition.

The minimum set principle is complementary to the frame assumption (Rule 4): together they state, more or less, that nothing changes other than what is explicit in the postcondition.

Applying the minimal set principle to the Post clause of the applyCriteriaToFavoriteNums operation shown earlier, one is able to write the following postcondition and get the desired effect:

```
Post:
   cust.myBranch.postcode = 1000 implies cust.favoriteNums->includes (10) &
   cust.bought->exists (b | b.date.month = 12) implies cust.favoriteNums->includes (12) &
   cust.myTickets->size () > 50 implies cust.favoriteNums->includes (99)
```

With respect to the above Post clause, intuitively the minimum set principle states that cust.favoriteNums is equivalent to the cust.favoriteNums@pre with the addition of 0, 1, 2, or 3 different numbers (i.e., 10, 12, and/or 99) depending on which of the three implies-conditions hold in the postcondition.

Clearly, this version is markedly shorter and easier to read/write/understand (compared to the previous Post clause). Note that the relationship between the two approaches is exponential: there are 2^n - 1 separate cases for n cases. Hence, the usefulness of an incremental description becomes even more noticeable as n gets larger.

The minimum set principle can also be applied to collections in general. If the minimum set principle is applied to a bag, duplicates are not accounted for. For example,

```
Pre:   bagX->isEmpty ();
Post:  bagX->includes (x1) &
       bagX->includes (x1);
```

is equivalent to:

```
Post:  bagX->includes (x1);
```

therefore the second “includes” expression is redundant. An additional constraint is required to assert that the bag contains two x1 elements, e.g., bagX->count(x1) = 2. In fact, this is a necessary condition for one to be able to reason about the size of the bag, otherwise the resulting bag could be any size, i.e., it could contain any number of repeated x1 elements and still conform to the postcondition.

Beyond the advantages of avoiding case distinction explosion, the incremental style, facilitated by the minimum set principle, has an advantage over the “all in a single condition” style in that it allows for easier maintenance of case distinctions in postconditions. For example, there could exist an operation (of the lottery ticket manager) that allows a branch to swap a customer with another branch in the following way.

```
Operation: LotteryTicketManager::swapMembers (cust1: Customer, cust2: Customer,
                                              b1: Branch, b2: Branch);
Post:  b1.members->excludes (cust1) &   -- the link between b1 and cust1 is removed
       b2.members->includes (cust1) &   -- the link between b2 and cust1 is added
       b2.members->excludes (cust2) &   -- the link between b2 and cust2 is removed
       b1.members->includes (cust2);    -- the link between b1 and cust2 is added
```

To illustrate a change to the Post clause, one could imagine in maintenance activities that it is necessary to add another customer to branch b1 as part of the operation. The change to the Operation Schema would be the following:

```
Operation: LotteryTicketManager::swapMembers (cust1: Customer, cust2: Customer,
                                           cust3: Customer, b1: Branch, b2: Branch);
Post:  b1.members->excludes (cust1) &
```

```
          b2.members->includes (cust1) &
          b2.members->excludes (cust2) &
          b1.members->includes (cust2) &
          b1.members->includes (cust3); -- the link between b1 and cust3 is added
```

This example shows the constructive nature of postconditions that apply the minimum set principle.

Advocating the use of the minimum set principle, Rule 5 is as follows:

Rule 5: The minimum set principle is applied to the interpretation of sets in the postconditions of Operation Schemas.

It would be valuable to be able to use a similar constructive approach for the incremental description of other data structures too. In this direction, incremental plus and minus for numeric types are discussed in the next subsection.

6.5.4.2 Incremental Plus and Minus

Incremental plus and minus can be used to construct the value of numeric types in a Post clause of an Operation Schema. Incremental plus and minus can be realized in a Post clause by the use of the following operators: "+=" and "-=". These operators have the following meaning: the post-operation value of a numeric type is equivalent to its pre-operation value plus the sum of all values on the right-hand sides of all += operators used in the postcondition that refer to the value of the numeric type, minus the sum of all values on the right-hand sides of all -= operators that refer to the value of the numeric type. This means that the following postcondition, for example:

```
Post:   obj.x += 5 &
        obj.x -= 4;
```

is equivalent to this one:

```
Post:   obj.x = obj.x@pre + 1;
```

Taking a larger example, there is an operation (of the lottery ticket manager) that applies a new ticket valuation scheme to a given ticket called applyNewValueSchemeToTicket. The Operation Schema for this operation takes advantage of incremental plus and minus to describe the postcondition (Post clause), shown below.

```
Operation: LotteryTicketManager::applyNewValueSchemeToTicket (t: Ticket);
Post:   t.seller.postcode > 2000 implies
                t.value += 100 & -- an additional 100 if the seller has a postcode greater than 2000
        t.seller.members->size > 100 implies
                t.value += 20 & -- an additional 20 if the seller of the ticket has over 100 members
        t.owner.favoriteNums->intersection (Set {5, 10, 15})->notEmpty () implies
                t.value += 50 & --an additional 50 if the owner of the ticket has 5,10,15 as one of his/her favorite #s
        t.bought.date.year = 2000 implies
                t.value += 10 & -- an additional 10 if the ticket was bought in the year 2000
```

```
t.draw.drawNum < 15 and t.value > 100 implies
        t.value -= 10;   -- withdrawal of 10 if the ticket was for a draw that had a num less than 15
                         -- and the final value of the ticket is greater than 100
```

For this type of postcondition, the traditional approach would have required that one enumerates each case (all 31 of them!).

Nevertheless, care needs to be taken when the incremental style is mixed with the other styles.

```
Post: ...
   obj.x += 5 &                -- line one
   obj.x -= 4 &                -- line two
   obj.x = obj.x@pre + 2 &     -- line three
   obj.x = 2                   -- line four
```

The above example is an erroneous specification: line three is in contradiction with the result defined by the incremental plus and minus, and line four would require that obj.x@pre be either 0 or 1 depending on whether line three was brought into agreement with line one and two or vice versa.

Apart from supporting incremental descriptions, an advantage of using incremental plus and minus is that it facilitates more concise expressions by reducing the size of constraints on numeric types. Unfortunately, the incremental plus and minus facility can not be extended to other types of numeric operations (e.g. multiplication), because it relies on the commutativity of additions and subtractions.

6.5.5 Shorthands and Additional Language Constructs

In this subsection, some shorthands and additional language constructs are defined to make the job of the specifier less laborious and therefore less error-prone.

Shorthand Expression: if-then-elsif*-else expressions

If-then-else expressions prove to be a common and natural way to define case distinctions. With multi-case distinction, if-then-else expressions can be considerably more concise than logical-implies expressions, and arguably easier to write and read by those specifiers that are more comfortable with imperative styles (due to the similarity of if-then-else expressions to if-then-else statements in imperative programming languages).

Often in writing pre- and postconditions, one comes across cases where the assertion of an effect is dependent on the truth of a series of conditions. In an imperative programming language, one usually has the facility of an "elsif"-like construct to define the hierarchy of conditions in a uniform manner. In OCL, it is only possible to construct such a hierarchy by nesting if-then-else expressions, i.e., placing if-then-else expressions within the else-part of other if-then-else expressions. However, this practice can become cumbersome as the depth of nesting increases, and it most probably has a negative impact on clarity for the reader and even the writer. Taking a leaf out of the imperative programming language book, I propose the use of an elsif-part for if-then-else expressions. The elsif-part can be seen simply as a shorthand for the nested version (much like if-then-else can be defined in terms of implies expressions), therefore the semantics of this addition is defined in terms of the semantics for standard if-then-else expressions. For example, below, the expression on the left-hand-side can be rewritten as the expression on the right-hand-side, i.e., the two expressions are logically equivalent:

```
if condA then            | if condA then
   effectB               |    effectB
elsif condB  then        | else
   effectB               |    if condB  then
else                     |       effectB
   effectC               |    else
endif                    |       effectC
                         |    endif
                         | endif
```

Shorthand Expression: if-then-endif expression

By promoting the use of if-then-else expressions for the purpose of case distinction, one can write conditional constraints in a Post clause in the following way, making use of LotteryTicketManager shown in figure 6.1 (cust is an object of type Customer):

```
if cust.myBranch.postcode = 1000 then
   cust.favoriteNums->includes (10)
else
   cust.favoriteNums = cust.favoriteNums@pre
endif
```

The above if-then-else expression results in true in the case that the branch of cust has a postcode of 1000 and cust's favorite numbers include the number 10, or cust's branch does not have a postcode of 1000 and cust's favorite numbers are the same before the operation as they are after the operation; otherwise the expression results in false.

However, taking into account the frame assumption, one is not required to state which variables stay unchanged; consequently, the else part, in this case, can be simply made true:

```
if cust.myBranch.postcode = 1000 then
   cust.favoriteNums->includes (10)
else
   true
endif
```

Clearly, in such cases it seems somewhat unnecessary to write "else true". Consequently, I propose that in such cases the "else true" part can be made implicit. Illustrating this proposal on the same example, one would get the following (equivalent) expression:

```
if cust.myBranch.postcode = 1000 then
   cust.favoriteNums->includes (10)
endif
```

Note that the resulting type of an "if-then-endif" is always boolean. In terms of semantics, this addition is proposed as a shortcut, and therefore the semantics are defined in terms of if-then-else expressions.

Additional Language Construct: aggregate

Writing constraints on objects and messages (see section 5.7) can be quite cumbersome, verbose and even error-prone if one does not have a mechanism for defining their *composite values*. The composite value of an object is defined as the combination of all its value attributes (recall that according to a Concept Model, all attributes of objects are data values). And the composite value of a message is defined as the combination of all its parameters: object references and/or data values.

For the purpose of denoting composite values of both objects and messages, an Ada-style *aggregate* notation for denoting composite values is introduced [Bar96]. An aggregate is written by associating a value with each attribute, denoted by its name. The following aggregate defines the composite value of a Ticket object (conforming to the Concept Model shown in figure 6.1):

```
(cost => 5, value => 5.0E6, chosenNums => Set {5, 7, 23, 27, 32})
```

An advantage of aggregates is that related values are kept together in one place. An aggregate is more than just a shorthand, defined in terms of standard expressions. It is an additional construct of the language used by Operation Schemas, and therefore can be seen as an extension of OCL.

Additional Language Construct: all property

Due to the "by-reference" semantics of objects and messages, one can not refer to their composite value directly. For this purpose, the property "all" is introduced to denote the composite value of an object or message (again taken from Ada [Bar96]). The all property can be useful in comparing values of objects/messages, and as will be shown in the next subsection, it can be used to assert that an object is created as a clone of another object.

The all property can be used to write expressions like the following, given that t is an object of type Ticket:

```
t.all = (cost => 5, value => 5.0E6, chosenNums => Set {5, 7, 23, 27, 32})
```

This expression evaluates to true if the object referenced by t has the corresponding attribute values. The above condition is equivalent to:

```
t.cost = 5 &
t.value = 5.0E6 &
t.chosenNums = Set {5, 7, 23, 27, 32}
```

Therefore in general, an aggregate can be used to denote the value of an object or message, and consequently the following (generic) expression:

```
x.all = (a1 => v1, a2 => v2, ...)
```

is equivalent to:

```
x.a1 = v1 & x.a2 = v2 & ...
```

The all property is more than just a shorthand that can be defined in terms of standard expressions. It is an additional construct of the language used by Operation Schemas, and therefore can be seen as an extension of OCL. One way to realize this extension would be to add the "all" property to the standard OCL type *OclAny*.

Shorthand: parameterized oclIsNew

Taking advantage of the aggregate notation, it would be useful to parameterize the oclIsNew property of objects in OCL by an aggregate that denotes all the (initial) attribute values of the object or all the actual parameters of a message. Thus, the oclIsNew property can optionally take a composite value as parameter. For example, in a Post clause one could assert the following:

```
newTicket.oclIsNew ((cost => 5, value => 5.0E6, chosenNums => Set {5, 7, 23, 27, 32}))
```

which means that newTicket has become part of the system with the execution of the operation, and all its attributes were initialized according to the composite value. This expression is logically equivalent to:

```
newTicket.oclIsNew () &
t.cost = 5 &
t.value = 5.0E6 &
t.chosenNums = Set {5, 7, 23, 27, 32}
```

The proposed notation ensures that all attributes of a newly created object were constrained to the given values, and none of them were forgotten.

In general, it is possible to define the parameterized oclIsNew property in terms of standard expressions and the "all" property. Therefore, the following (generic) expression:

```
newX.oclIsNew ((a1 => v1, a2 => v2, ...))
```

is equivalent to / and can be seen as a shorthand for:

```
newX.oclIsNew () & newX.all = (a1 => v1, a2 => v2, ...)
```

Taking into account the expanded form of the shorthand for the parameterized oclIsNew property, there is nothing stopping one from using any kind of composite value as parameter. For instance, one could assert that a new object is a clone of another object, i.e., in a Post clause one could write the following expression:

```
newTicket.oclIsNew (otherTicket.all)
```

which means that newTicket has become part of the system with the execution of the operation and all its attributes values match those of otherTicket.

Therefore, in general, the parameter of the oclIsNew property may be any composite value, where the following (generic) expression:

newX.oclIsNew (*value*)

in its expanded form is the following one.

newX.oclIsNew () & newX.all = *value*

In terms of semantics, the parameterized oclIsNew property can be defined in terms of other expressions, and therefore it can be seen simply as a shorthand.

Shorthand: oclIsNew for collections

In addition to using aggregates with oclIsNew, it would seem useful to allow oclIsNew to be a property of collections. In this case, the oclIsNew property would optionally take a natural number as parameter. For example, in a Post clause one could assert the following:

```
aBunchOfNewTickets.oclIsNew (32)
```

which means that there are 32 new tickets that have become part of the system with the execution of the operation, and they are members of the collection aBunchOfNewTickets. This expression is logically equivalent to:

```
aBunchOfNewTickets -> forall (t | t.oclIsNew () ) &
aBunchOfNewTickets -> size () = 32
```

The proposed notation allows one to concisely assert the creation of a number of new objects.

The signature of the property for collections is the following:

```
oclIsNew (size: Integer): Boolean
```

It takes a single parameter, having the meaning that the specified number of objects are newly created and they are members of the collection.

In terms of semantics, “oclIsNew for collections” can be defined in terms of other expressions, and therefore it can be seen simply as a shorthand.

6.6 Reusing Parts and Reducing the Size of Operation Schemas by Functions and Parameterized Predicates

In this section, I introduce two mechanisms that can be used in Operation Schemas for the purposes of reusing predicates and functions, and also for regulating the size of Operation Schemas, so that they do not become overly large.

The incremental style of description, proposed in section 6.5.4, the frame assumption, defined in section 6.5.3, and the various shorthands, proposed in section 6.5.5, help to make Operation Schemas more concise. These approaches by themselves however do not address the ease of writing and reading Operation Schemas that have a large number of constraints. Finney et al. have empirical evidence from controlled experiments that suggests that structuring a specification into schemas of about 20 lines significantly improves comprehensibility over a monolithic specification [FFF99]. Using this result as a guideline, it is necessary to offer the specifier mechanisms to structure large Operation Schemas into a form that obeys this guideline.

To this end, one common approach is to use multiple pre- and postcondition pairs for structuring operations. This approach was originally proposed by Wing, and she called it case analysis [Win83]. The idea is that each pre and post pair defines a distinct case, where partial correctness is defined as follows: $\bigwedge_i pre_i \Rightarrow post_i$ (where a different value of i represents a different pair). However, this approach is subject to the case explosion problem, which was demonstrated in section 6.5.4, because a pre and post pair is required for every single case. Furthermore, this style of structuring can make the precondition of the operation difficult to understand for the client: this is an important point, because the correct use of the operation relies on the fact that the operation is called only when the precondition holds.

Another approach is to provide mechanisms for defining constraints and functions that can be used across many Operation Schemas. Expression-substitutions of an Aliases clause already allow the declaration of constraints. However, they do not support parameterization, which would increase the opportunities of reuse, and they are local to an Operation Schema. Extending the concept of expression-substitution, I propose Parameterized Predicates, discussed in section 6.6.1. Also, it can be useful to define query functions, i.e., functions that do not have side effects. In section 6.6.2, I therefore propose a means to describe such functions.

6.6.1 Parameterized Predicate

A *Parameterized Predicate* can be used in Pre and Post clauses to better support readability of schemas and to allow one to reuse commonly recurring predicates. They are inspired from the concept of "effect" proposed in Catalysis [DW98], but their application is more general than the description of an effect, i.e., a Parameterized Predicate is a parameterization of any predicate (where it makes sense to parameterize it).

A Parameterized Predicate can be seen as a homogenous 'piece' of the pre- or postcondition, and therefore it can use the suffix '@pre' (in the case that it is destined for postconditions). Also, a Parameterized Predicate can be seen as a boolean (query) function.

Parameterized predicates are defined using OCL definitions. UML describes an OCL definition in the following way [Omg01, page 6-8]:

> *"... «definition» Constraint must be attached to a Classifier and may only contain let definitions. All variables and operations defined in the «definition» constraint are known in the same context as where any property of the Classifier can be used. In essence, such variables and operations are pseudo-attributes and pseudo-operations of the classifier. They are used in an OCL expression in exactly the same way as attributes or operations are used."*

Rather than using a let expression, I use a different syntax for declaring a Parameterized Predicate, which I believe better serves the specifier and reader. The syntax is the following:

```
ParameterizedPredicate =
    "Predicate" ":" SystemTypeName "::" PredicateName "(" [ParameterList] ")"  ";"
    [ "Aliases" ":" (ExpressionSubstitutionDecl ";")* ]
    "Body" ":" Condition ";"
```

The Aliases clause allows the declaration of expression-substitutions, and it can be used in the same way as the clause of the Operation Schema with the same name. For the purposes of better identifying Parameterized Predicates, I propose the following naming convention for Parameterized Predicates: the suffix "_p" is attached to the end of Parameterized Predicate names. For example,

```
Predicate: SystemX::parameterizedPredicate1_p (parm1: X);
```

The context for a Parameterized Predicate is the system, i.e., self refers to the system. Navigation expressions used in a Parameterized Predicate can be constructed from self and any parameter that is passed to it. However, it must be possible to resolve all references within this context.

To illustrate a Parameterized Predicate, consider a predicate that encapsulates the effect of linking a draw with a set of tickets, and ensuring that all tickets of this draw have a given cost and value. The following Parameterized Predicate could be defined for this purpose:

```
Predicate: LotteryTicketManager::updateDraw_p (d: Draw, tickets: Set (Ticket),
                                               cost: Money, value: Money);
Body:
  d.ticket->includesAll (tickets) &
  d.ticket->forall (t | t.cost = cost and t.value = value);
```

The Parameterized Predicate can then be used in the Post clause of an Operation Schema, for example, in the following way:

```
Post: ...
  if updateTickets then -- updateTicket is an expression-substitution defined elsewhere
    self.updateDraw_p (draw123, setOfNewTickets, 5, 5E6) &
    ...
```

The above Post clause shows that the Parameterized Predicate is navigated to from self.

Also, a Parameterized Predicate should not be suffixed by a decorator, i.e., @pre. In such circumstances, the containing expression is ill-formed. For example, the following expression is ill-formed:

```
self.updateDraw (draw123, setOfNewTickets, 5, 5E6)@pre -- this is illegal
```

The use of the pre-defined identifier self in OCL is optional. It is arguably more readable to "leave off" the self identifier, and therefore I propose that "instantiated" Parameterized Predicates do not have the self identifier prefixed. For example, the Post clause shown above would be rewritten in the following way:

```
Post: ...
  if updateTickets then
    updateDraw_p (draw123, setOfNewTickets, 5, 5E6) &
    ...
```

6.6.2 Function

A *Function* may be used to encapsulate a computation of a query. A Function does not have any side effects, and, to the contrary of a system operation, it does not change the system state. Functions may be used as a reuse mechanism for commonly recurring calculations. Also, they can be seen as special query methods of the system. Unlike system operations, they are not public and therefore are not visible to the outside world[1].

Following the practice of a number of modular programming languages, I propose to separate the Function declaration (its signature) from the Function definition [Bar96]. In that way, they can be used as a placeholder when the need for the Function is known, but its realization is deferred to a later stage of development, i.e. design or implementation. The syntax of the Function declaration is the following:

"**Function**" ":" SystemTypeName "::" FunctionName "(" [ParameterList] ")" [":" TypeName] ";"

For the purposes of better identifying Functions, I propose to attach the suffix "_f" to the end of Function names. For example, consider a Function that determines the most popular winning number of the LotteryTicketManager for a given branch. The declaration for this Function is shown below.

Function: LotteryTicketManager::mostWinningAndPopularNumber_f (b: Branch): LotNum;
-- A Function that hides the algorithm for choosing the most popular winning number for a branch

The syntax for the definition of the Function body is the following:

"**Function Body**" ":" SystemTypeName "::" FunctionName" "("[ParameterList]")" [":" TypeName] ";"
["**Aliases**" ":" (ExpressionSubstitutionDecl ";")*]
["**Pre**" ":" Condition ";"]
"**Post**" ":" Condition ";"

The Aliases clause allows the declaration of expression-substitutions, and it can be used in the same way as the clause of the Operation Schema with the same name. Also, the Condition in the Post clause must define result. result is a predefined identifier of OCL that represents the result of a function. The type of result matches the return type, i.e., TypeName in the grammar.

The context for a Function is the system, i.e., self refers to the system object. Navigation expressions used in a Function can be constructed from self or from any parameter that is passed to it. However, it must be possible to resolve all references within this context.

A possible definition of the mostWinningAndPopularNumber_f Function follows:

Function Body: LotteryTicketManager::mostWinningAndPopularNumber_f (b: Branch): LotNum;
Aliases:
  winNums: Bag (LotNum) **Is** b.sold.draw.winningNums;
  popularNums: Bag (LotNum) **Is** b.members.favoriteNums;

1. Note that if a system operation is side-effect free (i.e., it is an observer rather than a mutator) then it can also be used in the way that is proposed for Functions in this subsection (although the _f convention is not used). The idea is that all public operations of the system are described using Operation Schemas and Function declarations are only used to declare auxiliary operations.

```
    allNums: Bag (LotNum)  Is winNums->union (popularNums);
  Pre: b.sold->notEmpty() & b.members->notEmpty(); -- branch must have a member and sold at least a ticket
  Post: result = allNums->any (n | allNums->forall (n2 |
                                  allNums->count (n) >= allNums->count (n2)));
```

The above Post clause asserts that the result of the Function is equivalent to the most occurring number of the following combination: the winning numbers of draws for which the branch has sold tickets in, and the favorite numbers of customers of the branch.

The Function can then be used in the Post clause of an Operation Schema, for example, in the following way:

```
  Operation: LotteryTicketManager::opX (targetBranch: Branch, cust: Customer);
  Post: ...
    cust.favoriteNums->includes ( self.mostWinningAndPopularNumber_f (targetBranch) ) & ...
```

The above Post clause shows that the Function is navigated to from self.

Unlike Parameterized Predicates, a Function may be suffixed by a decorator in the Post clause (e.g., @pre). Attaching the @pre decorator to an expression that denotes the call to a Function means that it is evaluated at pre-operation time, otherwise it is evaluated at post-operation time. Note that this highlights that Functions need not be pure mathematical functions, because, for pure mathematical functions, it would not matter at which moment the function is called.

The choice as to which moment the function is evaluated may be important. For example, consider an Operation Schema for an operation called updateFavoriteNumsAndDestroyBranchTicketsAndMembers, which adds the most winning and popular number for a given branch to the favorite numbers of a given customer and destroys all the tickets and members for that branch. In this case, it would be necessary to attach the @pre decorator to the expression that denotes the function call. The Operation Schema for this operation is given below.

```
  Operation: LotteryTicketManager::updateFavoriteNumsAndDestroyBranchTicketsAndMembers (
                                  targetBranch: Branch, cust: Customer);
  Pre:
    targetBranch.sold->notEmpty() &
    targetBranch.members->notEmpty() &
    targetBranch.members->excludes (cust);
  Post:
    self.ticket->excludesAll (targetBranch.sold) &
    self.customer->excludesAll (targetBranch.members) &
    cust.favoriteNums->includes (self.mostWinningAndPopularNumber_f (targetBranch)@pre);
```

In the above Post clause, the Function mostWinningAndPopularNumber_f is called at pre-operation time, and therefore it will result in a certain number—unlike if it was called on the post-operation state, where all members and tickets of the branch are no longer part of the system state.

Functions can also be used when OCL is not suitable for expressing the algorithm, e.g. in the case of numeric computations. Functions are therefore a way to escape the limited expressive power of OCL when necessary. However, it is important to not abuse such a facility.

6.7 Messages

In this section, I describe how an Operation Schema can assert that messages are output with the execution of the operation. Also, I discuss the various kinds of messages that can be mentioned in an Operation Schema.

6.7.1 Role of Messages

An Operation Schema not only makes assertions about the state of the system after the execution of the corresponding operation, but it can also describe assertions about the messages that are output by the operation. Note that in UML, communications between the system and actors is through message instances (also referred to as stimuli in UML).

The execution of an operation may cause one or more message instances to be generated and placed in the system's output message queue. Once placed in the system's output message queue, the system's communications infrastructure delivers it to the designated actor instance(s). An output message instance corresponds to a request, notification, or reply, which is transmitted from the system to one or more actors.

6.7.2 Message Semantics

Message instances, as objects, have by-reference semantics and have unique identities. All messages have a property called destination, which denotes the set of addresses of the receiving actor(s). Therefore, the destination property of a message instance must contain all the addresses of the actor instances who are sent the message instance. Furthermore, a message instance that is a member of the system output queue is guaranteed to be delivered to its destination(s). Also, the order in which message instances are placed in the system's output queue is preserved by the delivery mechanism, i.e., two message instances destined for the same actor instance will arrive at the actor instance in the same order they were placed on the system's output queue. Rule 6 summarizes these two characteristics.

Rule 6: All communications between the system and its actors have guaranteed and in-order delivery semantics

6.7.3 Output Message Queue

All message instances generated by system operations are placed in the system's output message queue, ready for delivery to the target actor instances by the system's communication infrastructure. The system's output message queue holds references to message instances. The reference to a message instance is not removed from the output queue at the time that it is delivered to the actor instance(s), as one would intuitively imagine, but rather it continues to persist on the output queue for at least as long as the completion time of the respective operation. The reason for this constraint is that an Operation Schema describes message sending by making assertions about the state of the system's output queue in its Post clause. This means that the assertion must hold true immediately after the execution of the operation, and therefore all message instances that are generated by an operation must be present in the system's output queue at this time.

The output queue is a property of the system called outMessages. Therefore, in an Operation Schema, one can navigate to this property from the system in the following way: self.outMessages. The type of the property is a sequence of any kind of message instances.

6.7.4 Message Sending

In an Operation Schema, the Messages clause is used to declare all possible messages that may be output with the execution of a system operation. Illustrating the declaration of output messages, consider an operation that sends a message instance msg1 of type Message1 to actorX1 of type ActorX, and sends a message instance msg2 of type Message2 to actorX2 of type ActorX. The Messages and New clauses of the Operation Schema that describes this operation could be written in the following way:

Messages: ActorX::{Message1; Message2;};
New: msg1: Message1; msg2: Message2;

It is also possible to declare a collection of message instances in the New clause of an Operation Schema. Such a declaration indicates that possibly many messages of the same type are output with the execution of the operation. For instance, revisiting the example of the informWinners operation (see section 6.4.1) of LotteryTicketManager, one could make the following declarations:

```
Messages:  Client::{WonLottery;};
           Manager::{NumOfWinnersNotification; WinnerStatistics;};
New:       youWon: WonLottery;
           numWinners: NumOfWinnersNotification; stats: Set (WinnerStatistics);
```

The first line of the Messages clause declares that actors of type Client may be sent messages of type WonLottery. The second line declares that actors of type Manager may be sent messages of types NumOfWinnersNotification and WinnerStatistics. With respect to the New clause, the first line declares a message instance of type WonLottery called youWon. And the second line declares a message instance of type NumOfWinnersNotification called numWinners, and declares a set of message instances of type WinnerStatistics called stats.

6.7.4.1 Asserting Message Sending in the Post clause

The Post clause can be used to assert that message instances were sent to actors with the execution of the operation. This is achieved by asserting that the message instance is a member of the system's output queue, and asserting that it is addressed to the destination actor instance(s) and has the desired actual parameters.

To illustrate a Post clause that makes message sending assertions, consider the Post clause shown below (in addition to the Messages and New clause). It asserts that two message instances msg1 and msg2 are sent, assuming that actual1, actual2 and actual3 are the intended (and type correct) parameters for the messages.

```
Messages: ActorX::{Message1; Message2;};
New: msg1: Message1; msg2: Message2;
Post:
   msg1.oclIsNew ((parm1 => actual1)) &                       -- line 1
   self.outMessages -> one (m | m = msg1) &                   -- line 2
   msg1.destination = Set {actorX1} &                         -- line 3
   msg2.oclIsNew ((parm1 => actual2, parm2 => actual3)) &     -- line 4
   self.outMessages -> one (m | m = msg2) &                   -- line 5
   msg2.destination = Set {actorX2};                          -- line 6
```

The first line assert that the message instance msg1 was created with the execution of the operation and its parameter is initialized to the value defined by the aggregate. The second line asserts that msg1 is a member of the system's output queue and the output queue has only one reference to msg1. The third line asserts that the destination of msg1 is actorX1. Lines 4, 5 and 6 are similar to Lines 1,2 and 3.

In general, care must be taken that all parameters of an output message instance have defined values.

6.7.4.2 Shorthands for Message Sending Assertions

To increase the readability of the Post clause, a shorthand is introduced to make the assertion for message sending more concise. The syntax of the shorthand is the following:

```
_actor-instance-address_.sent (_message-instance_)
```

where _actor-instance-address_ denotes the address of the only actor instance who will receive the message instance, _message-instance_. This expression is a shorthand for the following (expanded but logically equivalent) expression:

```
self.outMessages -> one (m | m = _message-instance_) &
_message-instance_.destination = Set {_actor-instance-address_}
```

For example, the sent shorthand could be used to rewrite the Post clause shown above, as the one shown below:

```
Post:
    msg1.oclIsNew ((parm1 => actual1)) &                            -- line 1
    actorX1.sent (msg1) &                                           -- line 2
    msg2.oclIsNew ((parm1 => actual2, parm2 => actual3)) &          -- line 3
    actorX2.sent (msg2);                                            -- line 4
```

As illustrated by the example, the shorthand makes the Post clause considerably more concise.

Equally I define other useful shorthands for message sending assertions. These include shorthands for asserting that one message instance is sent to multiple actor instances, multiple message instances are sent to one actor instance, and multiple message instances are sent to multiple actors instances.

Shorthand: sentAllToOne

The shorthand notation used for asserting that multiple message instances are sent to one actor instance is the following[1]:

```
_actor-instance-address_.sentAllToOne (_collection-of-message-instances_)
```

and it is equivalent to/is a shorthand for:

```
_collection-of-messages-instances_->forall (m | m.destination = Set {_actor-instance-address_}
    and self.outMessages -> one (x | x = m))
```

Shorthand: sentOneToAll

The shorthand notation used for asserting that one message instance are sent to multiple actor instances is the following:

```
_collection-of-addresses-of-actor-instances_.sentOneToAll (_message-instance_)
```

and it is equivalent to/is a shorthand for:

```
_collection-of-addresses-of-actor-instances_.forall-> (a | a.sent (_message-instance_))
```

Note that the expanded form also makes use of the sent shorthand, i.e., it could be expanded into a even more elementary form.

Shorthand: sentAllToAll

The shorthand notation used for asserting that multiple message instances are sent to multiple actors instances is the following:

```
_collection-of-addresses-of-actor-instances_.sentAllToAll (_collection-of-message-instances_)
```

and it is equivalent to/is a shorthand for:

```
_collection-of-addresses-of-actor-instances_->forall (a |
    a.sentAllToOne (_collection-of-message-instances_))
```

Note that the expanded form makes use of the sentAllToOne shorthand.

1. Note that the underline symbol ("_") is used to delimit the placeholder for the expression, i.e., the actual name will be substituted at that spot.

Shorthand: precedesOnQueue

In some cases it may be necessary to define ordering constraints on messages with respect to their positions on the system's output queue. A useful constraint is to be able to assert that one message instance precedes another message instance on the system's output queue. For such situations, I propose the precedesOnQueue shorthand notation, which has the following form:

```
_message-instance1_.precedesOnQueue (_message-instance2_)
```

The expression is equivalent to/is a shorthand for:

```
self.outMessages->position (_message-instance1_) <
  self.outMessages->position (_message-instance2_)
```

Note that the position operator results in the index position of the item that is passed as parameter. In this context, it will result in the position of the message instance in the system's output queue. Also, it is important to note that by placing ordering constraints in the Post clause, one is able to define conditional ordering. For example, in one situation (of operation execution), it may be necessary to assert that two messages are sent in one order, but in a different situation, it may be necessary to assert that two messages are sent in the other order.

Shorthand: succeedsOnQueue

For reasons of symmetry (and convenience), I also propose a shorthand for asserting that one message instance succeeds another message instance on the system's output queue. It has the following form:

```
_message-instance1_.succeedsOnQueue (_message-instance2_)
```

The expression is equivalent to/is a shorthand for:

```
self.outMessages.position (_message-instance1_) >
  self.outMessages.position (_message-instance2_)
```

Example

Now, applying these shorthands to an example based on LotteryTicketManager, one could define the assertions for the message sending of the informWinners operation in the following way:

```
Messages: Client::{WonLottery;};
          Manager::{NumOfWinnersNotification; WinnerStatistics;};
New:      youWon: WonLottery;
          numWinners: NumOfWinnersNotification; stats: Set (WinnerStatistics);
Post: ...
  self.manager.sent (numWinners) & ...  -- the system only has a single manager rep object
  self.manager.sentAllToOne (stats) & ...
  stats->forall (s | s.precedesOnQueue (numWinners)) & ...
  winners.client.sentOneToAll (youWon) & ... -- winners is the expression-substitution defined in section 6.4.1
  youWon.succeedsOnQueue (numWinners);
```

The Post clause asserts in the first line that the message instance numWinner was placed in the system's output queue, and the manager rep object (there is only one according to figure 6.1) is the address of the destination actor instance. The second line asserts that the set of message instances denoted by stats were placed in the system's output queue with the manager set as destination. The third line asserts that all message instances in stats precede the message instance numWinners on the system's output queue. The fourth line asserts that message instance youWon was placed in the system's output queue and the winning clients were the destination actor instances. The fifth line asserts that the message instance youWon succeeds the message instance numWinners in the system's output queue.

6.7.4.3 Predefined Identifier: sender

The Post clause of an Operation Schema can also make use of the predefined identifier called sender. This identifier denotes the address of the actor instance who requested the operation. The sender identifier is an implicit parameter of every operation. However, it is important to note that in the case that an operation is time triggered, the sender identifier is strictly undefined, i.e., in this case it should not be used in an expression.

To illustrate the usage of the sender identifier, consider an Operation Schema for the informBestNumber operation. This operation sends backs to the sender a message that contains the result of the mostWinningAndPopularNumber_f Function (defined in section 6.6.2) applied to a given branch.

```
Operation: LotteryTicketManager::informBestNumber (branchPC: PostCode);
Messages:  Requestor::{MagicNumResult;};
New:       magicNum: MagicNumResult; -- MagicNumResult has one parameter only - num: LotNum
Aliases: targetBranch: Branch Is self.branch->any (b | b.postcode = branchPC
                                  and b.sold->notEmpty() and b.members->notEmpty());
Pre: self.branch->exists (b | b.postcode = branchPC and b.sold->notEmpty()
                          and b.members->notEmpty()); -- system has branch with mbrs and tickets sold
Post:
  magicNum.oclIsNew ((num => self.mostWinningAndPopularNumber_f (targetBranch)) &
  sender.sent (magicNum);
```

The Post clause above asserts that the message instance magicNum, with the most winning and popular number for the given branch as (actual) parameter, is sent back to the sender (whose type conforms to Requestor).

There are three possible ways to identify an actor instance in an Operation Schema: either by a parameter of the operation, by the sender identifier, or by a rep object of the system (see Chapter 5).

6.7.5 Message Kinds

Messages, by *default*, are sent asynchronously to their destination[1]. This means that an operation that generates a message does not synchronize with the destination actor, i.e., it does not wait until the destination actor has terminated processing the request. If the receiver does reply then it does so by a different message. Such a reply would cause a different operation of the original callee to be executed, assuming it is in a state to deal with the reply. These (default) kind of messages are called *Signals*. A message may also be an *Exception*. Exceptions denote an unusual outcome of the operation to the receiver, e.g., an invalid ticket number. Like for Signals, they are asynchronously sent to their destination. To visually distinguish between these two different kinds of messages in an Operation Schema, a naming convention is used. The naming convention is the following: suffix "_e" for an Exception, and no suffix for a Signal.

Illustrating the use of an Exception, I modify the Operation Schema for the informBestNumber operation, shown above, by removing the precondition. Consequently, one cannot assume that the given postcode (branchPC) is always valid. In such a situation, an exception could be sent. The updated Operation Schema is shown below.

```
Operation: LotteryTicketManager::informBestNumber (branchPC: PostCode);
Messages:  Requestor::{MagicNumResult; InvalidPostCode_e;};
New:       magicNum: MagicNumResult; -- MagicNumResult has one parameter only - num: LotNum
           invalidPC: InvalidPostCode_e; -- InvalidPostCode_e has no parameters
Aliases:   targetBranch: Branch Is self.branch->any (b | b.postcode = branchPC);
Pre: true;
Post:
if self.branch->exists (b | b.postcode = branchPC) then
   magicNum.oclIsNew ((num => self.mostWinningAndPopularNumber_f (targetBranch)) &
   sender.sent (magicNum)
else
   invalidPC.oclIsNew () &
   sender.sent (invalidPC)
endif;
```

In the Operation Schema above, the InvalidPostCode_e Exception is declared in the Messages clause, and an instance of it is declared in the New clause. The Post clause asserts that if there is a branch with the corresponding postcode, then the message instance that holds the magic number for that branch is sent back to the sender, otherwise an (parameter-less) exception is sent back to the sender instead. Note how Signals and Exceptions are treated in the same way (only the naming convention visually differentiates them).

In the next section, messages that denote synchronous communication between the system and an actor will be discussed (i.e., the calling operation waits on the callee to complete, possibly receiving a result in reply). Therefore, there are in fact three types of messages possible in an ANZAC specification: Signal (default), Exception, and Call.

1. Note that it is also possible to have synchronously sent messages (referred to as *Calls*); Calls are described in section 6.8.

6.8 Synchronous Communication, Results Returned by Operations, and Exceptions

In this section, I describe how one can specify that a system operation makes a (synchronous) call to an actor using Operation Schemas, show how one can specify results returned by calls with Operation Schemas, and describe the throwing and handling of exceptions in the context of synchronous communication.

To illustrate the difference between the asynchronous and synchronous communication approaches, consider figure 6.2, which shows a scenario for each approach—the two scenarios produce the same result. Both scenarios depict requestingActor making a request to systemX, where systemX requires information from helperActor to fulfill the request and thus to provide a result back to requestingActor.

The first approach (top) shows a call from requestingActor to systemX (1). During the execution of this operation, systemX makes a call on helperActor (1.1). Once the second call returns with the required result of the query, systemX returns the result of the request to requestingActor. The second approach (bottom) shows a signal sent from requestingActor to systemY (1). As part of the execution of the resulting operation, systemY sends a signal to helperActor (1.1) and then completes. As part of the execution of the operation triggered by 1.1, helperActor sends a signal to systemY (2), which has the result of the query as parameter, and then completes. As part of the execution of the operation triggered by 2, systemY sends a signal to requestingActor (2.1), which has the result of the request as parameter, and then completes.

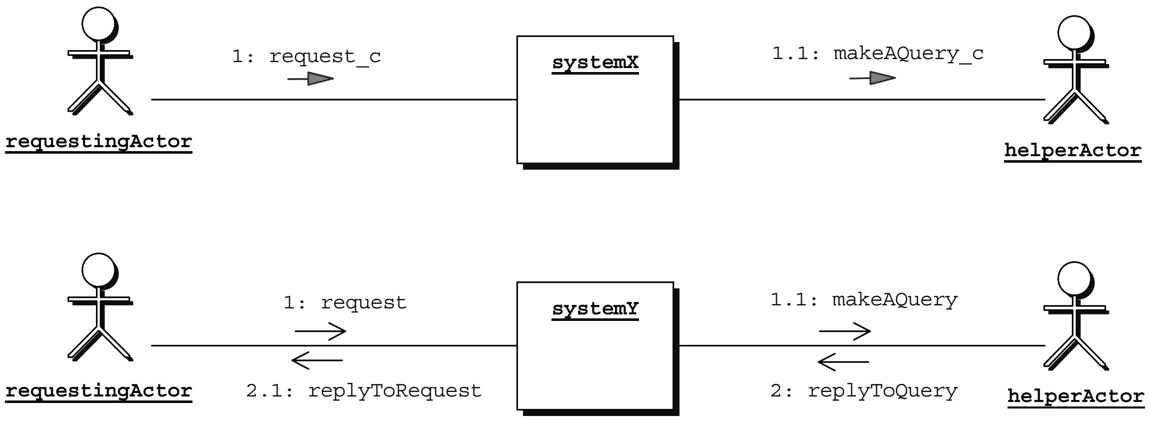

Figure 6.2: Alternatives for returning results from operations

It is important to point out that the top approach (calls) is in general only used to describe systems that are already existing, or where the system requirements mandate a synchronous protocol.

6.8.1 Calls

A *Call* denotes a message that is synchronously sent to its destination. Calls also observe a naming convention: suffix "_c". Also, all Calls that lead to operations that return results have an out parameter called result. The result parameter of a Call denotes the result that is returned to the operation with the completion of the call to the callee. Furthermore, Calls that can throw exceptions have an association with each Exception that may be thrown. For example, figure 6.3 shows a Call called CallX_c that refers to an operation call that may throw two kinds of exceptions: ExceptionA_e and ExceptionB_e. CallX_c has a 0..1 association with both these exceptions. The rolename is shown explicitly in the example, but it should always be the name of the Exception with a lower case first letter, i.e., the default rolename. Each rolename can be used to access the exception that is thrown by the corresponding call. Therefore, if an exception is found at the other end, then this exception was thrown as a result of the call.

All Calls that can throw exceptions have a special boolean property called noExceptions. The noExceptions property is used to indicate whether the call returned with or without exception, i.e., noExceptions is true if the call did not raise any exceptions, false otherwise. It is assumed that noExceptions will always evaluate to true or false in the post-state of the calling operation. This means that calls are always processed by the called actor and they always return. Therefore, it is assumed that actors are always able to service the call, i.e., actors are never "down" when called[1], and according to Rule 6, communication between the system and its actors is reliable.

Note that in a Post clause attaching the @pre decorator to result or noExceptions in an expression is illegal. Also, it is undefined to access the result of a call where noExceptions is true.

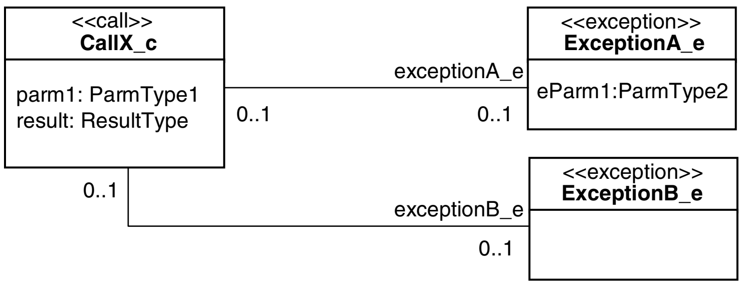

Figure 6.3: Call with two exceptions

A Call is specified in an Operation Schema in the same way as the other kinds of messages. The only constraint in addition is that a Call can only have a single destination, unlike Signals and Exceptions that can have one or more destinations. In other words, the destination

1. Note that designers of distributed systems would need to expand on this model, since this assumption is not usually realistic at the design/implementation level.

property of Call can never be greater than one; and therefore, the use of the shorthands sentOneToAll and sentAllToAll are not allowed.

Illustrating the use of a Call in an Operation Schema, I provide the following example:

Messages: ActorX::{CallMessage1_c;};
New: msg1: CallMessage1_c;
Post:
  msg1.oclIsNew () &
  actorX.sent (msg1) ...

The above Post clause fragment asserts that the Call instance msg1 was placed in the system's output queue, given that actorX is of type ActorX. Note that the approach is the same as the one for Signals, except for the difference in naming conventions.

Let us say that the Call CallMessage1_c returns a result on completion. It therefore has a result parameter. Since parameters are treated in much the same way as attributes on objects, it is possible to navigate to this returned result. For example, the following Post clause completes the one shown above:

Post:
  msg1.oclIsNew () &
  actorX.*sent* (msg1) &
  self.aPlaceToStoreResult = msg1.result;

The above Post clause assumes that the type of result conforms to the type of the expression self.aPlaceToStoreResult. Also, it is important to note that msg1.result refers to the value of the result at post-operation time, which is always defined because the operation cannot terminate before receiving the reply.

Just as it is possible to define an Operation Schema for an operation that makes a (synchronous) call, it is equally possible to define an Operation Schema for an operation that is (synchronously) called. In this case, the Operation clause of the Operation Schema must define the result type in its signature, and the Post clause must define what value the result takes with the execution of the operation. The predefined identifier result denotes the result of the operation. For example, the informBestNumber operation, shown in section 6.7.4, is changed into an operation that returns a result (rather than sending a message in reply); it is shown below.

Operation: LotteryTicketManager::informBestNumber_c (branchPC: PostCode): LotNum;
Aliases: targetBranch: Branch **Is** self.branch->any (b | b.postcode = branchPC);
Pre: self.branch->exists (b | b.postcode = branchPC);
Post:
  result = self.mostWinningAndPopularNumber_f (targetBranch);

As with the earlier version, the operation takes a postcode as parameter, but this time the operation returns a LotNum as result.

Staying with this same example, one could imagine an operation of a *different* system that calls the informBestNumber operation. Below, I show an Operation Schema for an operation of the ManagerBot system, which makes possibly many calls to the informBestNumber operation of the LotteryTicketManager system.

```
Operation: ManagerBot::sumBestNumbers (postcodes: Set (PostCode));
Messages: TicketingManager::{InformBestNumber_c;};
New: informNums: Set (InformBestNumber_c); -- set of message instances
Post:
postcodes->forall (pc |          -- for every postcode
  informNums->one (iMsg |        -- there is one msg instance, iMsg, which is a member of the set, informNums
    iMsg.oclIsNew ((branchPC => pc)) & -- that was generated
    self.ticketingManager.sent (iMsg)    -- and placed in output queue, with ticket manager as destination
  )
) &
self.currentSum = informNums->sum (result); -- the currentSum attribute was set to the sum of the results
```

In the above Operation Schema, the first expression (forall) in the Post clause asserts that as many calls as postcodes were made on the ticketing manager, each call having a different postcode as parameter. The second expression asserts that the currentSum attribute of the system (ManagerBot) was set to the sum of all the results. Note that it is assumed that the ManagerBot system has an attribute called currentSum (which has a numeric type).

Also, with respect to the above Operation Schema, it is important to note that the name of the actor in the Messages clause is not the same as LotteryTicketManager. In this example, it is assumed that LotteryTicketManager can play the role TicketingManager. This point highlights the fact that actors are roles, and from an external point of view, the concrete type need not be "seen", only the roles that the entity (i.e., the system) plays, e.g., TicketingManager.

6.8.2 Exceptions From Synchronous Calls

Despite our assumption of reliable communications, there are often situations where the requested system cannot provide what was requested for. In such situations, it is often necessary to throw an exception. Thus, it should be possible to specify the throwing and catching of exceptions.

When Signals are sent between the system and its actors, exception handlers are just operations of the client, i.e., the called operation will inform the caller by sending back the exception to the caller, which will be handled by a different operation of the caller. However, as I have already mentioned in this section, (synchronous) calls can also throw exceptions. In this case, the exceptions must be dealt with by the operation that makes the call.

All operations making synchronous calls to operations that may throw exceptions must declare the exceptions in their respective Operation Schema, i.e., the calling operation must declare the exceptions. More precisely, in the Messages clause, the **Throws** keyword is used to declare all exceptions that correspond to a particular Call. Revisiting the syntax of the Messages clause:

```
ActorWithMessages ::=
   ActorClassName "::" "{" ( SignalName ";"
                              | CallName [ "Throws" ExceptionMessages ] ";")* "}"
```

where ExceptionMessages are always declared immediately after the respective CallName (separated by the **Throws** keyword).

For instance, extending the example of the informBestNumber_c operation to throw two exceptions, say invalidPostCode_e and NoNumbersForBranch_e, then the Messages clause of the Operation Schema of an operation that calls InformBestNumberv2_c could be defined in the following way:

```
Messages: TicketingManager::
               {InformBestNumberv2_c Throws InvalidPostCode_e, NoNumbersForBranch_e;};
```

The Messages clause declares that the informBestNumberv2_c operation may throw the exceptions, InvalidPostCode_e and NoNumbersForBranch_e, if it is called.

An Operation Schema for an operation that can raise an exception must state under which circumstances it throws the exception and which type of exception is thrown. To illustrate an Operation Schema that describes an operation that can throw an exception, consider the following Operation Schema, which describes the informBestNumberv2_c operation.

```
Operation: LotteryTicketManager::informBestNumberv2_c (branchPC: PostCode): LotNum;
Aliases:  targetBranch: Branch Is self.branch->any (b | b.postcode = branchPC);
          noTickectsAndNoMbrs: LotNum Is targetBranch.sold->notEmpty() and
                                         targetBranch.members->notEmpty();
Messages: Manager::{InvalidPostCode_e; NoNumbersForBranch_e;};
New: ipc: InvalidPostCode_e; noNums: NoNumbersForBranch_e;
Post:
   if not self.branch->exists (b | b.postcode = branchPC) then
      ipc.oclIsNew ((invalidPC => branchPC)) &
      exception = ipc
   elsif noTickectsAndNoMbrs then
      noNums.oclIsNew () &
      exception = noNums
   else
      result = self.mostWinningAndPopularNumber_f (targetBranch)
   endif;
```

The above Post clause asserts that in the first two cases of the if-then-elsif-else expression, an exception is raised with the caller. Note that the predefined identifier exception is used in much the same way as the predefined identifier result.

As stated earlier, all Calls that trigger operations that can throw exceptions have associations with the corresponding Exceptions (see figure 6.3). This means that within the Post clause of a calling operation, one can check to see if any exception is thrown by navigating the appropriate association link. For example, one could write the following Post clause extract (and New clause):

```
New: bestNumMessage: InformBestNumberv2_c;
```

```
Post: ...
   actorX.sent (bestNumMessage) &
   if bestNumMessage.invalidPostCode_e->notEmpty () then
      ...
```

The if-condition in the Post clause extract evaluates to true if the operation referred to by bestNumMessage threw an InvalidPostCode_e Exception.

I extend the Operation Schema for the sumBestNumbers operation to take into account the possible throwing of exceptions by a InformBestNumber_c call. It is shown below (note that it makes use of a Parameterized Predicate, which is shown afterwards):

```
Operation: ManagerBot::sumBestNumbers_V2 (postcodes: Set (PostCode));
Messages: TicketingManager::
               {InformBestNumberv2_c Throws InvalidPostCode_e, NoNumbersForBranch_e;};
New: informNums: Set (InformBestNumberv2_c); -- set of message instances
Post:
postcodes->forall (pc |            -- for every postcode
   informNums->one (iMsg |         -- there is one msg instance, iMsg, which is a member of the set, informNums
      iMsg.oclIsNew ((branchPC => pc)) & -- message instance (a member of informNums) was generated
      self.ticketingManager.sent (iMsg) -- and placed in output queue, with ticket manager as destination
   )
) &
informNums->forall (i | addResult_p (i)); -- Parameterized Predicate is true for each message sent
```

The first expression (forall) in the above Post clause asserts that as many calls as postcodes were made on the ticketing manager, each call having a different postcode as parameter. The second forall expression asserts that the addResult_p Parameterized Predicate holds for all messages that were output.

To reduce the size of the Post clause shown above, I encapsulated into a Parameterized Predicate all of the expressions that relate to the treatment of the results. The addResult_p Parameterized Predicate is shown below:

```
Predicate: LotteryTicketManager::addResult_p (iMsg: InformBestNumber_c);
Body:
   if iMsg.noExceptions then
      self.currentSum += iMsg.result          -- result is added to sum
   elsif iMsg.invalidPostCode_e->notEmpty () then
      self.invalidCode->includes (iMsg.invalidPostCode_e.invalidPC)
      --invalidPC (parm. of exception) is added to the invalid code attribute of system (collection of post codes)
   elsif iMsg.noNumbersForBranch_e->notEmpty () then
      self.failure += 1                       -- the number of "no number for branch" failures are recorded
   endif;
```

The above Parameterized Predicate defines the three possible outcomes of the call to the operation implied by InformBestNumber_c. Either the call returned a result, in which case the result is added to the currentSum attribute of the system. Otherwise, the call threw an InvalidPostCode_e Exception, in which case the invalid postcode is added to the invalidCode attribute of the system (assumed to be a collection of PostCode). Otherwise, the call threw a

NoNumbersForBranch_e Exception, in which case the failure attribute of the system is incremented (failure is assumed to be a numeric type).

6.9 Concurrent Operation Execution

A reactive system often must interact with an inherently concurrent environment, i.e., it must simultaneously deal with message instances from different sources (actor instances). Add to inherent concurrency the fact that operations always have some duration, and the result is a system that has operations whose executions may overlap in time.

In terms of modeling such systems it is always possible to ignore the issues that surround concurrent execution of operations (e.g., assume system operations are instantaneous—à la Fusion), but such a model, although lucid, fails to give any assistance to developers for the difficult task of designing concurrent systems. Thus, in developing an ANZAC specification, any information in the Operation Model that provides insight into the inherently concurrent nature of operations, but still avoids premature design decisions, would assist designers in making informed choices on concurrency control mechanisms. In an Operation Model, it would be interesting to provide designers with information about resources shared by concurrent operations and information about any inherent synchronization requirements between concurrent operations.

I now analyze the capability of Operation Schemas for specifying concurrent operations, and I propose some extensions to them for the purposes of communicating to the designers the essential information about the concurrent nature of system operations. Note that the ordering of operations, and in particular, the description of which operations can execute concurrently to other operations is given by the Protocol Model. The Protocol Model, which is the third model of an ANZAC specification, is presented in the next chapter. In the discussion given in this section, it is therefore assumed that one is able tell which operations can execute concurrently to others (i.e., because this information is given by the Protocol Model).

6.9.1 Problems in Specifying Concurrent Operations with Operation Schemas

It has been assumed in previous sections that Operation Schemas describe operations that execute atomically and instantaneously, i.e., at most one operation is ever executing at a given moment. As a consequence, the postcondition of an Operation Schema is described as if the corresponding operation is the only one that can change the state of the system during the operation's execution period. However, in a concurrent system, this assumption fails because the state (population) of the system may be shared among many different operations at any one time, where operations access and update common parts of the system's state over their periods of execution.

Recall from section 6.5.3 that the frame of an Operation Schema contains all those objects and association links are changed by the operation, forming a subset of all objects and association links that populate the system. If operations that can execute in parallel have overlapping frames, i.e., frames that have a non-empty intersection, then the system state that results from the execution of the operations may not match the one asserted by the Operation Schemas. In fact, due to the many possible combinations of instruction interleaving, it is often difficult to predict exactly how concurrent operations will combine to form the new system state. (It is non-deterministic in fact.)

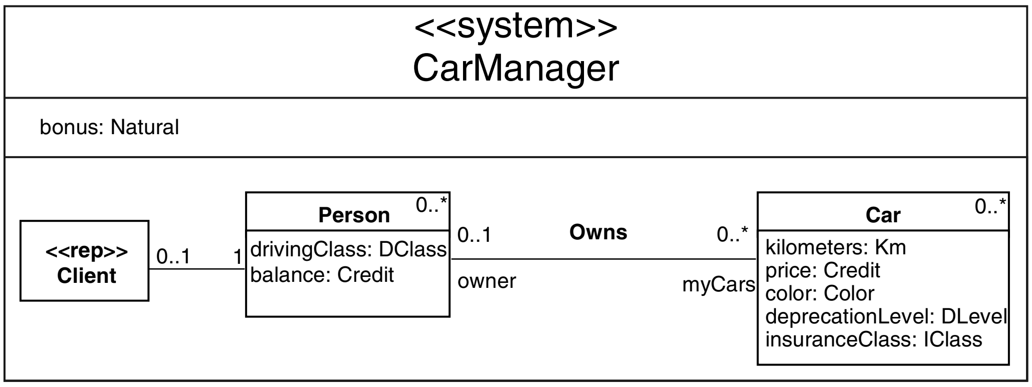

Figure 6.4: CarManager Concept Model

Even operations that are commutative are not always correctly described by Operation Schemas. Given the CarManager system shown in figure 6.4, consider the two Operation Schemas that describe operations for making a person the owner of a blue and red car respectively, shown in figure 6.5. If these two operations are allowed to execute in parallel then their postconditions would not hold true in all situations. For instance, consider the case when josh: Person has a blueSuzuki car added with the addBlueCar operation, and a redFerrari car added with the addRedCar operation in parallel, where they start execution at the same time. If they start execution when josh owns no cars, then the Owns association after the completion of the first operation to terminate could be in one of three states: Owns with the link (josh, redFerrari) and the link (josh, blueSuzuki); Owns with only the link (josh, redFerrari); or Owns with only the link (josh, blueSuzuki). Thus, the Operation Schema of the first operation to terminate will incorrectly define the resulting system state if the first case is true, and the second operation to terminate is guaranteed to incorrectly define the resulting system state, since it will state that only one car was added when in fact two were added.

Operation: CarManager::addBlueCar (
p: Person, c:Car);
Scope: Person; Car; Owns;
Pre: c.color = Color::blue;
Post: p.myCars = p.myCars@pre->including(c);

Operation: CarManager::addRedCar (
p: Person, c:Car);
Scope: Person; Car; Owns;
Pre: c.color = Color::red;
Post: p.myCars = p.myCars@pre->including(c);

Figure 6.5: Operation Schemas for adding cars in the CarManager system

To make matters worse, there is a darker problem when operations can execute concurrently. This problem relates to non-atomic updates of shared resources: interference between operations may cause resource corruption, which is almost guaranteed to make the system behave in an erroneous fashion.

6.9.2 Operation Schemas for Specifying Concurrent Operations

Revisiting the notions of safety and liveness properties that were introduced in Chapter 2, it is important to state what constraints need to be placed on the design/implementation of a concurrent operation in addition to those imposed on sequential operations. Note that this means the liveness property of operation termination and the safety property of correct operation result, which were part of the contract for sequential operations, are retained for the contract on concurrent operations.

In terms of safety properties, atomic updates and absence of deadlocks are candidates for inclusion in the contract. As highlighted in the previous section, it is important that resources are updated atomically, so that resources do not become corrupted. At first glance, ensuring that there are no deadlocks across the Operation Model may not be necessary, because the sequential contract ensures operation termination. However, for concurrent operations, termination does not necessarily guarantee that its enclosing activity is not in deadlock. In the worst case, an operation may (continually) terminate but its activity may be in livelock (a busy-waiting form of deadlock), where it repeatedly executes the operation in vain, because another activity has a "death-hold" on resources it needs.

In terms of liveness properties, it is important that a scheduler is at least unconditionally fair, because any assertions about updates to shared resources should conform to the requirement for (eventual) operation termination. In other words, with an unconditionally fair scheduler, one can assume in writing an Operation Schema that a resource can be eventually accessed. In considering the other two types of scheduling policy, it may be too restrictive to impose a weakly fair scheduler and it would definitely be too restrictive to impose a strongly fair scheduler.

Bringing the safety and liveness properties together, the list includes: atomic updates, absence of deadlocks and an unconditionally fair scheduler. However, rather than impose atomic updates as part of the contract on concurrent operations, I instead propose a calculus for denoting atomic updates in Operation Schemas. Thus, atomic updates are specifier-enforced. In this way, the synchronization constraints are explicit so that designers do not “miss” them.

Taking into account the above discussion, the correctness rule for Operation Schemas is updated by revising the assumptions that were defined by definition 6.1.

Definition 6.5:Revised Assumption of 6.1 – the precondition and all system invariants are satisfied immediately before the operation is executed and the scheduling policy is (at least) unconditionally fair.

One overlying constraint on the system, which is independent of the contract on an operation (and can be seen as a special system invariant), is the safety property: absence of deadlocks.

6.9.2.1 Granularity of Serialization

One goal with describing concurrent operations using Operation Schemas is to impose as few constraints on the required granularity of serialization as possible. This is because an Operation Schema acts as a contract on the operation, where the implemented operation must be shown to at least fulfill the contract specified by it. Therefore any constraint on the granularity of concurrency defined by an Operation Schema must at least be satisfied by the implementation (according to the rule of refinement). By describing a fine-grained notion of concurrency, the designer is then given the choice to take a more pessimistic concurrency policy in the implementation. However, strictly speaking the inverse, i.e., a more optimistic implementation than the contract, is not possible (in the sense of refinement).

6.9.2.2 Shared Clause

Ensuring that operations update resources atomically means that resources can not become corrupted by interference between operations. To enforce this constraint on shared resources in Operation Schemas, we add a clause to the Operation Schema format called Shared. This clause is placed directly above the Messages clause in an Operation Schema.

Resources listed in this clause are either classes or associations. In the case that a class is contained in the Shared clause, the meaning is that the attributes of one or more objects of the class may be shared. It is possible to have one attribute of an object that is shared and another one that is not. Thus, sharing is defined at the attribute-level. In the case that an association is contained in the Shared clause, the meaning is that the set of all links of the association is shared, i.e., the set of tuples. The syntax for the clause is the following:

“**Shared**” “:” (SharedItem “;”)*
SharedItem ::= AssociationName | ClassName

As a rule, all shared resources are updated atomically by the operation. However, it is important to note that the Shared clause does not state exactly which object attributes are shared. The exact information is provided by the Post clause, which is discussed in the next subsection.

To illustrate a Shared clause, consider the CarManager system of figure 6.4. It would be possible to write the following Shared clause for an Operation Schema (assuming they are shared resources used by the respective operation).

```
Shared:
Owns; Person;
```

The Shared-Clause Algorithm

The associations and classes that are shared by different operations can be calculated by the following informal algorithm, which is described using pseudo-OCL. This algorithm, which I refer to as the shared-clause algorithm, can be used to calculate the Shared clause of an Operation Schema, given as input: the Protocol Model for the system and the Scope clauses of every Operation Schema of the system.

```
By analyzing the Protocol Model,
let grpsOfgrpsOfOps equal the group of all groups of operations that may execute in parallel:
grpsOfgrpsOfOps->forall (grpsOfOps | -- for each group of concurrent operations:
   grpsOfOps->forall (op | -- for each operation op:
      op.shared->includesAll( op.scope->intersection ((grpsOfOps - op)->collect(scope)) )
      -- the Shared clause of op includes all associations and classes that result from the intersection of the Scope clause
      -- of operation op with the Scope clauses of all other operations in grpsOfOps
   )
)
```

The meaning of the algorithm is the following: all operations that may execute in parallel (according to the Protocol Model) are placed into common groups (note that an operation may be a member of many groups); for each operation in a group, all classes and associations in the Scope clause of its Operation Schema is intersected with the union of the Scope clauses of all other operations in the same group. The result of this intersection forms the Shared clause for the Operation Schema of the respective operation.

6.9.2.3 Shared Resources

In a concurrent system, the state of the system may be changed by many operations at any one time, when operations access and update common parts of the system's state. As a consequence, it is problematic to define the changes made to shared resources in a Post clause by asserting its post-operation value in terms of its pre-operation value. This is because competing operations may have also changed the value of the resource during the execution period of the operation, invalidating any pre-post assertion made on the resource by the Operation Schema.

Rather than making an assertion about the change to the value of a shared resource over the whole period of the operation, we can only reason about the change that the operation makes over the time that it updates the resource—taking advantage of the atomic update constraint. Moreover, we can describe the update by making an assertion about the change to the resource immediately before and after the atomic update. For example, if an operation called incrementSpeedometer of the CarManager system increments the kilometers attribute of a given object c of class Car, the effect that one wishes to ensure is that 1 was added to the attribute as a result of the atomic update. To denote the value of the resource immediately before and after the atomic update by the operation, the decorators @preAU and @postAU are used, respectively, where AU stands for Atomic Update. These decorators are postfixed to an expression that denotes a shared resource. And a @preAU or @postAU decorator denotes the *snapshot* of the value of the respective resource at a particular moment during the execution period of the operation. The incrementSpeedometer operation could be described by the following Operation Schema:

Operation: CarManager::incrementSpeedometer (c: Car);
Shared: Car;
Post: c.kilometers@postAU = c.kilometers@preAU + 1;

The above Post clause asserts that the post-atomic-update value of car c's kilometers attribute is equal to its pre-atomic-update value plus one. From the observational point of view, one could interpret the Post clause in the following way: "after the execution of the operation, observers of the operation reported that during its execution, the operation atomically updated c's kilometers attribute by one".

Taking a fine-grained view of updates means that the Post clause of an Operation Schema for a concurrent operation redefines a postcondition as an assertion that refers to different moments during the execution period of the operation, rather than just making reference to the before and after states of the operation; this is expressed with the following rule.

Rule 7: An Operation Schema that describes an operation that can execute concurrently with other operations defines a postcondition that may refer to different moments during the execution period of the operation, but it must be completely satisfied on termination of the operation (given that the precondition initially held).

Applying this rule, one would interpret the contract asserted by the Operation Schema for incrementSpeedometer in the following way: the incrementSpeedometer operation must terminate in a state such that during its execution, it made an atomic update to the kilometers attribute of the car object parameter c, incrementing its value by one.

Reading Shared Resources

A shared resource may be updated at any time during the execution period of the operation, thus the value of a shared resource that is read by an operation may be different to the value it would have at precondition and postcondition time, or any other time for that matter. To correctly describe the effect of an operation, it is important to be able to denote the value of a shared resource that is read by the operation. To denote the snapshot of the value at the time of reading the resource, the suffix @rd is used. Note that the decorators @preAU and

@postAU also imply reads of the corresponding resources, however @rd does not imply any relationship to an update, whereas @preAU and @postAU do.

To illustrate the use of the @rd decorator, consider an operation of the CarManager system (figure 6.4) that results in the value of the kilometers attribute of a given object c of class Car. The Operation Schema for this operation is shown below:

Operation: CarManager::informKilometers (c: Car): Km;
Shared: Car;
Post: result = c.kilometers@rd;

The Post clause asserts that the operation results in the value read for the attribute kilometers of object c. There are several points to make about the @rd decorator:

- The snapshot implied by the @rd can be taken any time during the operation, i.e., there is no constraint on the time at which it is read. With respect to the example, this means that the resource denoted by the expression c.kilometers could be read at any stage during the operation period.
- A snapshot values implied by @rd are consistent. Thus, even if the resource is being updated at the time of the read, the result of the read will be the last consistent value of the resource. This fact is different to the interpretation of the snapshots implied by both @preAU and @postAU, because they are constrained to be taken at one particular instant in time.

Predicate Nature of the Post Clause

It is important to remember that the Post clause is just a predicate. For example, an operation that implements the Operation Schemas shown below would not have any effect on the state of the system if the boss' cars were already red.

Operation: CarManager::makeCarsOfBossRed (boss: Person);
Shared: Car;
Post: boss.myCars->forall (c | c.color@postAU = Color::red);

The point to make is that the decorator @postAU does not imply that an update was necessarily made, only that a snapshot was made at a time that would correspond to the time that the atomic update terminated.

6.9.2.4 Usage of Decorators

The decorators defined in the previous section can also be used in the Aliases clause of an Operation Schema. That is, expression-substitutions can make use of the three suffixes @preAU, @postAU, and @rd. For example, with respect to the CarManager system (figure 6.4), one could define the following Aliases clause (taking into account the Shared clause above it):

Shared:
    Person; Car;

```
Aliases:
   premierClassDriver: Person Is self.person->any (p | p.drivingClass@rd = DClass::premier);
   carsToUpdate: Set (Car) Is self.car->select (c | c.kilometers@preAU > 10000);
```

Note that expression-substitutions that use one of the three decorators (i.e., @rd, @preAU, @postAU) should not be used in a Pre clause; in fact, all expressions that make use of the @rd, @preAU and @postAU decorators can only be mentioned in the Aliases and Post clauses.

6.9.2.5 Multiple Snapshots of the Value of a Resource

In general, if there are multiple expressions denoting the same resource and using the @rd decorator in an Operation Schema, then each expression refers to *possibly* different snapshots, and thus they may denote different values. For example, consider an operation of the CarManager system that multiplies the price, kilometers, and deprecationLevel of a given car by the system's bonus amount:

```
Operation: CarManager::multiplyByBonus (c: Car);
Shared: CarManager;
Post:
   c.price = c.price@pre * self.bonus@rd &
   c.kilometers = c.kilometers@pre * self.bonus@rd &
   c.deprecationLevel = c.deprecationLevel@pre * self.bonus@rd;
```

The above Post clause does *not* constrain each of the three self.bonus@rd snapshots to be taken at the same time; thus, they may refer to a different value of the resource denoted by the expression.

In the case of the @preAU decorator, if there is at most one update to the corresponding resource implied by an Operation Schema, then multiple expressions denoting the same resource and using the @preAU decorator in an Operation Schema will denote the same value; similarly for the @postAU decorator. However, in the case that multiple updates are implied, the snapshots of the expressions may denote different values, i.e., they may be taken at different times. Thus, in such cases, it is impossible to know which snapshots of a resource relate to each other. For example, it is not clear in the following Post clause what the relationships between the snapshots taken on the left-hand-side and the right-hand-side of the two lines are.

```
Post:
   self.bonus@postAU = self.bonus@preAU + 1 &
   self.bonus@postAU = self.bonus@preAU + 2;
```

More precisely, there is no constraint that states that self.bonus@postAU and self.bonus@preAU on line 1 should refer to snapshots of the same atomic update, and similarly for line 2. This is because the Post clause implies that there could be possibly two updates.

This inability to determine which snapshots refer to which updates leads to non-determinism. In the case of the above Post clause, three interpretations would be possible. Either there is a single atomic update performed to the resource denoted by self.bonus, which would mean that the Post clause is a contradiction, or there are two atomic update performed to the resource denoted by self.bonus, in which case the equality between the preAU and postAU pairs on each line could refer to different atomic updates or to the same ones.

There are two possible ways to address this issue: either one enforces that a shared resource has at most one snapshot corresponding to each of the three suffixes @preAU, @postAU and @rd in an Operation Schema, or one introduces a marking scheme that is applied to each matching decorator pairs of the Operation Schema.

In terms of abstraction, the second option is far more inviting, because it would mean that a resource is not constrained to a single atomic update in an operation—such a constraint would clearly restrict the designer unnecessarily. However, the downside with the second option is that additional language constructs would need to be introduced and enforced to avoid "non-deterministic contradictions".

The second option is taken in this work. A marking scheme is introduced by the proposition of the *related-snapshots* convention. The related-snapshots convention involves adding a number (superscript) to the end of the corresponding decorators to clarify which snapshots are related. For example, the Post clause shown earlier could be rewritten in the following way, making use of the related-snapshots convention:

Post:
  self.bonus@postAU^1 = self.bonus@preAU^1 + 1 &
  self.bonus@postAU^2 = self.bonus@preAU^2 + 2;

This Post clause would now have one interpretation: there are two atomic updates to the resource denoted by self.bonus and each line defines an assertion on each update. Note that I use superscript font for the number; however, in the case that standard text is being used then clearly the number must be represented in non-superscript form, e.g., self.bonus@postAU1.

It is important to note that the related-snapshots convention is *only* used in the case that there are two or more updates implied by the Post clause. For example, the following Post clause does not require the related-snapshots convention to be applied, even though there are two matching expressions, i.e., self.bonus@preAU.

Post:
  self.bonus@preAU > 2 **implies** self.bonus@postAU = self.bonus@preAU + 1;

In other words, the numbering scheme does not need to be applied to the self.bonus@preAU expression on the left-hand-side of the implies expression and the self.bonus@preAU expression on the right-hand-side of the = expression, because there is no possible confusion over which updates the expressions refer to.

Motivation for Incremental Descriptions

One unfortunate consequence of allowing multiple snapshots of a single resource is that the numbering scheme of the related-snapshots convention needs to be applied to all expressions with @preAU and @postAU decorators that are used in multiple places, because contradictions are possible otherwise. However, looking at the structure of such contradictions, it is possible to relate back to an approach that was discussed earlier, i.e., incremental descriptions. If an incremental style is used, then there would be no possibility for contradiction. For example, rewriting the Post clause shown earlier with an incremental style would result in the following Post clause:

Post:
  self.bonus@postAU += 1 &
  self.bonus@postAU += 2;

In this case, the numbering scheme is not needed, because there is no possibility for contradiction. In fact, this Post clause is less restrictive than the earlier one because it allows for one or two updates to be made without bias toward either choice. This aspect is discussed further in section 6.9.2.6, where incremental descriptions for expressions with shared resources are described.

Synchronizing Multiple Reads

In the case of multiple reads in an Operation Schema, it is possible to show which expressions denote the same snapshots using the numbering scheme of the related-snapshots convention. In other words, if it is necessary (i.e., a requirement) to constrain the value of a read that is shown in multiple places in an Operation Schema to be the same one, then the numbering scheme can be used to show that the expressions denote the same snapshot. For example, if it was necessary to constrain the multiplyByBonus operation to "use" the same value of the system's bonus attribute (i.e., self.bonus) for all three multiplications, then it could be described by an Operation Schema in the following way:

Operation: CarManager::multiplyByBonus_v2 (c: Car);
Shared: CarManager;
Post:
  c.price = c.price@pre * self.bonus@rd^1 &
  c.kilometers = c.kilometers@pre * self.bonus@rd^1 &
  c.deprecationLevel = c.deprecationLevel@pre * self.bonus@rd^1;

The Post clause asserts that the three reads of the resource denoted by self.bonus refer to the same snapshot, and therefore they all have the same value.

Synchronizing Reads With Updates

Consider the following Operation Schema. It is an operation of the CarManager system that fixes the price of a car to that of the balance of the person divided by 2, where the balance and the price are both shared resources.

Operation: CarManager::makePriceForPerson (c: Car, p: Person);

Shared: Car; Person;
Post:
c.price@postAU = p.balance@rd div 2;

According to the interpretation of the @rd decorator, the snapshot denoted by p.balance@rd could be taken at any time during the operation, i.e., it is not constrained to be related to the period of the update to the resource denoted by c.price. However, in many situations, one would like to constrain the read to be performed within the period of the update. If such a requirement exists then, it is possible to use the related-snapshots convention to indicate that a read is synchronized with an update. With respect to the example, it would be written in the following way:

Operation: CarManager::makePriceForPerson (c: Car, p: Person);
Shared: Car; Person;
Post:
c.price@postAU^1 = p.balance@rd^1 div 2;

The above Post clause asserts that the price of the car c immediately after it was atomically updated by the operation had half the value of the balance of the person p that was read during the atomic update period.

6.9.2.6 Incremental Descriptions

Incremental descriptions are also possible in the “concurrent” version of Operation Schemas. A nice characteristic of incremental descriptions is that they allow one to avoid the issues that surround “non-deterministic contradictions” (discussed in section 6.9.2.5), which was one reason for introducing the related-snapshots convention. To this end, incremental descriptions do not prescribe whether multiple expressions that denote updates in a Post clause for a single (shared) resource are to be made as part of the same or different updates. Therefore, I strongly recommended that an incremental style of description be used for describing concurrent operations with Operation Schemas.

Minimum Set Principle

The minimum set principle can be applied to shared collections. It defines the difference between the collection before and after the atomic update performed by the operation. For example, taking the CarManager system (figure 6.4), one could imagine an operation that makes a given person p the owner of all unowned cars in the system.

Operation: CarManager::giveAwayAvailableCars (p: Person);
Shared: Owns; Person;
Aliases: availCars: Set (Car) **Is** self.car->select (c | c.owner@preAU->isEmpty ());
Post:
p.myCars@postAU->includesAll (availCars);

The Post clause highlights the use of an expression-substitution, availCars, which denotes the set of all unowned cars at the time immediately before the update to the set p.myCars.

Applying the minimum set principle to the Post clause, we see that the set p.myCars after the atomic update is equal to the minimum change to it between the snapshots: @preAU and @postAU. Note that p.myCars represents a subset of the tuples of the association Owns—the Owns association can be seen as a set of tuples of (Person and Car).

Incremental Plus and Minus

As for the minimum set principle, incremental plus and minus is reinterpreted for use with shared resources. This means incremental plus and minus can be used to define the incremental change to shared resources that have numeric types.

For example, the Operation Schema for incrementSpeedometer operation (shown earlier) can be rewritten in the following way:

```
Operation: CarManager::incrementSpeedometer (c: Car);
Shared: Car;
Post: c.kilometers@postAU += 1;
```

The Post clause of this Operation Schema and the one shown earlier for the same operation both make the same assertion on the operation.

Illustrating the incremental style of incremental plus and minus, consider the following Post clause:

```
Post:
    c.kilometers@postAU += 1 &
    c.kilometers@postAU += 10;
```

Note that the above Post clause can be interpreted in two ways (using the non-incremental style to illustrate the interpretation). It is either equivalent to the following Post clause,

```
Post: c.kilometers@postAU = c.kilometers@preAU + 11;
```

Or it is equivalent to the Post clause shown below:

Post:
c.kilometers@postAU^1 = c.kilometers@preAU^1 + 1 &
c.kilometers@postAU^2 = c.kilometers@preAU^2 + 10;

6.9.2.7 If-Then-Else Expressions

The additions that have been proposed so far (a Shared clause and the new decorators @preAU, @postAU, @rd) are aligned with the predicate nature of Operation Schemas. Equally, if-then-else expressions can be used to structure Operation Schemas that describe concurrent operations.

Illustrating the use of an if-then-else expression, consider an operation of the CarManager system, shown below, that resets the speedometer of a given car only if it has less than 100 kilometers, otherwise it throws an exception to the calling actor.

```
Operation: CarManager::resetSpeedometer (c: Car);
Shared: Car;
Messages: Client::{CarWarning_e;};
```

```
New:        stopTampering: CarWarning_e;
Post:
   if c.kilometers@preAU < 100 then
      c.kilometers@postAU = 0
   else
      stopTampering.oclIsNew ((reason => Reason::tamperingWithSpeedo, car => c)) &
      sender.sent (stopTampering)
   endif;
```

The Post clause asserts that the car's kilometer attribute is set to zero if it had a value of less than 100 immediately before it was updated, otherwise a warning message was sent to the caller of this operation (sender) instead. Note that in a system where operations can execute concurrently, the output message queue of the system can also be a shared resource. Therefore, I propose that the output message queue is implicitly constrained to be atomically updated in all Operation Schemas that describe concurrent operations. As a consequence, the sent shorthand can be used in the usual way, as shown above. I discuss further the use of shared resources and shorthands in section 6.9.2.10.

Care should be taken not to confuse if-then-else expressions with if-then-else statements like those found in imperative programming languages. To emphasize that an if-then-else expression is simply a predicate, below, I show the above Post clause rewritten using implies expressions:

```
Post:
   c.kilometers@preAU < 100 implies c.kilometers@postAU = 0 &
   not (c.kilometers@preAU < 100) implies stopTampering.oclIsNew...;
```

Explaining the above Post clause in more detail: an atomic update on the kilometers attribute of car c is only performed in the case that the snapshot of the resource denoted by the @preAU decorator has a value of less than 100. In the case that the snapshot of the resource has a value of 100 or more, a warning message was sent to the sender. Note that the @preAU decorator denotes a snapshot of the value of the respective resource at the moment that would correspond to immediately before the update of the resource, if and only if the update is judged necessary. Also, note that the related-snapshots convention is not needed, because at most one update is implied by the Post clause to the resource denoted by the expression c.kilometers.

If-Then-Elsif-Else Expressions

It is also possible to use if-then-elsif-else expressions for describing multi-branch expressions. For example, the CarManager system could have an operation that allocates credit to a particular person, where the amount given depends on the balance of the person. An Operation Schema for the giveCreditPayout operation is shown below.

```
Operation: CarManager::giveCreditPayout (p: Person);
Shared: Person;
Post:
```

```
if p.balance@preAU > 500 then
   p.balance@postAU += 50
elsif p.balance@preAU > 300 then
   p.balance@postAU += 30
elsif p.balance@preAU > 100 then
   p.balance@postAU += 10
else
   p.balance@postAU += 5
endif;
```

Recall that an if-then-elsif-else expressions can be rewritten in terms of a nested if-then-else expression, according to section 6.5.5. To clarify this point, below, I show a nested if-then-else expression that is the expanded form of the above if-then-elsif-else expression (which could further expanded into implies expressions).

```
if p.balance@preAU > 50 then
   p.balance@postAU += 50
else
   if p.balance@preAU > 30 then
      p.balance@postAU += 30
   else
      if p.balance@preAU > 10 then
         p.balance@postAU += 10
      else
         p.balance@postAU += 5
      endif
   endif
endif;
```

Multiple Effects Described by If-Then-Else Expressions

An if-then-else expression that describes multiple effects (in the "body" of the expression) does not place any constraint on those effects to be executed atomically (as a unit). For example, extending the giveAwayAvailableCars operation of the CarManager system, shown earlier, one could constrain the operation to only give away the unowned cars to a person who has greater than 50 credits in his/her balance, and in addition, if this is the case, the person could be given 50 more credits. An Operation Schema for this operation is shown below.

```
Operation: CarManager::giveAwayAvailableCarsAndCredit (p: Person);
Shared: Owns; Person;
Aliases: availCars: Set (Car) Is self.car->select (c | c.owner@preAU->isEmpty ());
Post:
   if p.balance@preAU > 50 then
      p.balance@postAU += 50 &
      p.myCars@postAU->includesAll (availCars) -- p becomes owner of all available cars
   endif;
```

In the above Post clause, the then-part of the if-then-else expression defines two effects: adding 50 credits to p's balance and making p the owner of all unowned cars of that moment. Due to the predicate nature of the Post clause and the absence of order for conjunction, the time at which p's balance is updated is independent of the time at which p is made owner of the available cars, i.e., there is no temporal relationship between the executions of the effects. Clearly, in terms of implementation, it is unlikely that one would choose to allocate the car before the balance could be guaranteed, but the point to make is that logical-and does not impose order.

In some situations, it is necessary to constrain different effects to be executed as a combined update that is seen as an atomic action from the exterior (i.e., other operations). This issue is discussed in section 6.9.2.9.

If-Conditions that Include "@postAU" Decorators

An if-condition of an if-then-else expression may include expressions that have the @postAU or @rd decorators postfixed. Illustrating the use of the @postAU decorator in an if-condition, one could elaborate on the incrementSpeedometer operation, shown earlier, by taking into account the effect that incrementing the kilometers of a car can have on the deprecation level of the car. The revised Operation Schema for the incrementSpeedometer operation that takes into account these aspects is shown below.

```
Operation: CarManager::incrementSpeedometer_V2 (c: Car);
Shared: Car;
Post:
   c.kilometers@postAU += 1 &
   if c.kilometers@postAU.mod (10000) = 0 then --mod (i.e., modulo) is a prefined (numeric) operator in OCL
      c.deprecationLevel@postAU += 1
   endif;
```

The Post clause makes use of the @postAU snapshot of the kilometers attributes to make an assertion about the necessary change to the deprecationLevel attribute of the car object parameter. The if-condition imposes a pseudo-ordering constraint. I use the term pseudo-ordering constraint, because it does *not* impose upon the implementation the requirement to realize the update in the order: increment the kilometers attribute, and then increment the deprecationLevel attribute (if necessary), however, this order would be the most usual interpretation[1]. Note that it would be inappropriate to use an @rd decorator instead, because this snapshot could be taken at any time of the operation—having no relationship to the @postAU snapshot of the kilometers attribute.

1. It would be equally possible to have an implementation that performs the effects in the other order and then check after the increment to kilometers whether it needs to roll back the increment to deprecationLevel. However, this choice of implementation would be unusual, and it would make little sense for this example.

If-Conditions that Include @rd Decorators

Illustrating the use of the @rd decorator in an if-condition, one could elaborate on the giveAwayAvailableCars operation of the CarManager system, shown earlier, by ensuring that person p is only given the unowned cars if s/he has a premier driving class.

```
Operation: CarManager::giveAwayAvailableCars_V2 (p: Person);
Shared: Owns; Person;
Aliases: availCars: Set (Car) Is self.car->select (c | c.owner@preAU->isEmpty ());
Post:
   if p.drivingClass@rd = DClass::premier then
      p.myCars@postAU->includesAll (availCars)
   endif;
```

The use of the @rd decorator in the if-condition of the Post clause poses an interesting problem. What happens if the time that the snapshot is taken of the corresponding shared resource (i.e., the one denoted by the expression with the @rd decorator) does not coincide with the time that the effect implied by the *then-part* is performed?

Recall that an if-then-else is just a logical expression and no ordering is implied between the conditions and effects, unless constrained otherwise. In this example, we cannot know at which time the snapshot of the expression p.drivingClass is taken because there is no constraint relating the @rd snapshot to the update. Concretely, say the operation starts at time 0 and the atomic update made to the resource denoted by p.myCars is performed at time 2, then the operation would still fulfill the Post clause if say the read of the resource denoted by p.drivingClass@rd is taken at time 5. Using @rd decorators in if-conditions can therefore be misleading if one interprets the condition as a guard, as in procedural programming languages.

Clearly, the if-then-else expression, shown above, is not very useful if one can not guarantee that the if-condition is true at the time of the update. For instance in this example, the desired interpretation would be to only allow the person p to become owner of the unowned cars stored by the system if p had a driving class of premier at the time of the car allocation. Rather than making use of the related-snapshots convention, a new kind of expression is proposed in the next subsection for the purpose of addressing this issue and a number of other ones too.

Possible Ways to Realize If-Then-Else Expressions

An if-then-else expression does not constrain the designers to either immediate or wait semantics on the condition, i.e., whether the operation should wait on the if-condition to become true or whether the if-condition should be evaluated immediately.

For example, the following Operation Schema describes an operation that allocates a car c to a person p if p does not already own 5 cars or more.

```
Operation: CarManager::allocateCar (p: Person, c: Car);
Shared: Owns;
Post:
```

```
if p.myCars@preAU->size () < 5 then
   p.myCars@postAU->includes (c)
endif;
```

An implementation of this Operation Schema could immediately evaluate whether p has less than 5 cars and then allocate c to p given that it was the case, and if not, do nothing (else-part is true). Equally, an implementation could wait on the condition to become true and then perform the allocation when it becomes true. Both possibilities are valid refinements of the Operation Schema.

Note that an operation may be constrained to terminate execution within a certain time period due to performance constraints (see Chapter 7 for information on describing timing constraints using an ANZAC specification). Furthermore, according to the correctness rule expressed by an Operation Schema, the operation *must* eventually terminate, and therefore an operation is not allowed to wait on a condition forever. So the underlying semantics requires that the operation waits on the condition to become true for x time units, where x is a natural number, which includes zero (the immediate case) and any number that is not infinite (the wait cases).

6.9.2.8 Rely Expressions

In concurrent systems, it is often necessary to define the synchronization dependencies on shared resources between operations. As we have seen with the use of if-then-else expressions in the previous subsection, it is not possible to express some kinds of synchronization constraints.

In some situations, it would be useful to be able to describe conditions that could be relied upon to stay true during the fulfillment of an effect, i.e., for the effect to be valid. For this purpose, I introduce a new kind of expression called a rely expression, which has similarities to the concept of a rely-condition, first introduced by Jones[1] [Jon83], and it offers the possibility to express a number of synchronization constraints that are not possible to express using if-then-else expressions.

A *rely expression* declares a condition, called the rely-condition[2], and two effects, one called the rely-effect and the other one called the fail-effect. The form of a rely expression is the following:

```
rely rely-condition then
   rely-effect
fail
   fail-effect
endre;
```

1. For an overview of rely-/guarantee-conditions see Chapter 2.
2. Note that the rely-condition of rely expression has a different meaning to the one proposed by Jones; therefore, they should not be confused.

The meaning of the rely expression is that there is an obligation on the operation to fulfill the rely-effect (i.e., the boolean expression must be made true by operation) if and only if the rely-condition is observed (i.e., the boolean expression stays true) by all other activities[1] during the rely-effect fulfillment period. If there exists a moment during this period such that the rely-condition fails (i.e., the boolean expression becomes false), then there is an obligation on the operation to fulfill the fail-effect. In other words, any change to the system state by any other operation over the period of execution of the rely-effect is supposed to satisfy the rely-condition, but, if it can not be satisfied, then the operation is obligated to carry out the fail-effect instead.

It is important to stress that the rely-condition has (timing) scope only over the period that the corresponding effect is being carried out and not the period of the whole operation. It is also important to stress that the rely-condition is "evaluated" with respect to changes of system state made by all *other* operations, i.e., it does not constrain the one being described by the Operation Schema.

The syntax for the rely expression is the following:

```
rely-expression = "rely" rely-condition "then" boolean-effect-expression
                  [ "fail" boolean-effect-expression ] "endre"
```

A rely expression has a similar form to an if-then-else expression.

Illustrating a rely expression in an Operation Schema, consider an operation of the CarManager system that allocates a car to a person only if it can be relied upon that the selected person has less than 10 credits in his/her account during the allocation of the car to the person. The Operation Schema for this operation could be written in the following way:

```
Operation: CarManager::allocateCar_V2 (p: Person, c: Car);
Shared: Owns; Person;
Post:
  rely p.balance < 10 then
    p.myCars@postAU->includes (c)
  fail
    true
  endre &
  -- other effects
  ...;
```

The rely expression (of the above Post clause) ensures that if person p was given a car c then the condition that p had less than 10 credits in his/her balance was held true by all other operations immediately before, during, and immediately after the car allocation was carried out, otherwise the rely expression obliges no effect to take place (i.e., the fail-effect is true). Note that a rely expression *always* has a fail-effect. However, in the cases that it is just "true", it may be omitted from the text, where the implication is that *all* rely expressions that do not express a fail-effect assert that there is no effect, i.e., the fail-effect expression is

1. those ones other than the one executing the corresponding operation

equivalent to true. For instance, the above rely expression could be rewritten in the following way and it would have the same meaning:

```
    rely p.balance < 10 then
        p.myCars@postAU->includes (c)
    endre
```

Time Window

It is important to note that the rely-condition does not necessarily need to hold true during the whole execution of the operation, but only during the period that its encapsulated rely-effect is being carried out. This would mean that while “other effects” are being carried out, the rely-condition may or may not hold true without consequence on the evaluation of the rely expression. In other words, the rely-condition of the above Post clause may only be true (i.e., p has less than 10 credits) for a small window of time during the operation’s execution period, but as long as p was added as owner of c within this time window, the rely-effect must be satisfied.

Consider the following Post clause, which expresses two disjoint rely-conditions of two rely expressions.

```
  Post:
    rely p.balance < 10 then
        p.myCars@postAU->includes (c)
    endre &
    rely p.balance > 10 then
        p.drivingClass@postAU = DClass::premier
    endre;
```

The rely expressions in the above Post clause do no contradict each other. Rather they ensure that when their respective (encapsulated) effects are carried out that their respective rely-conditions are held true by all other operations, otherwise no obligation is placed on the operation to do anything. The usefulness of an operation that fulfills the above Post clause may be questionable, but it nevertheless makes the point.

Rely-Conditions Do Not Have Decorators

As a rule, expressions in the rely-condition of a rely expression do not have any decorators (i.e., @rd, @preAU, @postAU), because the rely-condition does not denote a single snapshot of the respective resources, like it would for an if-condition. (Theoretically, using a continuous model of time, it would mean that an infinite number of snapshots would be taken over the period that the effect is carried out. But clearly this could be loosened to all snapshots of each micro-state change on the system, or even just the changes to the resource in question).

Relating Rely Expressions to If-Then-Else Expressions

In some cases, a rely expression may define the same effect as an if-then-else expression. For example, the two expressions below have equivalent effects.

```
rely p.myCars->size () < 10 then
   p.myCars@postAU->includes (c)
endre
----
if p.myCars@preAU->size () < 10 then
   p.myCars@postAU->includes (c)
endif
```

The if-then-else expression has the same effect as the rely expression, because it takes advantage of the fact that the change defined in the then-part is the only change that can be made over the period of the update (because it is forcibly atomic). It is a matter of style as to whether one uses a rely expression or an if-then-else expression in those cases when it is possible to use both. However, in order to avoid redundancy, I strongly recommend that the rely expression be chosen in preference to the if-then-else expression. The reason is simple: rely expressions give the immediate impression that one is dealing with shared resources (if-then-else expressions, on the other hand, can be used for both shared and non-shared resources, so the distinction is not as clear).

The recommendation is stated in the form of a guideline, and it is observed by all examples given in the rest of this thesis.

Guideline 1: All if-conditions of if-then-else expressions that use the @preAU and @rd snapshots should be replaced by rely-expressions.

The only time that I make an exception to this guideline is when multiple branches are required—often needed to differentiate failure cases. In this case, an if-then-elsif-else expressions should be used. In fact, this points out a limitation of rely-expressions: there is no way to enumerate failure cases[1].

Applying a rely expression to the giveAwayAvailableCars_V2 operation from the previous subsection, which posed some problems for describing with if-then-else expressions, we get the following Operation Schema.

```
Operation: CarManager::giveAwayAvailableCars_V2 (p: Person);
Shared: Owns; Person;
Aliases: availCars: Set (Car) Is self.car->select (c | c.owner@preAU->isEmpty ());
Post:
   rely p.drivingClass = DClass::premier then
      p.myCars@postAU->includesAll (availCars)
   endre;
```

For the above Post clause, the rely expression ensures that the car allocation is made only in the case that p's driving class is premier and stays that way during the fulfillment of the allocation, otherwise no allocation is ensured.

1. The semantics for such an extension (i.e., enumerated rely expressions) is quite tricky, and consequently, enumerated rely expressions are not addressed by this work.

Fail-Part of Rely Expressions

The fail-part of a rely expression is obligated to hold only if the rely-condition fails to hold over the period of rely-effect fulfilment. To illustrate a rely expression with a meaningful fail-part, consider an elaborated version of the Operation Schema for the allocateCar operation, shown below.

```
Operation: CarManager::allocateCar_V3 (p: Person, c: Car, boss: Person);
Shared: Owns; Person;
Post:
  rely p.myCars->size () < 10 and p.balance < 100 then
    p.myCars@postAU->includes (c)
  fail
    boss.myCars@postAU->includes (c)
  endre;
```

The fail-part of the rely expression is interpreted in the following way: if all the other ongoing operations fail to keep person p's number of cars below 10 and its balance under 100 credits immediately before, during, and after the car allocation, then the boss will receive the car instead.

Negation Issues

Due to the "during" aspect of rely expressions, negation of a rely expression does not follow the same rules as it does for if-then-else expressions. For example, the following rely expression is *not* logically equivalent to the one below it:

```
rely john.balance > 0 then
  john.drivingClass@postAU = DClass::premier
fail
  john.drivingClass@postAU = DClass::second
endre

rely not (john.balance > 0) then -- i.e., john.balance <= 0
  john.drivingClass@postAU = DClass::second
fail
  john.drivingClass@postAU = DClass::premier
endre
```

The problem relates to the fact that the condition is not evaluated at a single time but many times over the fulfillment period of effect. Thus, the effect of negating a rely condition is to apply negation to each evaluation during the period.

Possible Ways to Realize Rely Expressions

Similarly to if-then-else expressions, rely expressions do not constrain the designers to either immediate or wait semantics on the rely-condition, i.e., whether the operation should wait on the rely-condition to become true or whether the rely-condition should be evaluated immediately.

For instance, an implemented operation that does a wait until the condition becomes true (assuming it will become true) and then tries to execute the code that would correspond to the body of the rely expression, rolling back if the rely-condition fails to hold during execution (where it would be obligated to fulfil the fail-effect instead), is a valid refinement. Equally, an implemented operation that immediately tries to carry out the rely-effect and fails if the rely-condition is initially false is also a valid refinement, because the rely-condition could not be held during the rely-effect fulfillment period.

This means that the starting time of carrying out the rely-effect of a rely expression can be seen as a "parameter" (a natural number of time units) of the expression. Rely expressions at the Operation Schema level abstract above a particular value for this parameter, and thus, by refinement, an implementation can specialize this parameter to a particular value (or range of values).

Another "parameter" of a rely expression that is abstracted above at the Operation Schema level is the number of attempts that can be made by the operation to fulfill the rely-effect. This means that the rely expression at the Operation Schema level does not restrict an implementation from making several attempts and associated rollbacks (in the transactional sense of the word). It does however require that the postcondition be established on termination of the operation (an obligation). In the case that the rely-effect is fulfilled, the rely-condition must have held during the fulfillment period of the last attempt (which may equally have been the first and only one). In the case that the fail-effect is fulfilled, the rely-condition must have became false at some time during the fulfillment period of the rely-effect for all attempts.

Possible Analyses that Could Be Performed on Rely Expressions

By analyzing the properties used in the rely-conditions, it may be possible to provide some hints to designers: whether immediate or wait semantics may be preferable, whether once the rely-condition becomes false it will stay that way from thereon, or whether once the rely-condition becomes true it will stay that way from thereon.

To provide an example of the last case, consider a rely expression that asserts an effect of buying popcorn only if it can be relied upon that a rugby match does not finish before the popcorn has been bought. If the rugby match object has an attribute finished, then one could describe the situation with the following rely expression:

```
rely not allblackMatch.finished then
   buyPopcorn_p (2);
endre
```

In this case, it is possible to reason that once allblackMatch.finished becomes true it will stay that way from thereon, since a particular rugby match will only ever occur once. Therefore, in design terms, it would be pointless to use a mechanism that waits for such a condition to become true if it were already false.

Being able to supply such information would assist the designers to choose appropriate control mechanisms to use. However, techniques to conduct an analysis of the rely-conditions are not discussed any further in this chapter; they are a possible area for future work.

6.9.2.9 Atomicity of Effects

In some situations, it is necessary to constrain several effects to be executed atomically from the point of view of the exterior (i.e., from the perspective of other system operations). Taking an example based on the CarManager system, consider an operation that exchanges car ownership between a buying and selling person, and that ensures that the appropriate funds are also exchanged between them. The exchange is fulfilled if it can be relied upon that the buyer has sufficient funds and s/he does not already have a car. An Operation Schema for this operation is shown below.

```
Operation: CarManager::exchangeOwnership (buyer: Person, seller: Person, c: Car);
Shared: Owns; Person;
Post:
   rely buyer.balance >= c.price and buyer.myCars->isEmpty () then
      buyer.balance@postAU -= c.price & -- note that c.price is not a shared resource
      seller.balance@postAU += c.price &
      buyer.myCars@postAU->includes (c) &
      seller.myCars@postAU->excludes (c)
   endre;
```

The interpretation of the above Post clause does not require the balance of the buyer and seller to be updated in unison, i.e., it is not required to be performed as a combined atomic action, and similarly for the exchange of ownership. Basically, the four atomic updates may take effect at different stages of the operation, and therefore each one is possibly *visible* to the exterior as four separate changes to the state of the system.

Taking the case when four separate updates are visible to the exterior, a problem arises when other operations make use of information about the state of the system while this operation is still executing. To illustrate the problem, consider a periodically executed operation of the CarManager system that checks to see if the system has over 100 car owners, and if so, it gives the system a bonus point. An Operation Schema for this operation is shown below.

```
Operation: CarManager::applyBonus ();
Shared: Owns; CarManager;
Aliases: numOfCarOwnersInSystem: Boolean Is self.car.owners->asSet ()->size ();
Post:
   rely numOfCarOwnersInSystem > 100 then
      self.bonus@postAU += 1
   endif;
```

To highlight the problem, take the situation where the exchangeOwnership operation started to execute with the system containing 100 different owners, and it has just added the tuple (buyer, c) to the Owns association, but the operation has not yet removed the tuple (seller, c) from the association. At this moment, the applyBonus operation is executed and the expression numOfCarOwnersInSystem is evaluated. In this situation, it would evaluate to 101, and the system would unrightfully receive a bonus point. Equally, in the case where the removal is executed before the addition then the system could be not given a bonus point that it in fact deserved.

Note that another reason why this behavior is erroneous is because the multiplicity of the association end role owner would be equal to 2 at the time of executing the applyBonus operation. Therefore the actual multiplicity is outside of the required 0..1 range (specified in figure 6.4). Recall that all invariants are evaluated at the pre- and post-execution time of the operations. In this case, the assumptions of the applyBonus operation would fail.

This problem highlights the necessity of combined updates that are seen as atomic from the perspective of other concurrently executing operations. For the exchangeOwnership operation, it is necessary that the exchange of ownership be visible to other operations as an atomic action, and similarly for the exchange of funds. Therefore, one needs to be able to explicitly capture those effects described by the Post clause that should be combined to form a single atomic, compound effect.

Atomic Compound-Effects

The @preAU and @postAU decorators provide a good base for defining such a constraint, because they refer to the value of the corresponding resource immediately before and after the atomic update, respectively. Thus, if the time at which the @preAU snapshot of the resource is taken is the same as the time at which the @preAU snapshot of another resource is taken, and equally for their @postAU snapshots, then the two effects described are an *atomic compound-effect*.

Providing a notation to describe an atomic compound-effect constraint, I propose a small modification to the @preAU and @postAU decorators of the corresponding expressions. As a convention for this constraint, the @preAU and @postAU decorators become @preACU and @postACU, respectively. For instance, if exprX and exprY refer to two resources that are updated atomically, then exprX@preACU is a snapshot that is constrained to be taken at the same time as exprY@preACU, and equally for @postAU snapshots, i.e., exprX@postACU is "snapped" at the same time exprY@postACU. Note that the 'C' stands for compound, and thus, "ACU" stands for Atomic Compound-Update.

In the case that there are two or more different atomic compound-effects, the numbering scheme of the related-snapshots convention is used to differentiate them, i.e., @preACU# and @postACU#, respectively, where the # is a number that is used to match common "participants" of an atomic compound-effect. For example, exprA@preACU^1 and exprB@preACU^2 are part of expressions denoting different atomic compound-effects.

With respect to the example from above, one can rewrite the Post clause in the following way to take into account the proposed convention.

```
Post:
   rely buyer.balance >= c.price and buyer.myCars->size () < 10 then
      buyer.balance@postACU¹ -= c.price &
      seller.balance@postACU¹ += c.price &
      buyer.myCars@postACU²->includes (c) &
      seller.myCars@postACU²->excludes (c)
   endre;
```

The above Post clause asserts that, if the rely-condition holds true, then the exchange of funds between the buyer and seller is an atomic action and the exchange of cars between the buyer and seller is also an atomic action.

Note that it would have been equally possible to define the four updates as a single atomic compound-update. Clearly, the choice of which granularity of atomic update to make depends on the "other" operations that can execute concurrently to the one being specified, where the decision is made by working out what we can allow them to see.

To clarify further the use of the atomic compound-effect convention, I show another example of an operation that needs to ensure that a compound effect is carried out atomically; the Operation Schema for this operation is shown below. It describes the bossLosesBet operation of the CarManager system that transfers a certain amount of credit from the boss' balance to each of the employee's balances. However, transfers of bet winnings are only allowed if the system has more than 100 bonus points and the boss has sufficient credit. The meaning of the operation is that the boss lost a bet to his/her employees and thus must pay them out[1], but s/he is exempt if either the system bonus cannot be relied upon to stay over 100 points or s/he does not have sufficient funds.

```
Operation: CarManager::bossLosesBet (boss: Person, employees: Set (Person), bet: Credit);
Shared: Owns; Person;
Messages:  Client::{NoPayOut;};
New:       bossSaved: NoPayOut;
Aliases:   totalPayOutAmount: Credit Is bet * employees->size ();
Pre: self.person->includesAll (employees->including (boss));
Post:
   rely self.bonus > 100 and boss.balance >= totalPayOutAmount then
      employees->forall (e |              -- for each employee
         boss.balance@postACU -= bet &    -- asserts that bet amount is taken from boss' balance
         e.balance@postACU += bet )       -- asserts that bet amount is put on employee's balance
   fail
      bossSaved.oclIsNew () &
      sender.sent (bossSaved)
   endre;
```

The Post clause makes use of the ACU convention to indicate that the transfer from the boss' balance to the employees' balances is an atomic compound-effect. In addition, note that by

1. Clearly, one could generalize this operation to "person loses bet" but that would not be as interesting!

using increment minus (-=) to indicate the incremental change to the boss' balance, one is able to express the relationship between the decrement of the boss' balance and the increment of each employee's balance. The other option would have been to write the following:

```
boss.balance@postACU -= totalPayOutAmount
employees->forall (e | e.balance@postACU += bet )
```

However, this form gives a less intuitive understanding of the relationship between the credit taken from the boss' balance and the credit placed on each employee's balance.

6.9.2.10 Shorthands and Parameterized Predicates that Use Shared Resources

To facilitate the writing of concurrent operations using Operation Schemas, it should be possible to use shared resources with shorthands and Parameterized Predicates without having to expand them. I now make some proposals in relation to shorthands and Parameterized Predicates with respect to atomic updates and atomic compound-updates.

Shorthands and Atomic Updates

I propose that the oclIsNew property can be used without concern for whether the resource is shared or not. This means that if the oclIsNew property of a shared resource is used in the Post clause of an Operation Schema, then it is true if and only if the corresponding resource was atomically added to the system as a result of the operation. Note that this notion of "shared" is somewhat artificial due to the way creation is modelled in Operation Schemas.

Using message creation as an example, the following expression:

```
msgA.oclIsNew ((p1 => a1))
```

is equivalent to the following one:

```
MessageA.allInstances@postAU->includes (msgA) &  msgA.all = (p1 => a1)
```

given the following declaration:

Messages: ActorX::{MessageA;}; **New**: msgA: MessageA;

As discussed in section 6.9.2.7, the output message queue is implicitly constrained to be atomically updated in all Operation Schemas that describe concurrent operations. This means that the sent shorthand asserts that the output queue of the system is atomically updated by the operation. For example, the following expression:

actorX.*sent* (msgA)

can be expanded to the following expression:

```
self.outMessages@postAU->one (m | m = msgA) &    -- from sent
msgA.destination@postAU = Set {actorX};          -- from sent
```

Shorthands and Parameterized Predicates as Part of Atomic Compound-Effects

In specifying atomic compound-effects using Operation Schemas, it can be useful to include a shorthand or Parameterized Predicate expression as part of the compound-effect. For the purpose of explicitly showing that a shorthand or Parameterized Predicate expression is part of an atomic compound-effect, I propose to use the following convention: the 'ACU decorator is postfixed to the target expression.

To illustrate this convention, consider the following Operation Schema, which describes an operation that fixes the price of the car, modifies the car to make it more desirable, and informs the client and the dealer as to the price of the car. Furthermore, it is assumed that this operation needs to be executed as a single atomic compound-effect.

```
Operation: CarManager::queryMadeOnCar (c: Car, p: Person; amount: Credit);
Shared: Car;
Messages:  Client::{NewPrice;}; Dealer::{NewPrice;};
New:       msg1, msg2: NewPrice;
Post:
   c.price@postACU = amount &
   changeCarDetails_p (c)'ACU;
   msg1.oclIsNew ((price => amount, comment => "What a Good Buy!"))'ACU &
   p.client.sent (msg1)'ACU &
   msg2.oclIsNew ((price => amount, comment => "Another Sucker!"))'ACU &
   sender.sent (msg2)'ACU;
```

The Post clause makes use of the 'ACU decorator to indicate that all effects are carried out atomically. The Parameterized Predicate that is used on the second line is defined below.

```
Predicate: CarManager::changeCarDetails_p (car: Car);
Body:
   car.deprecationLevel@postAU = lClass::lowest &
   rely car.kilometers > 10E5 then
      car.kilometers@postAU = 10E5
   endre;
```

Note that the 'ACU decorator would mean that each @preAU and @postAU decorator of this Parameterized Predicate is transformed into a @preACU and @postACU, respectively.

It is also possible to use the numbering scheme of the related-snapshots convention together with the 'ACU decorator convention.

6.9.3 Issues and Remarks

In general, the extensions that were proposed in this section for specifying concurrent operations match the declarative style of the Operation Schemas proposed in section 6.2 through section 6.8. However, I admit that the introduction of additional language constructs for enforcing synchronization constraints makes the approach considerably more complicated. Furthermore, these additional language constructs give the specifier more power to "shoot themselves in the foot" when writing Operation Schemas, because it is easier to write Post clauses that do not make sense or that exhibit obscure contradictions. It is also clear that performing any kind of formal analysis to "concurrent" Operation Schemas would be considerably more difficult than for the sequential versions. However, this is not unusual because concurrent software systems are considered to be more difficult to develop and reason about than sequential software systems [Mas98].

The 'ACU decorator convention provides a means for describing atomic compound-effects. However, I do not provide any rules or guidelines for helping specifiers to find candidate groups of effects that could form atomic compound-effects. When considering whether a group of effects should be executed atomically or not, it can be useful to look at the invariants of the system. In general, effects should be combined to form an atomic compound-effect if they break an invariant as individual atomic effects, as was shown in the case of the Operation Schema for the exchangeOwnership operation (section 6.9.2.9). The development of a comprehensive set of rules and guidelines would be a possible area for future work.

Rely expressions as they stand are able to define only two cases: success and failure. A useful enhancement would be to allow the enumeration of the different fail cases in rely expressions—in much the same way as it is possible to use elsif parts in an if-then-else expression for this purpose. Such a facility would allow one to be more precise in dealing with failure.

Another idea for an expression that I believe could be useful is one that constrains a shared resource to stay unchanged during the execution of an effect. Such an expression would have a similar form and meaning as a rely expression, except it would be subtly different, because it would impose a no-change rule, and it would not allow failure, as rely expressions do. Clearly, in the case that shared resources are updated, such a (conceptual) mechanism is already more or less available—all shared resources are only changed by the controlling operation during the period of their updates. However, for shared resources that are read rather than changed and where one must guarantee that they stay unchanged over the period of an atomic compound-update, an expression that defines such a constraint would be useful.

6.10 Example of an Operation Schema

In this section, I present an Operation Schema that describes an operation of an auction system for placing a bid. It highlights many of the different aspects of Operation Schema introduced in the preceding sections.

6.10.1 Background

This example is about placing bids in an electronic auction, which is controlled by a system called AuctionManager. A bid represents a user's intention to buy goods for a certain amount of money in the format of an English-style auction. In an English-style auction, the goods on auction are given a starting price which is, in theory, low enough to tempt other users into bidding for the goods. Users can make as many bids as they wish until the auction closes. On closure, the highest bid is compared to the reserve price of the seller to see if the goods are sold or not.

In this system, customers have credit with the system that is used as security on each and every bid. Customers can increase their credit by asking the system to debit a certain amount from their credit card, and equally they can take their credit back by having it placed on their credit card. Also, customers can allow the system to automatically withdraw a certain amount of money from their credit card at the moment that there is not enough money on their account when making a bid (i.e., the customer does not have sufficient funds to make a bid).

Customers that wish to bid in an auction must first make a request to the system to join the auction; once joined, they can make as many bids as they wish, until the auction closes. Bidders are allowed to place their bids across as many auctions as they please. A bid is accepted if it is over the minimum bid increment (calculated purely on the amount of the high bid, e.g., 50 cent increments when bid is between $1-10, $1 increment between $10-50, etc.), and if the bidder has sufficient funds, i.e. the customer's credit with the system is at least as high as the sum of all pending high bids (in all auctions) plus the bid amount proposed in that bid. This last constraint will guarantee that bids are *always* solvent. Note that this idea of solvent bids in the context of auctions comes from Kienzle [Kie01]. A complete description of the AuctionManager system can be found at [Se02a].

6.10.2 Concept Model for AuctionManager System

To be able to describe the Operation Schema for the place bid operation, which I refer to as placeBid, it is necessary to define the Concept Model for the AuctionManager system. The Concept Model for the AuctionManager system is shown in figure 6.6.

It consists of six (normal) classes: *Auction*, FixedPeriodAuction, BidTimedAuction, Bid, Customer and Credit, and two <<rep>> classes: User and CreditInstitution. The system contains seven (non-composition) associations. The SellsIn and JoinedTo associations connect customers to the auctions that they sell goods in and that they are joined to, respectively. The Makes association connects customers to their bids. The Has association connects customer to their credit account. The HasHighBid association is derived: it stands for a link between an auction and its highest bid.

The *Auction* class contains bids that are made for the goods on sale in the auction. *Auction* is an abstract class that has two subclasses: FixedPeriodAuction and BidTimedAuction. These two classes inherit all attributes and associations of *Auction*. FixedPeriodAuction signifies an auction that closes after a fixed time period and BidTimedAuction signifies an auction that closes only when a certain period has expired after the placement of the last bid.

Credit contains four attributes and one derived attribute: actualBalance represents the amount of credit that the associated customer has with the system. autoWithdraw is true if the associated customer wishes to have the system automatically take money from his/her credit card, so that it can be used for buying auctioned goods. autoWithdrawAmount is the amount of money that is withdrawn when an auto-withdraw is performed. guaranteedBalance is a derived attribute, which is discussed below. Finally, creditDetails represents the information needed by a credit institution to perform a transfer into or out of the account of the associated customer. The system also contains a similar attribute (with the same name), which represents the information needed by a credit institution to perform a transfer into or out of the account of the enterprise owning the system.

The guaranteedBalance derived attribute is defined by the following invariant:

```
context: Credit inv:
  self.guaranteedBalance = self.actualBalance - self.customer.myBids->select (b |
                                    b.auctLeading->exists (a | not a.closed))->sum (amount)
```

The HasHighBid derived association is defined by the following two invariants:

```
context: Auction inv:
  self.highBid =
    if self.bid->notEmpty () then
      self.bid->any (b | b.amount = self.bid.amount->max ())
    else
      self.bid -- i.e., the empty set
    endif;
```

The above invariant states that if there is at least one bid then the highBid of an auction (highBid is one association-end role of the derived association HasHighBid) is equivalent to the bid that has the maximum amount, otherwise highBid is equivalent to the empty set. Note that in OCL, a 0..1 association-end can be treated as a set or a single element.

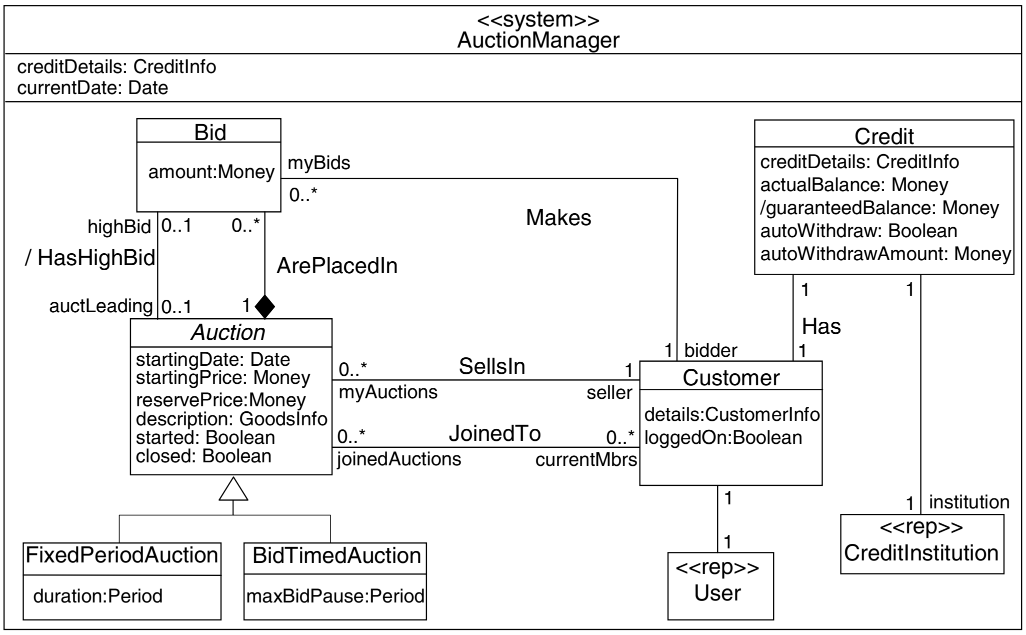

Figure 6.6: Concept Model for AuctionManager system

The invariant, below, states that the auctLeading of a bid (the other association-end role of the derived association HasHighBid) is equivalent to its containing auction (ArePlacedIn composition association).

```
context: Bid inv:
    self.auctLeading = self.auction;
```

The system has a number of state invariants. One of these invariants (which is relevant to the placeBid operation) is allPositiveBalancesForCusts. It is described using OCL, below.

```
context: Credit inv allPositiveBalancesForCusts:
    self.actualBalance >= 0;
```

The above invariant states that all (actual) customer balances must stay above zero.

6.10.3 Operation Schema for placeBid Operation

The placeBid operation represents the intention of a user to place a bid in a certain auction. This operation is responsible for determining whether the bid is valid or not. If it is valid, then the operation ensures that the bid of the user is placed in the target auction.

Concurrency and Feature Interaction

A placeBid operation can execute concurrently to a number of other operations. Of these operations, only some of them share resources with this operation. These include other placeBid operations of different users placing bids in the *same* auction, operations that change the credit of the calling user, and any operation that closes the auction. Usually, the information on which operations can perform concurrently to each other is provided by the Protocol Model. The Protocol Model is presented in the next chapter.

An interesting feature of the AuctionManager system is the auto-withdrawal option for users. This feature allows the system to augment the requesting user's credit in the system by a pre-defined amount (an amount previously agreed upon by the user) whenever the user attempts to place a bid that would not otherwise be solvent. Therefore, by performing the auto-withdraw within the execution of the placeBid operation, the user may thereafter have sufficient funds to place that bid. Note that this is an example of desirable feature interaction (the term was introduced in Chapter 3), where the auto-withdraw feature is said to interact with the place bid feature.

In terms of concurrency, the auto-withdrawal feature for a user is either turned on or off during the execution of any placeBid operation for the user, and therefore, the operation for switching the option on or off cannot execute concurrently to any placeBid operation of the user.

placeBid Operation Schema

Figure 6.7 shows the Operation Schema for placeBid. It takes as parameters an auction a, a customer c, and a bid amount bidAmount. The Shared clause shows that the operation accesses the following shared resources: one or more attributes of objects of the Credit and Auction classes, the Makes and HighBid associations, and the ArePlacedIn composition association. The Messages clause shows that the operation may (synchronously) call the Credit Institution using a transfer message. This call can throw an exception, RequestFailure_e. Also, in the case that the system judges the bid to be invalid, the system sends an InvalidBid_e Exception to the user. The Pre clause asserts that the customer c is a member of the auction a and that the auction has already started. The most complicated part of the placeBid operation is the Post clause.

```
Operation: AuctionManager::placeBid (a: Auction, c: Customer, bidAmount: Money);
Description: A user requests to place a bid in the given auction: the system must decide
whether the bid is valid and if so make the bid becomes the current high one for the auction;
Scope: Auction; Bid; ArePlacedIn; Customer; Credit; Makes; JoinedTo; HasHighBid; Has;
(Credit, CreditInstitution);
Shared: Credit; Auction; HasHighBid; Makes; ArePlacedIn;
Messages: CreditInstitution::{Transfer_c Throws RequestFailure_e;}; User::{InvalidBid_e;};
New: transfer: Transfer; invalidBid: InvalidBid_e; newBid: Bid;
Pre:
   a.currentMbrs->includes (c) &
   a.started;
Post:
❶ rely not a.closed and bidAmount >= self.minNextBidAmount_f (a.highBid.amount) then
❷     rely c.credit.guaranteedBalance >= bidAmount then
❸        placeBid_p (newBid, a, c, bidAmount) -- a parameterized predicate defined in the next subsection
      fail
❹        if c.credit.autoWithdraw then -- customer has auto-withdraw option turned on
            transfer.oclIsNew ((src => c.credit.creditDetails, dest => self.creditDetails,
                                    amount => c.credit.autoWithdrawAmount)) &
❺           c.credit.institution.sent (transfer) & -- call was made to customer's credit institution
❻           if transfer.noExceptions then -- call was successful
❼              c.credit.actualBalance@postAU += c.credit.autoWithdrawAmount &
❽              rely c.credit.guaranteedBalance >= bidAmount then
                  placeBid_p (newBid, a, c, bidAmount)
               fail
                  informInvalidBid_p (sender, invalidBid, Reason::insufficientFunds)
               endif -- informInvalidBid_p is a parameterized predicate defined in the next subsection
            else
                informInvalidBid_p (sender, invalidBid, transfer.requestFailure_e.reason)
            endif
         else
❾           informInvalidBid_p (sender, invalidBid, Reason::insufficientFunds)
         endif
      endre
   fail
❿    informInvalidBid_p (sender, invalidBid, Reason::auctionClosedOrInsufficientBid)'AU
   endre;
```

Figure 6.7: Operation Schema for placeBid

The Post clause asserts that the bid is placed by the system (asserted by the parameterized predicate marked by ❸) if during the execution of this effect:

- the auction stays open (first part of the rely-condition—❶),
- the bid stays over the minimum next bid amount (second part of the rely-condition—❶),
- and the (guaranteed) funds of the customer stays over the amount necessary to place the bid (the rely-condition at ❷).

In the case that either of the first two conditions fail during the placement of the bid or during the execution of the failure effect asserted by the fail part of the nested rely expression (❷), the requesting user is informed that the bid is invalid (the parameterized predicate at ❿). This means that the operation is obligated to send out the invalid bid message, and only that. Note that the parameterized predicates and functions are defined under the next heading.

In the case that the second rely-condition (❷) fails to hold and the customer does not have the auto-withdraw option enabled (i.e., the if-condition at ❹ is false), then the requesting user is informed that the bid is invalid (the expression at ❾).

However, in the case that condition ❷ fails to hold and the customer does have the auto-withdraw option enabled then a transfer call is made to his/her credit institution (asserted by the expression at ❺). This call is a request from the system to the customer's credit institution to transfer money from the customer's account (at the credit institution) to the account of the auction enterprise. The amount transferred is equal to the predefined auto-withdraw amount that was initially agreed upon by the customer. In the case that the transfer is successful (i.e., the if-condition at ❻ is true and a RequestFailure_e exception was not thrown), then the customer's credit in the system is (atomically) augmented by the predefined auto-withdraw amount (asserted by the expression at ❼).

The rely expression (marked by ❽) asserts that the bid is placed by the system if during the execution of this effect the (guaranteed) funds of the customer stays over the amount necessary to place the bid. This means the place bid has a second chance. Note that the augmentation of the customer's credit (❼) does not guarantee that the "second chance" will succeed. The reasons that it may not succeed are the following:

- the augmentation of the credit may still be insufficient to cover the bid;
- a different operation "stole" the credit while the place bid was being performed or before it would have been performed, and hence the rely condition could not be held; and/or
- the window of time for the rely expression did not extend past the execution period of the augmentation to the customer's credit.

Note that this last point is quite particular, because it is probably not an intuitive observation. Looking closer at this point, it is important to recognize that there is no ordering implied between ❼ and ❽ (or any expression for that matter due to &). Also, recall that rely expressions leave the choice open as to whether the implementation chooses wait or immediate semantics on the condition. As a consequence, a valid implementation of the rely expression at ❽ could continually fail on the rely-condition with each execution of the operation if it used immediate semantics and the rely-condition was always checked before the customer's credit is augmented. In this case, it is important that designers use commonsense when coming up with a design, so as to ensure that the "second chance" is not just a "no chance". This point highlights a limitation of the approach: it may not be possible to exclude all possible "silly" implementations.

Parameterized Predicates and Functions

The following Parameterized Predicates and Functions are used in the placeBid operation.

```
Predicate: AuctionManager::placeBid_p (b: Bid, auct: Auction, cust: Customer, amount: Money);
Body:
   b.oclIsNew ((amount => amount)) &-- the new bid has the given amount
   auct.bid@postACU->includes (b) & -- the ArePlacedIn composition association has the link between
      -- auct and b, which means b became part of the system state (i.e., it is equivalent to b.oclIsNew ())
   cust.myBids@postACU->includes (b); -- the Makes association has the link between cust and b

Predicate: AuctionManager::informInvalidBid_p (targetActor: User, invalidBidMsg: InvalidBid_e,
                                                r: Reason);
Body:
   invalidBidMsg.oclIsNew ((reason => r)) &   -- invalidBidMsg was created with reason set as parameter
   targetActor.sent (invalidBidMsg);          -- invalidBidMsg was sent to targetActor

Function: AuctionManager::minNextBidAmount_f (currentBid: Money): Money;
-- results in the minimum amount that can be bid given the current bid amount
```

Chapter 7: Protocol Model

This chapter presents the Protocol Model, which is one of the three models that form an ANZAC specification. It describes the various concepts that are used to express a Protocol Model, explains the way those concepts are represented in a Protocol Model, proposes some guidelines that may assist specifiers of Protocol Models, and addresses the issue of modelling concurrent system behavior. Finally, it proposes a possible way to integrate time-based properties in Protocol Models for the purpose of specifying timing constraints.

7.1 Introduction

In developing a reactive system, one needs to carefully consider its behavior over time. According to Harel [Har92]: "Behavior over time is much less tangible than either functionality or physical structure, and more than anything else, this is the aspect that renders reactive systems so slippery and error-prone". In this regard, it is important that ANZAC specifications are able to precisely describe how the system reacts to stimuli from its environment over time. For this purpose, an ANZAC specification includes the Protocol Model.

The Protocol Model defines the correctness contract on the system's input protocol. More precisely, it defines what messages from external actors and internal triggers are accepted by the system and the rules for deciding which operation (if any) to execute based on the kind of the message, its (actual) parameters and the current state of the system. The last point highlights that a Protocol Model must take into account that the system is necessarily state-sensitive. Note that incoming messages represent requests, notifications and replies made by external actors, and internal triggers correspond to time- or condition-triggered requests and notifications.

This chapter is structured as follows: In section 7.2, I explain the various concepts that are used to describe a Protocol Model. In section 7.3, I explain the notation used to construct Protocol Models and its meaning in this context. In section 7.4, I describe how one can capture temporal relationships between events using Protocol Models, and I propose some guidelines for dealing with concurrent behavior. In section 7.5, I discuss events that are generated by system time-outs, and I propose a possible way to integrate time-based properties in Protocol Models for the purpose of specifying timing constraints. In section 7.6, for the purpose of demonstrating the various concepts and notations introduced in this chapter, I present an example of a Protocol Model for a Roulette system.

7.2 Concepts and Basic Model Elements of a Protocol Model

In this section, I explain the various concepts that are used to describe the Protocol Model. The Protocol Model is described by a restricted form of UML state machine with some small enhancements for addressing some issues in describing concurrent system behavior and timing constraints. The (UML) state machine formalism is used to describe the system's behavior over time with respect to the actions that it performs in response to the requests, notifications and replies communicated to it by the actors. UML state machines are based upon the classical statecharts of Harel, both in terms of semantics and notation [BCR01]. There are however a few small variations between the two, which are described in [pp. 2-175, Omg01] and [HG97].

A state machine's view of the world is based on events. In UML, events have a correspondence to the messages that are sent between the system and its actors, and to internal triggers (time-outs or condition-changes). More precisely, the system's communication infrastructure generates an event occurrence for each message instance that it receives from an actor (instance) and for each internal trigger, and it places these event occurrences on the system's input event queue, ready for processing.

According to UML [pp. 2-148, Omg01], an event can be one of four kinds: CallEvent, SignalEvent, TimeEvent, and ChangeEvent. UML defines a call event in the following way [pp. 2-148, Omg01]: "A call event represents the reception of a request to synchronously invoke a specific operation", and a signal event in the following way [pp. 2-152, Omg01]: "A signal event represents the reception of a particular (asynchronous) signal". TimeEvent and ChangeEvents are related to internal triggers; they are discussed in section 7.5.

An event occurrence is processed by the state machine at the moment that: it reaches the head of the queue and the event occurrence that was previously the head of the queue has been consumed. At this time, it is referred to as the *current event*. State machines process event occurrences according to the run-to-completion assumption, which intuitively means that event occurrences are processed one at a time. The next event occurrence is not processed until all the consequences of the previous event occurrence have been realized.

From the contractual perspective of the design contract, the state machine of a Protocol Model can be seen as a set of rules that defines how the system is constrained to behave in all possible circumstances. However, since a state machine is a model of a system, from hereon, I refer to the state machine as though it is the system, because it is often more natural to do so, e.g., the state machine goes from state X to state Y, the state machine processes the event, etc.,

A state machine describes the reaction of a system over time to all possible event occurrences that are placed in its input event queue. Essentially, a state machine is described by two basic model elements: state and transition. A *state* defines the condition or current situation that the system is in, which corresponds to a certain subset of system states (as defined by the Concept Model), where the state machine of a Protocol Model specifies only those states that are deemed noteworthy or relevant to the description of the system's behavior at the granularity of an operation. A *transition* defines a rule for processing a certain kind of event and the way the system should react to this event. A transition denotes a relationship between two states (a source and a destination state). A transition is said to "fire", causing the state machine to go from the source state to the target state, if the state machine matches an event occurrence to the event placeholder associated to the transition, its guard is true, and it has firing priority. Note that guards are discussed in section 7.3.1 and the rules that define firing priorities for transitions are given in [pp. 2-169, Omg01][1].

The moment an event occurrence becomes the current event, the system dequeues the event occurrence, dispatches it to the state machine to process, and once the state machine is finished processing the event occurrence, it is consumed. The state machine reacts to the dispatched event occurrence in one of two ways:

- It is rejected – this occurs if all outgoing transitions of the active state(s) in the state machine do not match with the event occurrence or the guards of matching transitions fail.
- One or more transitions are fired, causing the state machine to change state(s) and possibly perform some actions (those ones associated to fired transitions) – this occurs if there are one or more outgoing transitions of the active state(s) in the state machine that match with the current event, and that have permission to fire. The state machine consumes the current event when all fired transitions have completed. A transition that has just fired does not complete before its associated actions have terminated execution.

1. Also a good discussion of firing priorities in UML is given in [BCR01].

In a Protocol Model, all actions associated with transitions are operations. Therefore, a transition that fires will take as long to complete as it does for any associated operation to terminate execution.

7.3 Notation and Semantics of Protocol Models

In this subsection, I explain the notation used to construct Protocol Models and its meaning. In particular, I discuss those model elements of UML that are used in Protocol Models, and highlight some modifications to some of these elements for the purpose of specifying concurrent behavior.

According to UML [Omg01], a state belongs to one of the following categories: *simple*, *composite*, *final*, or *pseudo-state*; a composite state can be further categorized into: *sequential*, *concurrent*, and *submachine*; and a pseudo-state can be further categorized into: *initial*, *history*, *stub*, *junction*, and *synch*. The subcategory of composite state: submachine, and the subcategories of pseudo-state: stub, junction and synch, are unused in Protocol Models. Also according to UML [Omg01], a state machine includes *external*, *internal* and *completion* transitions. Internal transitions are unused in Protocol Models. Note that from hereon, because Protocol Models use a restricted form of UML state machine, I use and explain only this subset of UML state machines.

7.3.1 Transitions

Unless stated otherwise, when I use the term transition, I mean, in UML terms, an external transition; completion transitions are discussed in section 7.3.2.2. A transition is a directed relationship between two states—the source and the target states. The direction of the transition is visually depicted by an arrow (from source to target state), where states are visually depicted by rounded boxes, and each transition has a label. Transitions may relate any kind of state to another (even the same one), within the following constraints: there can only be one outgoing transition from an initial pseudo-state and no incoming transitions; also there can be no outgoing transitions from a final state.

The label on a transition contains three parts: guard, event and action. The guard part of a transition consists of a boolean expression within square brackets, and it precedes the event part. The event part of a transition consists of the name of an event and an optional parameter list, and it precedes the action part, separated from it by the '/' symbol. The action part of a transition consists of the name of an operation, and optionally its parameters. For reasons of convenience and concision, in those cases where the event and action signature match (e.g. a CallEvent), only the action part needs to be placed in the label.

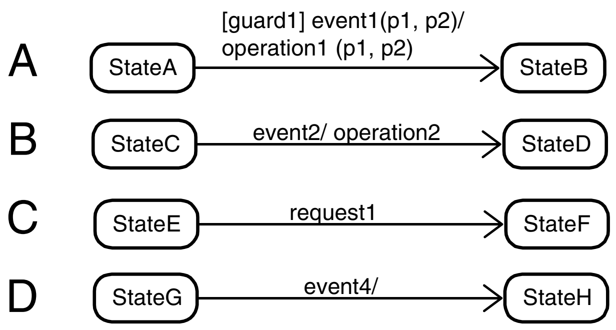

Figure 7.1: Four examples of transitions

Figure 7.1 shows the graphical representation of four different transitions between four different pairs of states. Part A of figure 7.1 shows a transition from the source state StateA to the target state StateB. The interpretation of the label on the transition is that if the system is in StateA, and the current event matches event1, then the system will change its state to StateB, and it will execute the operation operation1, if and only if the guard expression guard1 is true at that time (and the transition has firing priority). If the transition fires, then state StateA is said to go from active to inactive, and state StateB goes from inactive to active. Part B of figure 7.1 shows a transition from the source state StateC to the target state StateD. The label on this transition does not have a guard. Two interpretations are possible for a missing guard: either the guard is always true, or it has simply been left off (this situation is discussed further in section 7.4.2). Also, this example shows that it is not compulsory to show parameters. Part C of figure 7.1 shows a transition from the source state StateE to the target state StateF. The label on this transition uses a single name, where both the event name and the operation name are request1. This means that if the current event matches request1, then the transition fires, the system will change its state from StateE to StateF, and it will execute the operation request1. Note that the situation is different in part D of figure 7.1. Part D indicates that on the firing of the transition no operation is performed. An event followed by '/' implies that the action part is empty.

7.3.2 Composite States

If a state contains nested states, then it is a composite state, otherwise it is a simple state[1]. A nested state may itself be a composite state, where a state machine can have any depth of nesting. A composite state can be either sequential or concurrent, i.e., sequential composite state or concurrent composite state. Note that the terms: sequential simple state and concurrent simple state are unused. Therefore, from hereon, I use the terms sequential state and concurrent state for sequential composite state and concurrent composite state, respectively.

1. However, it may be difficult to judge whether a state is composite just by looking at a diagram of the corresponding state machine, because the diagram may provide an incomplete view of the state, i.e., another diagram may show that the state in question contains nested states.

7.3.2.1 Sequential States

Whenever a sequential state is entered exactly one of its nested states becomes active (but note that this state could be a composite state as well). A sequential state can fire at most one transition as a result of the current event, i.e., there is a single run-to-completion step for the whole state (which encapsulates an activity).

Figure 7.2 shows a sequential state called StateX. It contains two sub-states StateY and StateZ, one initial pseudo-state (filled circle on left hand side), and one final state (bulls-eye on right hand side). It also shows four transitions, three with labels. The label-less transition from the initial pseudo-state to state StateY is used to indicate the default state of the composite state. An initial pseudo-state is not a state as such, because it can never be active: it is used simply to point to the default state. If state StateX becomes active then state StateY is initially active also. If state StateY is active and the current event matches x1, then the transition from StateY to StateZ is fired and the operation that corresponds to x1 is executed, making StateZ active instead of StateY, otherwise the transition is not taken and StateY stays active. If state StateZ is active and the current event matches x2, then the self transition is fired, taking StateZ from active to inactive and back to active again; otherwise if the current event matches x3, then the composite state StateX is said to complete, where the final state of StateX becomes active. When a sequential state is in its final state then it is classed as completed. While a sequential state is in its final state no transitions can be made within StateX, only at the next level up. This means that the state machine can at most exit StateX.

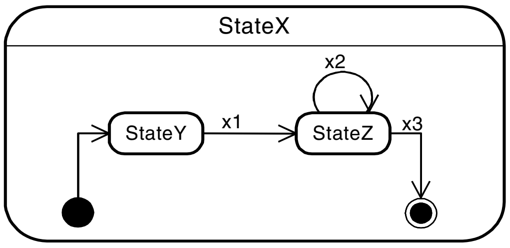

Figure 7.2: Example of a sequential state

7.3.2.2 Concurrent States

Whenever a concurrent state is entered, each of its regions become active, i.e., one state in each region becomes active. Furthermore, each region of a concurrent state can fire at most one transition as a result of the current event, assuming the regions are not themselves concurrent states.

Regions of a concurrent state are partitioned graphically through the use of a dotted line. Figure 7.3 shows a concurrent state called S2. It contains two unnamed regions. If state S2 becomes active, then both states A1 and B1 also become active initially (default entries).

Each region can then fire independently of the other. The composite state S2 completes only when both regions are in their final states.

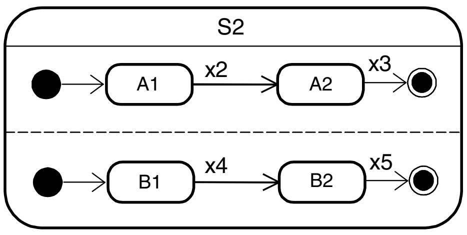

Figure 7.3: Example of a concurrent state in UML (with two regions)

7.3.2.3 Real Concurrency and UML Concurrent States

Unfortunately, concurrent states in UML do not allow two events to be processed at the same time. Thus, a UML concurrent state cannot describe truly concurrent behavior, where operations triggered by different events are executed in parallel. According to the UML specification [pp. 2-168, Omg01]:

> *"It is possible to define state machine semantics by allowing the run-to-completion steps to be applied concurrently to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement.*
> *Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration."*

As a consequence of UML's stance on concurrency in state machines, state machines that strictly adhere to this rule can not express truly concurrent behavior. Therefore, UML's choice of the term "concurrent" is unfortunate; it is probably a legacy term of Harel's statecharts.

Nevertheless, in this work, I assume that state machines can express activities that can execute concurrently to others. The definition of the semantics of concurrent state machines are out of the scope of this work, but I point interested readers to those that are defined for Harel's statecharts [DH94, HN96, HKV97]. Note however that statecharts do not use the run-to-completion assumption [HG97].

In light of this situation, from hereon, I make a distinction between a *concurrent state* and an *orthogonal state*—following [Dou99, LP99]. I classify these two kinds of (composite) states as *AND-states*. AND-states can also be auto-concurrent and auto-orthogonal states (these two are explained in a later section).

Whenever a concurrent state is active, each of its regions are active, and they can each perform a run-to-completion step in parallel to the others. On the other hand, an orthogonal state has only a single run-to-completion step for all of its regions, even though all regions are active. Consequently, all concurrent behavior described by state machines of Protocol Models is specified using concurrent states. As I will discuss in section 7.4.5, orthogonal states are used as structuring mechanisms for defining independent behavior and for avoiding state explosion.

A concurrent state is graphically differentiated from an orthogonal state by attaching the <<concurrent>> stereotype to the state in question. Therefore, figure 7.3 would be an orthogonal state, whereas figure 7.4 would be a concurrent state in a Protocol Model. This means that by default AND-states are orthogonal states (because they are without stereotype).

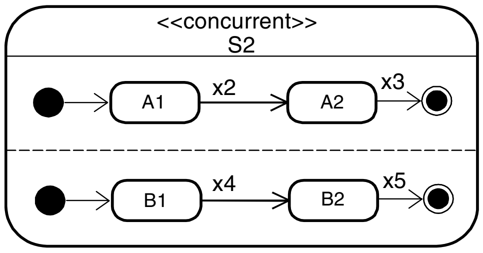

Figure 7.4: Example of a concurrent state for Protocol Models

7.3.2.4 Transitions into and out of Composite States

Transitions between states can traverse the boundaries of composite states at any depth of nesting, i.e., the source state of a transition may be a state within a composite state at any nesting depth, and/or the target state of a transition may be a state within a composite state at any nesting depth. If such a transition fires, all nested states below the source state will be exited and the new state entered [BCR00, BCR01].

A transition to the boundary of a composite state is equivalent to a transition to the default state of the composite state (the one indicated by the initial pseudostate). The initial pseudostate must be present in such cases. If the state is an AND-state, then the transition indicates a transition to the default state of each of its regions.

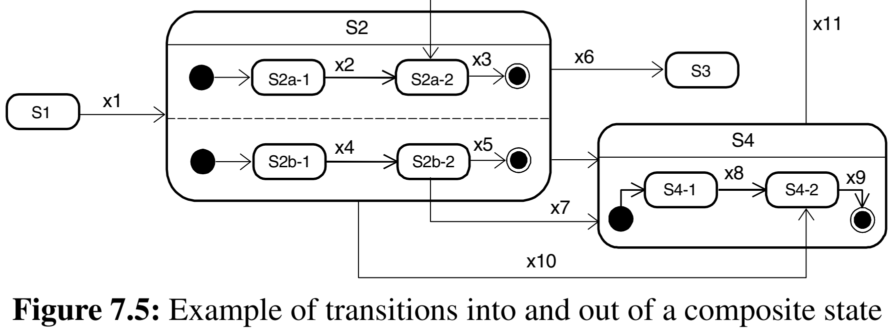

Figure 7.5: Example of transitions into and out of a composite state

Figure 7.5 shows a set of transitions into and out of states S2 and S4, an orthogonal state and sequential state, respectively. The following policies for transitions in and out of composite states are explained using this example. Some policies refer to an AND-state, but use the orthogonal state as an example; this is possible because the policies are the same for all four kinds of AND-states.

- *Transition to the boundary of an AND-state*—The firing of the x1 transition causes state S1 to become inactive and state S2 to become active, in which case both S2a-1 and S2b-1 also become active (default sub-states of S2).
- *Transition from the boundary of a composite state*—If an event that matches x6 is received, and state S2 is active, then the x6 transition fires: state S2 becomes inactive, and state S3 becomes active. Note that the transition is made regardless of which sub-states S2 was in.
- *Transition out of a composite state to the boundary of another composite state*—If an event that matches x7 is received, and state S2 and its sub-state S2b-2 are active, then the x7 transition fires: state S2 becomes inactive, and state S4 becomes active, entering its default state S4-1.
- *Transition from the boundary of a composite state into another composite state*—If an event that matches x10 is received at anytime that state S2 is active, then the x10 transition fires: state S2 becomes inactive, and state S4 and its sub-state S4-2 become active.
- *Completion transition from the boundary of a composite state*—The unlabeled transition from state S2 to state S4 denotes a completion transition. A completion transition is taken when the source (composite) state completes. This means the completion transition between states S2 and S4 fires as soon as both regions of state S2 are in their final states. The transition causes state S2 to become inactive, and state S4 to become active, entering its default state S4-1.
- *Transition into an AND-state*—If an event that matches x11 is received at anytime that state S4 is active then the x11 transition fires: state S4 becomes inactive, and state S2 and its sub-states S2-a2 and S2-b1 both become active (S2-b1 becomes active due to the default transition, since both regions of an (active) AND-state must be active).

7.3.2.5 Auto-Concurrent States

Concurrent states can be used to describe concurrent activities of the system. However, in addition to different activities that may execute concurrently, a system may be involved in multiple concurrent activities of the same kind, i.e., auto-concurrent activities. For example, a system may interact with possibly many customers who are in the process of preparing orders with the system (and it is assumed that they can perform tasks with the system in parallel). Each customer is involved with the system in a separate, but similar, activity to the others and the number of these activities that are active can change over time, since different customers can start new orders and end existing ones. It would even be possible that the system could be involved in zero order taking activities in “slow times”.

For the purpose of specifying auto-concurrent activities in Protocol Models, *auto-concurrent states* are introduced. An auto-concurrent state is similar to a concurrent state. An auto-concurrent state is represented by giving the state in question a multiplicity, like one would give to an association end in a UML class diagram. In addition, an auto-concurrent state has the <<concurrent>> stereotype attached, so that one can differentiate it from an auto-orthogonal state, which is discussed in section 7.4.7. Figure 7.6 shows an example of an auto-concurrent state on the left-hand-side (LHS) that has a multiplicity of 4, which means that the state has exactly four “copies” of the same activity. In this case, the auto-concurrent state can be “rewritten” as a concurrent state with four regions of the same activity, which is illustrated by figure 7.6 on the right-hand-side (RHS).

Therefore, if an auto-concurrent state has a multiplicity of a single number, then it can be seen simply as a shorthand notation for the expanded concurrent state version. One could also turn the argument around the other way and say that a concurrent state is an auto-concurrent state with a multiplicity of two, a sequential state is an auto-concurrent state with a multiplicity of one, and a sequential state that completes immediately (direct transition to its final state) is an auto-concurrent state with a multiplicity of zero.

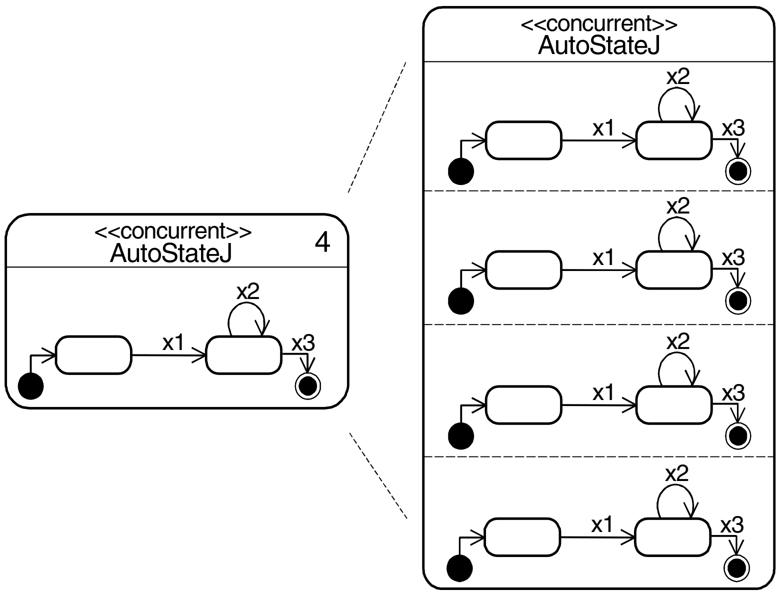

Figure 7.6: Example of auto-concurrent state (LHS) and its expanded form (RHS)

However, there is no “rewrite” for auto-concurrent states that have a multiplicity of a number range, e.g., 0..*. The interpretation of auto-concurrent states with number range multiplicities is that the number of (concurrent) regions is dynamic. On the other hand, using a concurrent state, the number of (concurrent) regions is always statically fixed (at specification time).

Figure 7.7 shows an auto-concurrent state called AutoStateK, which has a multiplicity range of 0..*. The number of regions must resolve to a certain number (any number that is greater than or equal to zero) at some point in time, and each of these regions will become active. The details of when this resolution takes place is discussed in the next subsection.

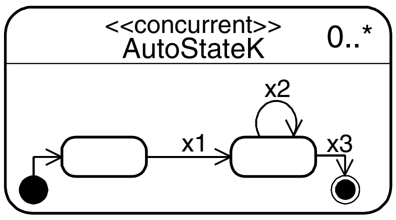

Figure 7.7: Example of auto-concurrent state with multiplicity range

7.3.2.6 UML and Auto-Concurrent States

For the purpose of specifying auto-concurrency, UML offers what it refers to as *dynamic concurrency*. Unfortunately, the use of dynamic concurrency in state machines is not recommended by UML; instead dynamic concurrency is meant for UML activity graphs.

UML's notion of dynamic concurrency can however be introduced into UML state machines by a trivial extension. The extension is trivial because UML defines activity graphs in terms of state machines, and therefore the notion of dynamic concurrency is already defined in terms of state machines. According to the UML 1.4 specification [pp. 2-145, Omg01]:

> *"... the state machine formalism provides the semantic foundation for activity graphs. This means that activity graphs are simply a special form of state machines."*

Hence, the notation used to represent auto-concurrent states in a Protocol Model is the same as the one used to represent dynamic concurrency on activities in an activity graph. That is, a multiplicity is placed in the right-hand corner of the corresponding state/activity. Therefore, auto-concurrent states in Protocol Models are in fact states with dynamic concurrency, according to UML.

Having said that, UML's notion of dynamic concurrency is not exactly what is required in a Protocol Model. The problem with dynamic concurrency is that it is not as dynamic as one would expect (or hope), because the resolution of the number of regions is fixed on entry to the state in question. Ideally, for a (truly) dynamic software system, it would be convenient to not force any binding, so the number of (concurrent) regions (when the state is active) can grow and shrink according to demand, e.g., new order activities of customers can start and finish "at will".

The reason for not allowing a more flexible notion of dynamic concurrency can be found in the following extract of the UML 1.4 specification [pp. 2-184, Omg01]:

"... an activity specification that contains 'unconstrained parallelism' as is used in general activity graphs[1] is considered 'incomplete' in terms of UML."

Having said that, for conformance reasons, I keep UML's rule for dynamic concurrency. That is, in the Protocol Model, the resolution of the number of regions is made on entry to the state (for auto-concurrent states and for auto-orthogonal states, which are discussed in the next section).

7.4 Specifying and Interpreting Protocol Models

In specifying (and interpreting) a Protocol Model, it is important to keep in mind its goal, which is to describe the acceptable temporal ordering of input events and the rules for deciding which operation (if any) to execute based on the current event. In this section, I describe how one can capture temporal relationships between events using states and transitions of a Protocol Model, and I propose some guidelines for dealing with concurrent behavior.

7.4.1 Matching Events to Operations

In describing the communication protocol between the system and its actors, it is often the case that an event name corresponds to the name of the request that the client actor has on the system. This is the case for *all* CallEvents and some SignalEvents. In these cases, the event signature and the operation signature are the same. For example, a customer may make a request to place an order with the system. In this case, only a single name is used (with an optional parameter list), as shown in figure 7.8.

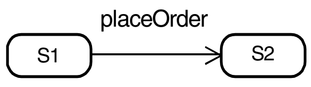

Figure 7.8: Example of a direct event/operation match on a transition

However, there are situations where the name of the event does not make a good candidate for the name of the operation. This is often the case for events that represent notifications. For example, a sensor may generate an alarm that is sent to the system, and the system's response to this alarm is to unlock all the doors. In this case, it would be better to separate the name of the event from the name of the operation that is performed, e.g., figure 7.9.

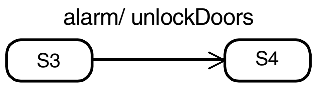

Figure 7.9: Example of a different event/operation transition pair

1. This means activity graphs that are used in other languages, i.e., not UML activity graphs.

7.4.2 Guards

In UML state machines, guards on transitions are used to indicate the circumstances in which a certain transition is desirable. A guard on a transition controls the change of state by requiring that 1) the event parameters have certain values and/or 2) the system satisfies a certain condition before or after the action of the transition has been executed. This last option means that the guard can be evaluated before or after the execution of the operation that is associated to the transition. Note that in either case, at most one guard can be true at the time of evaluation, that is, if there are two or more guards for the same event transition.

UML state machines use the junction pseudo-state to represent alternative destinations due to guards. If the guard is evaluated before the execution of the operation (i.e., the action part of the transition), then the junction is called a static branch point. If the guard is evaluated after the execution of the operation, then the junction is called a dynamic choice point. UML uses a different visual representation to distinguish the two possibilities.

In the Protocol Model, junctions are not used, because they can lead to visual clutter (an additional pseudo-state is required for each decision point), and because there are few tools that support them. Therefore, it is necessary to indicate through other means whether guards are evaluated before or after operation execution. With respect to Protocol Models, it is important to note that an event will trigger at most one operation, independent of whether the guard is evaluated before or after the operation's execution. Also, it is important to note that guards should not mention shared resources[1]. This issue is discussed further in Chapter 9.

Figure 7.10 shows two different kinds of guard use, both are evaluated before the operation is executed. The example on the left-hand-side of the figure 7.10 shows that if an order has an invalid productId as parameter, then it will cause the system to go into a different state (S2) than if it had been valid (S3). Note that both transitions imply that the same operation is executed, i.e., placeOrder_V1. The example on the right-hand-side of figure 7.10 shows that if an order is placed when the system has already 500 current orders and over, then it will cause the system to go into a different state (S3) than if it had been placed when the system had less than 500 current orders (S2).

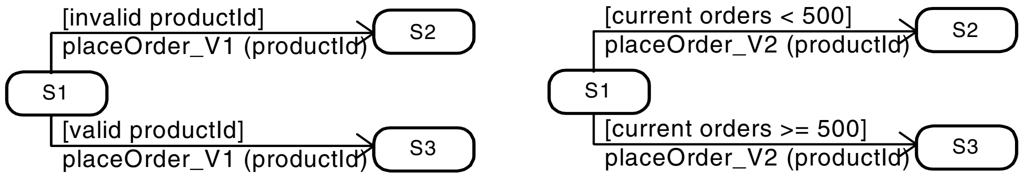
[invalid productId]
placeOrder_V1 (productId)
S2
S1
[valid productId]
placeOrder_V1 (productId)
S3
[current orders < 500]
placeOrder_V2 (productId)
S2
S1
[current orders >= 500]
placeOrder_V2 (productId)
S3

Figure 7.10: Examples of guards

1. Shared resources are resources that can be changed by possibly many operations over the execution period of any one of them.

Just as pre- and postconditions in Operation Schemas can be written in OCL (see Chapter 6), guards can also be written in OCL. In fact, it is often preferable to write guards in OCL, because this makes it easier to reason about consistency between the Operation Model and the Protocol Model, and it also forces the specifier to write guards that are grounded in the vocabulary that is defined by the Concept Model. For instance, figure 7.11 shows the same two examples as figure 7.10, but the guards have been rewritten in OCL. The OCL expressions assume that the context is the system and that the system has an attribute currentOrders and a collection of products that have the attribute id.

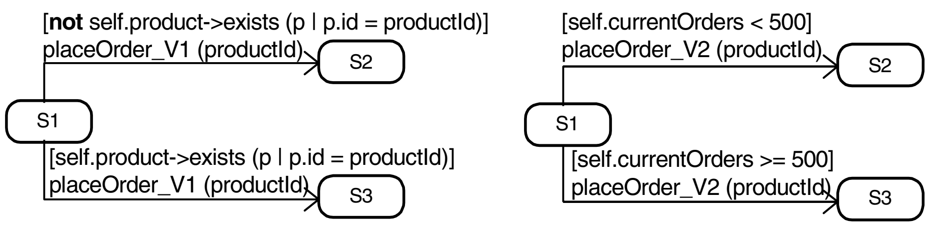
[**not** self.product->exists (p | p.id = productId)]
placeOrder_V1 (productId)
S2
S1
[self.product->exists (p | p.id = productId)]
placeOrder_V1 (productId)
S3
[self.currentOrders < 500]
placeOrder_V2 (productId)
S2
S1
[self.currentOrders >= 500]
placeOrder_V2 (productId)
S3

Figure 7.11: Examples of guards written in OCL

However, the downside of specifying guards in a Protocol Model is that they can visually clutter the diagram. Also, they often only provide information that is already stated in the corresponding Operation Schema. For example, the Operation Schema for the placeOrder_V1 operation would have a Post clause similar to the one below:

```
Post:
if self.product->exists (p | p.id = productId) then
    -- process request
else
    -- inform customer and delete his/her shopping cart
```

In light of the redundancy that is possible between the Protocol Model and the Operation Model, I propose that guards are optional in Protocol Models if they can be derived from an Operation Schema. If guards are left off Protocol Models, the diagrams become less visually cluttered with guards. However, a Protocol Model with no guards does take on a non-deterministic appearance. For example, figure 7.12 shows the transitions from the examples above without guards.

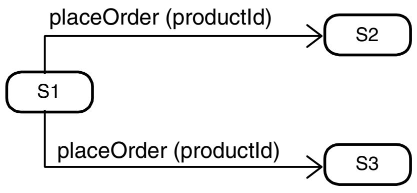

Figure 7.12: Example of guard-less transitions

The "non-deterministic" appearance is rather a product of an incomplete view rather than a non-deterministic system. The Protocol Model together with the Operation Model provide sufficient information to determine the exact behavior of the system. This highlights the point that each of the three models of an ANZAC specification, i.e., Concept Model, Operation Model, and Protocol Model, are just views on the system, which compose to define the complete specification of the system.

7.4.3 Specifying Temporal Ordering of Events

In specifying a Protocol Model, it is necessary to define the acceptable order in which events are expected, and in the case of a successful match, which operation (if any) is executed as a result.

7.4.3.1 Event Sequences

It is often necessary to specify which events precede other events and in what circumstances. For instance, the acceptance of one event may require that another event has already been processed. A sequencing constraint between two such events should be specified.

A sequencing constraint between two events evt1 and evt2, say evt1 precedes evt2, can be specified in a Protocol Model by ensuring that the target state of the evt1 transition is at least the same state or a preceding state of the source state of the evt2 transition. For example, consider an activity that represents the habits of a passive ice cream lover that must be told what to do. The ice cream lover is subject to the sequencing constraint that s/he must first accept a buyIceCream event before s/he can accept an eatIceCream event. Figure 7.13 shows a possible way to ensure that the event eatIceCream is always preceded by buyIceCream in the Protocol Model for the ice cream lover.

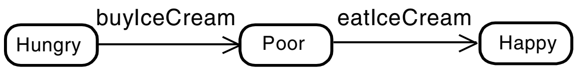

Figure 7.13: Example of a sequencing constraint between two events

7.4.4 Concurrency

In some cases, the order in which two events can be processed need not be constrained to be in a particular order; for instance, the interaction that an actor has with the system may take place concurrently to the interactions that other actors have with the system. In a Protocol Model, it is necessary to specify which operations can be performed concurrently, and which ones can not.

Inherently, each actor instance can be seen as a concurrent source of stimuli that can interact with the system independently of all other actor instances. Note that the implication is that a single actor instance is not able to interact with the system in parallel.

For multi-actor systems, it is important to be able to capture this inherent concurrency of the problem in a Protocol Model. For the purpose of identifying activities of a system that are inherently concurrent, I propose the following guideline, which I refer to as the *inherent concurrency guideline*:

> *Guideline: In a Protocol Model, independent activities of the system are deemed to be concurrent if and only if they represent activities of the external world that can happen simultaneously, otherwise the activities are deemed to be orthogonal.*

Note that the inherent concurrency guideline is not a rule, and it can and should be "overridden" in cases where the system is required to exhibit a different notion of concurrency, according to the software requirements[1].

In a Protocol Model, a concurrent state is used to specify concurrent activities. Each activity is described by a different region of the concurrent state. Furthermore, each (concurrent) region can perform its own run-to-completion step independently of the others.

To illustrate a concurrent state, consider a system that manages information on DVD rentals for a video store. While customers can take out a DVD, one at a time, the clerk should simultaneously be able to input a new DVD to the catalog. The Protocol Model for such a system would look like figure 7.14. It shows a possible way to ensure that the transitions with the events/operations addDVDToCatalog and hireDVD can take place in parallel.

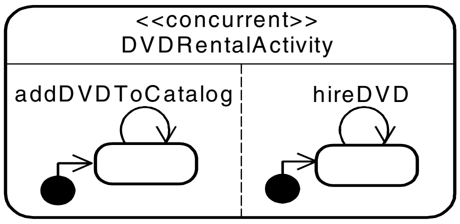

Figure 7.14: Example of specifying inherent concurrency

Note that the state shown in figure 7.14 has a <<concurrent>> stereotype attached, highlighting the fact that it is a concurrent state and not an orthogonal state.

To illustrate a requirement that "overrides" the inherent concurrency guideline, consider an activity that involves the user performing tasks using an email application and a web browser. In specifying this activity, an AND-state is used to describe the two activities of the user: one for his/her emailing task, and one for his/her browsing task. Applying the inherent concurrency guideline, these two tasks would be classed as orthogonal, since they come from a single source (clearly a specific user is physically unable to make two requests

1. Care should be taken that these kinds of requirements are not in fact premature design decisions, which are liable to change.

at the same time). However, imagine that there is a requirement that states a user should not have to wait for his/her browser to be able to work with his/her email application. In considering this requirement (albeit contrived), it would be necessary to specify the two "orthogonal" activities as concurrent, as shown in figure 7.15 (especially in considering the time that it takes to launch your average web browser and email application).

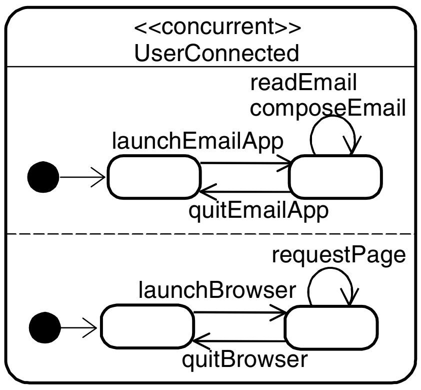

Figure 7.15: Concurrent state that does not conform to inherent concurrency guideline

From the designers' perspective, a point to take from this example is that orthogonal states are good candidates for implementation concurrency.

7.4.5 Orthogonality

If independent activities are not concurrent, they are orthogonal. There are two possible ways of specifying orthogonal transitions in a Protocol Model: either orthogonal transitions are placed in separate regions of an orthogonal state, or all possible interleavings of the transitions are specified in a sequential state by hand. In either case, there is only a single run-to-completion step for all transitions. Note that orthogonal states can not have regions that are themselves concurrent (and similarly for auto-orthogonal states).

To illustrate these two approaches, consider an example of an activity for Tarzan's daily tasks. At any one time, Tarzan is in a certain situation: he either has his sun glasses on or off, he is either on the ground or in a tree, and he is either scratching or not. Tarzan may switch between any of these states independently of the others. However, Tarzan can only change one state at a time, e.g., he cannot be putting his sun glasses on while jumping to the ground (well, not this Tarzan, anyway); this follows from the inherent concurrency guideline. Figure 7.16 illustrates the first approach for specifying orthogonal transitions, where TarzanActivity is an orthogonal state that contains three unnamed regions. Figure 7.17 illustrates the second approach for specifying orthogonal transitions, where TarzanActivity is a sequential state that contains eight sub-states.

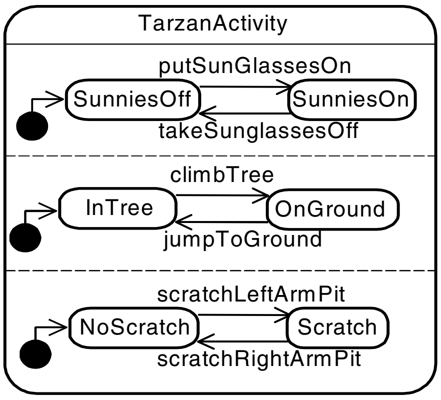

Figure 7.16: Example of specifying orthogonal transitions with an orthogonal state

These two approaches can be seen as equivalent. However, the first approach, using an orthogonal state is considerably more concise and undoubtedly easier to understand. Furthermore, the second approach is prone to state explosion (as illustrated even by this small example). Another advantage of the first approach is that orthogonal states highlight to designers operations that are good candidates for implementation concurrency. For these reasons, I strongly recommend the use of orthogonal states for specifying orthogonal transitions in Protocol Models.

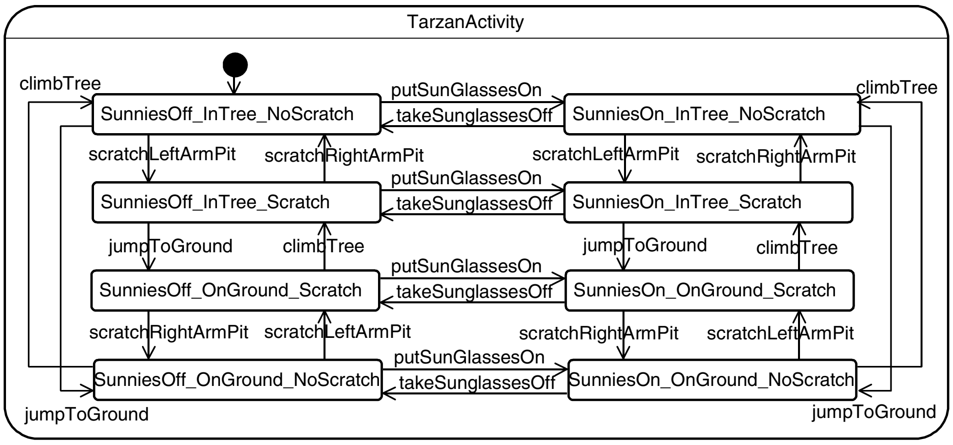

Figure 7.17: Example of explicitly specifying all possible states using a sequential state

7.4.6 Auto-Concurrent Activities

As I pointed out in section 7.3.2.5, if a system has an auto-concurrent activity, then it is able to perform the same kind of activity with multiple actors at the same time. Auto-concurrent activities are commonplace in multi-user systems, where each “replicated” activity defines the interactions between the system and a user. Auto-concurrent activities are described by auto-concurrent states in a Protocol Model. Similar to concurrent states, each region of an auto-concurrent state can perform its own run-to-completion step independently of the others.

To illustrate an auto-concurrent state, consider a game system that allows multiple players to play the same game. A player may join the game and then issue a number of game commands and then quit the game. Note that players can issue commands concurrently to others (unlike chess for example). Figure 7.18 shows an auto-concurrent state specifying that there are 2 to 10 players concurrently participating at any one time. Note that the number, i.e., between 2 and 10 inclusive, must be resolved at the time of entering the state (according to section 7.3.2.6). Each "replicated" activity described by the auto-concurrent state expresses the protocol between the system and a single participant.

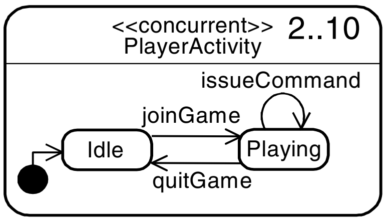

Figure 7.18: Example of an auto-concurrent state

It is important to note that events of a player are dispatched to the appropriate concurrent region, i.e., the one that corresponds to the player (this issue is discussed further in section 7.4.11).

7.4.7 Auto-Orthogonal Activities

Just as auto-concurrent activities are possible, it is also possible to have auto-orthogonal activities. An auto-orthogonal activity is very similar to an auto-concurrent activity, except the operations of an auto-orthogonal activity may not execute concurrently (usually a consequence of the inherent concurrency guideline).

In the case that there are many orthogonal (but not concurrent) activities of the same kind that can all be active at the same time, then these activities are classed as auto-orthogonal. For instance, an actor may have many activities of the same kind going on with the system at any one time. Such activities are auto-orthogonal rather than auto-concurrent, because under the inherent concurrency guideline, the interactions that the system has with a single actor are sequential.

An auto-orthogonal activity can be described by an auto-orthogonal state in a Protocol Model. Like for an orthogonal state, an auto-orthogonal state has only a single run-to-completion step for the whole state. Illustrating an auto-orthogonal state, consider an investment management system, which handles the investments of a single investor. The system allows the investor to have several investments underway at any one time, where the investor follows the protocol: speculate, then buy shares, then sell shares, for each investment.

Figure 7.19 shows an auto-orthogonal state that could be used to specify the system-investor auto-orthogonal activity.

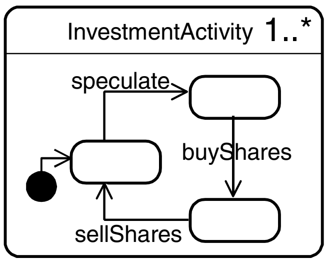

Figure 7.19: Example of auto-orthogonal activity specified by an auto-orthogonal state

7.4.8 Divide-by-Actor Technique

It is important in specifying Protocol Models that one can methodically partition the system's interactions into coherent activities. One technique that I propose for partitioning the system is the divide-by-actor technique. This technique recommends that the interaction protocol between the system and each actor type is described separately.

For each actor type, the protocol of interaction between the system and the actor type is described using a composite state, which I refer to as an actor-activity-state. Once all actor types have been covered, each actor-activity-state is given a multiplicity that matches the number of instances for the respective actor type, and it is made auto-concurrent if the multiplicity is greater than 1, according to the inherent concurrency guideline. The resulting actor-activity-states are conjoined (in the concurrent sense) with each other to form the Protocol Model for the system. The technique ensures that all events generated on the reception of messages from actors have been captured, and that all inherent concurrency has been established.

To illustrate the divide-by-actor technique, consider a system called SystemW that interacts with three different kinds of actors, which are called ActorX, ActorY and ActorZ. The system interacts with possibly many actors of type ActorX, three actors of type ActorY, and one actor of type ActorZ. According to the approach, the interaction between the system and ActorX, ActorY and ActorZ would be each described as a composite state. And then, each state would be given a multiplicity indicating the number of instances of each actor. Finally, all states are conjoined to form the Protocol Model for SystemW, which is shown in figure 7.20.

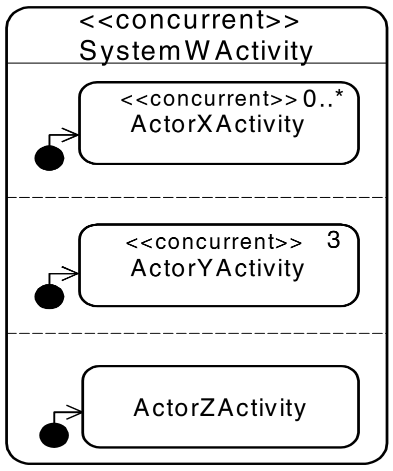

Figure 7.20: Protocol Model for the system SystemW

7.4.9 Synchronization between Regions of Different Concurrent States

The divide-by-actor technique works well for specifying systems that interact with actors in isolation. However, often the interactions that the system has with a certain actor is affected by the interactions that the system has with other actors. For example, consider an internet cafe activity that has an administrator activity and possibly many user activities, which can all perform concurrently to one another. A user first logs on to the system, allowing him/her to surf the web, and eventually s/he logs off the system. Also, the administrator has the right to remove any misbehaving users. Moreover, in describing the user activity, it would become difficult to describe the user's protocol with the system if one did not take into account the possibility that the administrator is able to log him/her off at any time.

One way to show events that affect multiple activities, and not just the one of the actor that generates the event, is to mention them in those activities that are affected in addition. Demonstrating this approach, figure 7.21 shows a concurrent state that describes the administrator and user activities. Synchronization is shown in UserActivity by adding a transition with the kickOffUser event attached. Note that the transition has a '/' suffixed after the event name. Recall that this means that there is no action associated with the transition. In fact, the transition is used purely for synchronization purposes and there is no implication of an operation in the target activity. This type of transition is referred to as a *synchronization transition* from hereon.

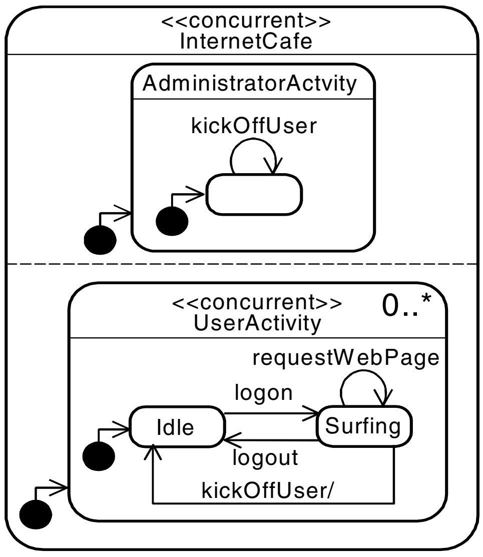

Figure 7.21: Example of synchronization between concurrent states

7.4.10 Divide-By-Collaboration Technique

For systems that exhibit collaborative behavior between actors, specifying the system's interaction protocol using the divide-by-actor technique (see section 7.4.8) can cloud the cooperative nature of the system with its actors. Furthermore, for some systems whose interaction with each actor is simple but whose collaborative behavior is complex, the divide-by-actor technique may lead to an uninformative Protocol Model of the system.

Another technique that I propose for partitioning the system is the divide-by-collaboration technique. This technique recommends that the interaction protocol between the system and its actors is described in terms of distinct types of collaboration between them.

For each collaboration type, the protocol of interaction between the system and the actors involved in this type of collaboration is described using a composite state, which I refer to as a view-state. Once all collaboration types have been described, each view-state is given a multiplicity that matches the number of collaborations that may occur at any one time in the system. Each view-state is made auto-concurrent if the multiplicity is greater than 1.

I propose that all view-states use a naming convention: the suffix 'View' is attached to the end of the name of the state. The naming convention allows the reader to visually differentiate a view-state from an actor-activity-state. Also, I propose that the interactions that have already been captured by an actor-activity-state be shown as synchronization transitions in the view-state, i.e., all labels of corresponding transitions have empty action parts. Note that the reason for this policy is that an event occurrence should only trigger a single operation at any one time in a Protocol Model.

A view-state may, however, be used as an alternative to one or more actor-activity-states. In this case, the transition will contain the corresponding operation. However, care must be taken that the view-state is in line with the concurrency requirements and it includes all operations mentioned in the actor-activity-states that are replaced by it.

It is also important to mention that all events and operations that relate to internal triggers are described by view-states. This is because, by definition, internal triggers are instigated by the system and not by an actor (directly).

The view-states found by the divide-by-collaboration are conjoined (in the concurrent sense) with each other and any actor-activity-states. In general, the divide-by-collaboration technique is applied as a supplement to the divide-by-actor technique, so as to complete the description of the input protocol of the system.

To illustrate the application of the divide-by-actor and divide-by-collaboration techniques, consider a production system that has a single worker, a single supervisor and possibly many requesters. Each requester repeatedly gives an item and picks up a finished product afterwards. The worker can repeatedly, in any order, take items, transform items into products, and put back products. The supervisor approves each and every product. Figure 7.22 shows the factory activity, which was constructed using the divide-by-actor technique. Thus, FactoryActivity show actor-activity-states for the worker, the supervisor and the requesters. Each actor is described in isolation, which makes it difficult to see the collaboration between the worker, the supervisor and each requester. Note that the auto-orthogonal state of WorkerActivity makes it easier to understand the protocol that the *single* worker is involved in. The alternative would be to describe a single sequential state that would have no ordering constraints, since the state would need to accept any of the three events in any order. In other words, even though the (auto-orthogonal) state has a multiplicity, it still only models the protocol of a *single* worker actor.

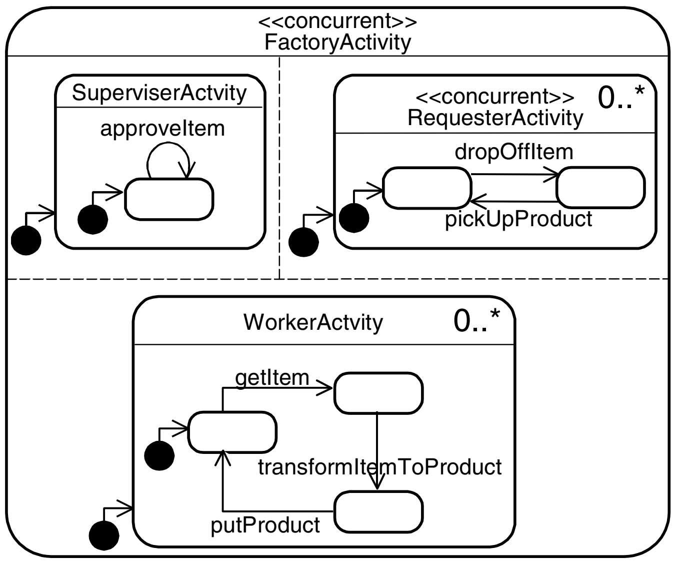

Figure 7.22: Example of Factory Activity for Production System

Applying the divide-by-collaboration technique, I specify the collaborative behavior between actors in fabricating a particular product. Figure 7.23 shows the resulting product fabrication view-state. This view-state details the collaboration between the worker, the supervisor and a particular requester. They collaborate by passing items amongst each other, where an item is put forward by a particular requester (same one each time), and making products out of them, one at a time.

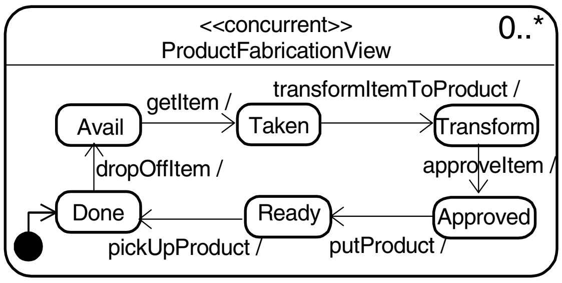

Figure 7.23: Example of a View on Production System

Note that all transitions are synchronization transitions. If this was not the case, the view-state would allow the worker and each requester to perform operations concurrently to themselves, since they both participate in multiple concurrent collaborations. This point however conflicts with the factory activity (and the inherent concurrency guideline).

7.4.10.1 Use Case View-States

A good source for finding collaboration types is high-level use case descriptions, i.e., either summary-goal level or user-goal level use cases. Furthermore, before tackling the description of a particular collaboration, it is often desirable to construct view-states for those use cases that describe a certain aspect of the collaboration. These view-states can then be composed to form a description of the complete collaboration.

However, composing use case view-states can be difficult if a common set of states are not used in each use case view-state. I therefore recommend that a common set of (sub-) states that match the essential state changes for the collaboration be defined beforehand. These states can then be used in each corresponding use case view-state.

7.4.11 Event Delivery for Concurrent States

Each concurrent state can perform its own run-to-completion step independently of all other concurrent states. Furthermore, each concurrent state is delivered those event occurrences that relate to the activity that it represents. For example, each concurrent state of ProductFabricationView (see figure 7.23) is given only those event occurrences that relate to its particular (worker, supervisor, requester) tuple. Clearly, there needs to be some mechanism to ensure that events are delivered to the appropriate concurrent states. However, the details and semantics of such a mechanism is out of the scope of this work.

7.4.12 Protocol Model Consistency

In developing a Protocol Model for a system, it is important to verify that it is internally consistent. Below, I provide a number of checks that can be performed on the Protocol Model.

- All non-synchronization transitions have an associated operation.
- View-states observe the concurrent behavior described by actor-activity-states. More precisely, all transitions in a view-state that include events that are used in an actor-activity-state are synchronization transitions.
- An event (occurrence) should not be allowed to trigger two operations in different concurrent states.
- Guards do not mention shared resources.

7.5 Specifying Internally-Triggered Actions and Timing Constraints

In this section, I describe how internally-triggered actions and timing constraints can be placed on a Protocol Model. In particular, I show how UML's change and time events can be used in state machines to specify time-triggered behavior, and I propose some extensions to the UML state machines for the purpose of specifying timing constraints in a Protocol Model.

7.5.1 Internally-Triggered Actions

In the previous sections, it has generally been assumed that events correspond to messages that are sent to the system by actors. This covers CallEvents and SignalEvents. There are nevertheless two other type events: ChangeEvents and TimeEvents, which are generated by internal triggers.

These two events cover the cases in a reactive system, where an operation may be triggered either by a time-out of the system, e.g., bank statements sent out at the end of the month, or by a change in a condition that is monitored by the system, e.g., the water level goes below 100 cm.

7.5.1.1 Change and Time Events

With UML state machines, transitions that are internally triggered by the system can be specified using change and time events.

Change events

A change event is an event that occurs when a certain condition becomes true. A change event takes as parameter a boolean expression, which denotes the condition that is monitored. According to UML [pp. 2-149, Omg01]:

> *"... conceptually, the boolean expression associated with a change event is evaluated continuously until it becomes true. The event that is generated remains until it is consumed even if the boolean expression changes to false after that."*

A change event is placed in the event part of the label of a transition. It is differentiated from other events in a state machine by the use of the **when** keyword.

Change events can be extremely useful for modeling resources that are monitored by the system, where upon a certain condition becoming true, an operation is performed in reaction to this event. A change event can be used to trigger normal and exceptional behavior.

Using the time-based properties of transitions and states that are introduced in section 7.5.2.1, it would be possible to use change events exclusively to describe all internal triggers (hence, a time event would just be syntactic sugar for a change event that makes use of a time-based property of the source or parent state). In addition, it would be possible to specify how the system reacts to violations of timing constraints.

Time events

A time event models the expiration of a specific deadline. A time event takes as parameter a time value. According to UML [pp. 2-155, Omg01]:

> *"The expression specifying the deadline may be relative or absolute. If the time expression is relative and no explicit starting time is defined, then it is relative to the time of entry into the source state of the transition triggered by the event. In the latter case, the time event instance is generated only if the state machine is still in that state when the deadline expires."*

A time event is placed in the event part of the label of a transition. It is differentiated from other events in a state machine by the use of the **after** keyword. Time events are the time equivalent to change events.

Illustration

Figure 7.24 shows a change event transition from State1 to State2 and a time event transition from State2 to State3. In the case of the first transition, as soon as the condition cond becomes true, a change event is generated and placed on the input event queue. If the state machine is in State1 and the change event is the current event, then the transition from State1 to State2 will fire. Once State2 has been active for 10 seconds, a time event is generated and placed on the input event queue. If the state machine is in State2 and the time event is the current event, then the transition from State2 to State3 will fire. Note that the parameter of the time event is relative with no explicit starting time defined, and therefore it is measured against the amount of time that the source state has been active for.

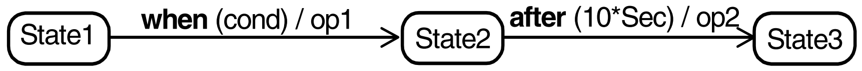

Figure 7.24: Example of change (when) and time (after) events

7.5.2 Timing Constraints

Many reactive systems exhibit behavior that is constrained by time-related factors, such as, response time, waiting time, arrival rate, the number of events processed in some interval of time, etc. Timing constraints are inherent consequences of real-time requirements. Nowadays, timing constraints are not just relevant to real-time (embedded) systems [Poo00]. Timing constraints have become particularly pertinent for internet-based systems, which are often subject to demanding performance requirements on reacting to requests.

Specifying timing constraints is an important activity in the development of reactive systems. In general, approaches that support the specification of reactive systems would better facilitate the job of designers if they provided support for specifying timing constraints that were integrated with the description of the system's behavior.

The Protocol Model is a good candidate for specifying timing constraints in an ANZAC specification, because state machines provide model elements, such as states and transitions, which can be used as building blocks for forming timing constraints, e.g., transition intervals, time that a state machine is in a state, number of transitions made over a certain period, etc.

In the tradition of the proposals made in this thesis, I propose that timing constraints be formalized in OCL rather than natural language for the same reasons that were given for formalizing guards in OCL. To realize these kinds of constraints in OCL, it is necessary to introduce timing information about transitions and states in Protocol Models.

One objective that I had in introducing timing information on transitions and states was to add as few new concepts as possible; in that way there would only be a bare minimum of concepts to learn and minimum changes would need to be made to UML. Consequently, only a very small number of additions are proposed.

7.5.2.1 Time-Based Properties

Timing information is provided to specifiers by time-based properties of transitions and states. In this subsection, I propose three new properties, two of which are properties of transitions, and the other one is a property of states.

Transition Properties

The two new properties of transitions are the following:

- transDur—The duration in time of the transition (from dispatch to completion), after its latest firing.
- numTrans—If the enclosing state is active then this value is equal to the number of times that the transition has fired since the enclosing state became active, otherwise it is equal to the number of times that the transition was fired over the period when the enclosing state was last active.

The transDur property stores a time period as value. It is equivalent to the execution time of the operation attached to the transition. This value is undefined until after the first transition taken. (Note that an expression that mentions an undefined property is itself undefined.)

The numTrans property stores a natural number as value. This value is initially set to zero. Note that although it is not a time-based property, it can nevertheless be used to define constraints on the frequency with which a transition is taken.

State Properties

The time-based property given to states is called actTime. Its meaning is the following:

- actTime—If the state is active, then this value is equal to the amount of time that the state has been active for, otherwise it is equal to the amount of time that the state was last active for.

The actTime property stores a time period as value. Similarly to transDur, this value is undefined until after the first transition taken. Note that both actTime and transDur are relative times, where it is clear at which time the timer is started, i.e., entry into state and trigger time of transition, respectively.

7.5.2.2 Using Navigation Expressions in Timing Constraints

Timing constraints can be defined on the Protocol Model in two ways: invariants or guards. Both invariants and guards can be defined using OCL. In OCL, constraints are constructed by navigation expressions, which require a starting point (to navigate from). Therefore, it is necessary to refer to the transitions and states that hold the properties of interest.

States can be specified by using the name of the state, as it appears in a state machine. For example, with respect to the Protocol Model shown in figure 7.25, one refers to state S3 simply by using the name S3. To differentiate states with the same name, but in different composite states or different levels of nesting, it is possible to fully qualify these names by the nested states and state machine that contain them. For instance, it would be necessary to write out in full states S1 and S2 so that it would be clear which ones are being referred to, e.g., System1Activity::S2::S1. A state's actTime property can be navigated to directly from the name of the state, e.g., S3.actTime, which would evaluate to the amount of time that S3 has been active (or was active) for.

Transitions are specified by using the name of a special label that is attached to them in a diagram, because, unlike states, transitions do not have names in standard UML state machines. These labels are written between curly parentheses ('{...}') to differentiate them from guard/event/action labels. Unlike names given to states, transition labels are assumed to be unique, therefore it is necessary to ensure that this is really the case in a diagram.

A transition must have a label to be used in an OCL navigation expression. The two properties: transDur and numTrans, can be navigated to directly from the transitions referred to by the label. E.g., t4.numTrans would evaluate to the number of times that the transition has been fired since the enclosing state (System1Activity) became active.

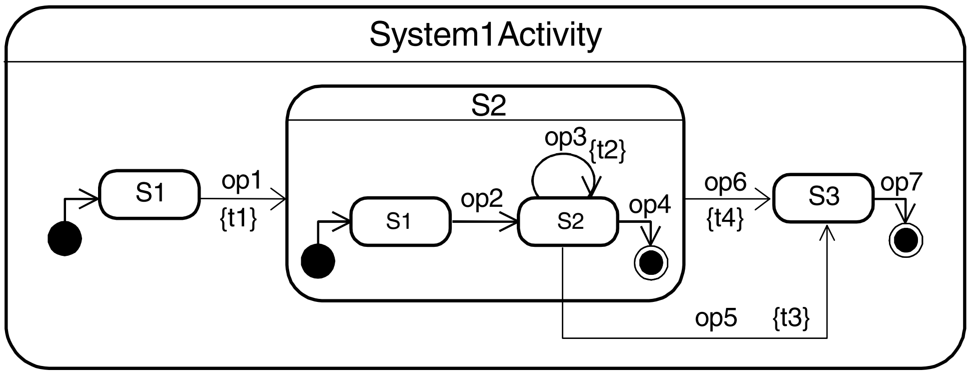

Figure 7.25: Protocol Model for System1

7.5.2.3 Invariants

A timing constraint can be defined as invariant, in which case the condition that it defines must be true in all consistent states of the corresponding system. To illustrate a timing invariant, consider the case where it is necessary to ensure that the operation op1 of the system System1 (figure 7.25, the transition from states S1 to S2) never takes longer than 5 seconds to execute. The timing invariant that captures this performance constraint is given below:

context: System1

inv: t1.transDur <= 5*Sec;

The invariant states that the transDur property of the transition labelled by t1 is always less than or equal to five seconds. Note that Sec is a constant that denotes the number of time units[1] for a second. Hence, this constraint can be seen as an obligation on the designers/implementers to deliver a system that meets this performance constraint. Note that the value for the transDur property only changes if the transition is fired, i.e., it does not change if an event is dispatched but rejected by the Protocol Model.

However, there is a slight problem with the interpretation of the above invariant when the transition has yet to fire (since invariants are checked in all consistent states of the sys-

1. The unit of time is unfortunately unspecified by UML. We could assume it to be milliseconds, in which case Sec would be equal to 1000.

tem), because the expression in the invariant will be undefined. Note that this is also a problem for the use of the actTime property of states. OCL (and UML for that matter) does not state what interpretation is given to an undefined invariant. Nevertheless, it is would seem *unreasonable* to assume that an undefined invariant is true (the desired interpretation in this case).

Note that initializing transDur and actTime to zero would not solve the problem either, because invariants that place minimum duration constraints would never hold initially. For example,

```
context: System1
inv: t1.transDur > 5*Sec;
```

Clearly, if the above invariant was evaluated with the expression t1.transDur equal to zero then the invariant will not hold, which is not the desired interpretation.

To remedy this situation, I introduce a special (global) function called defined, which is true whenever the expression is undefined, and equivalent to the truth of the expression, otherwise. Therefore, reworking the earlier invariant with the defined function, one gets the following one:

```
context: System1
inv: defined (t1.transDur <= 5*Sec);
```

The invariant now asserts what was desired: every time the transition is fired its duration is always less than or equal to five seconds.

It is important to note that in this example, the context is the system. I recommend that the context is always made to be a class of the system or the system, instead of making states or transitions the context. If states or transitions are the context of an invariant then only one property can be accessed, which would limit the specifier in the description of possible timing invariants.

Using the system as the context, one can describe more sophisticated invariants. For example, assuming that the system System1 has an attribute numOfClients, which denotes the number of clients logged onto the system, one could define the following timing invariant:

```
context: System1 inv:
if self.numOfClients < 10 then
   defined (t1.transDur <= 5*Sec)
elsif self.numOfClients < 100 then
   defined (t1.transDur <= 10*Sec)
elsif self.numOfClients < 1000 then
   defined (t1.transDur <= 50*Sec)
endif;
```

The above invariant asserts that if the numOfClients attribute of the system is under 10 then the transition labelled t1 must not take more than 5 seconds to complete, otherwise, if the attribute is under 100 (but clearly 10 or over), then t1 must not take more than 10 seconds to complete, otherwise, if the attribute is under 1000 (and 100 or over), then t1 must not take more than 50 seconds to complete, otherwise there is no constraint on the duration of the transition.

Consider another example of an invariant, which makes use of the properties numTrans and actTime: it asserts that the first 9 transitions of the transition t2 are achieved before the encompassing state has been active for less than 1 second.

```
context: System1 inv:
    defined (t2.numTrans < 10 implies System1Activity::S2.actTime < 1*Sec);
```

7.5.2.4 Guards

In addition to timing invariants, it is often necessary to state the circumstances in which events can not or should not be served because a certain time condition is not satisfied. For this purpose, guards on transitions with time conditions can be used. For example, one could define a transition with a guard that rejects all firing of the transition t3 when the source state has been active for more than 3 seconds. The following guard would be attached to transition t3:

```
[System1Activity::S2.actTime >= 3*Sec]
```

7.6 Protocol Model Example

To demonstrate the various concepts and notations introduced in this chapter, I present, in this section, the Protocol Model for a Roulette system called VirtualRoulette. This system is used in a Casino like usual roulette tables, except that the game is managed and performed electronically. Also, as an extension to the usual game of roulette, once bets have been placed, clients are allowed for a certain period to double their bets on the standing numbers (if they are feeling lucky).

VirtualRoulette allows clients to register with and log on to the system. Once logged on to the system, a client may join a roulette table and gamble. Each roulette game is performed by the system, but controlled by a single (human) game official. Each time the system determines that there is a certain number of players waiting to gamble, it opens a new table. The system closes a table that has been idle for a certain period. A game official can only be responsible for at most one table at any one time. Once a table is open for play, the game is controlled by the game official. The game official follows a standard procedure for completing a round of roulette (call for bets, end bets, play round, and collect funds and pay out winners). While a table is open, the game official can eject any gamblers that s/he dislikes from the table. Figure 7.26 summarizes all the input messages of the system from the two actors: Client and GameOfficial. Note that there is a one-to-one name correspondence between these messages and the events placed on the input event queue of the system's state machine.

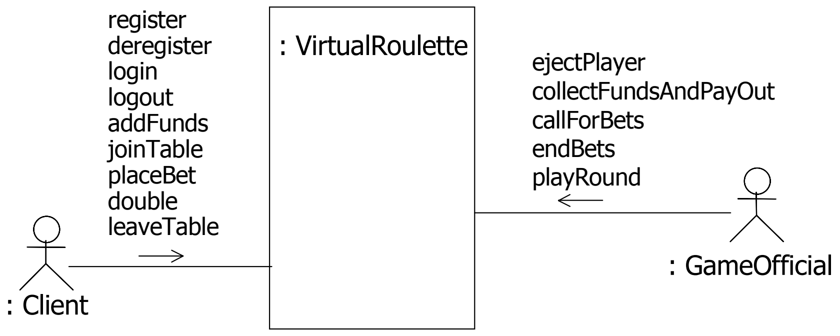

Figure 7.26: Overview of input messages for the system VirtualRoulette

In order to develop the Protocol Model for VirtualRoulette, I apply the divide-by-actor technique to the client actor, first of all. Figure 7.27 shows the result. There are possibly many client activities (auto-concurrent state). Once registered, the client is a member and s/he can log in and out of the system at will. Note that registration and logging in can fail: shown by the self transition (guards are omitted). Once logged in, the client can gamble on up to three tables and can add funds to his/her gambling credit with the system (no withdraw is provided, because it is assumed that the client is guaranteed to lose all his/her money).

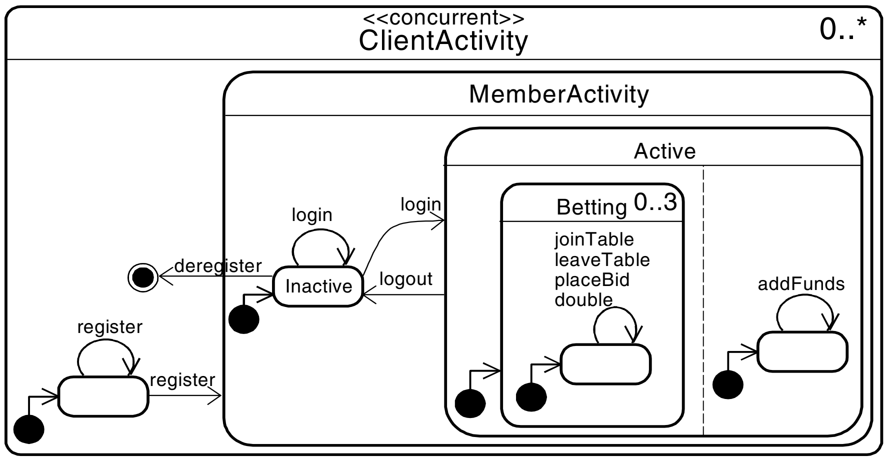

Figure 7.27: Auto-concurrent client activity for VirtualRoulette

The Betting state is somewhat uninformative as to the protocol that the user has in joining a table, placing bets, doubling, and leaving the table. The reason for this lack of information is that the betting protocol of the user is affected by the interactions that the game official of the respective table has with the system. To make the Betting state more informative, I introduce some synchronization transitions that take into account those events of the game official that relate to the user. Figure 7.28 shows the reworked betting state. It clarifies the protocol that the client has with the system in performing bets. Also, note that the self transition on the Idle state (with the joinTable transition) is shown to represent the fact that the operation may fail to allow the user to join the table, e.g., the table already has the maximum amount of participants. Note that two transitions with the same source state and event placeholder would require guards so that the target state may be determined. However, the guards have been omitted, according to the guideline stated in section 7.4.2.

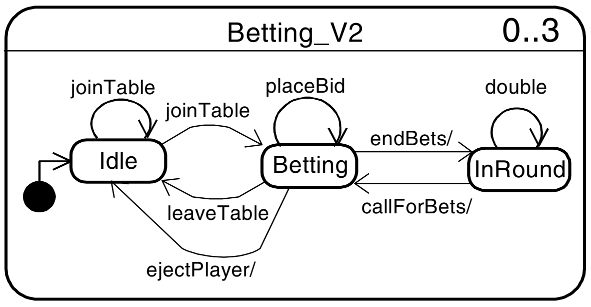

Figure 7.28: Revised Betting State: adds synchronization with GameOfficial actor

I now apply the divide-by-actor technique to the game official actor. Figure 7.29 shows the result. There are possibly many game official activities (auto-concurrent state). Once the game official becomes responsible for a table, s/he goes through the procedure of running game rounds. Also, while the table is open, s/he can eject players. The two transitions between the Idle state and the Active state are synchronization transitions that contain change and time events. The change event (event denoted by the **when** keyword) is generated each time the condition timeToOpenTable becomes true, where the meaning is that it is time for the system to open another game table. The event is processed by the game official activity, if the event causes the system to open a table that the corresponding game official actor will reside at. The time event (event denoted by the **after** keyword) is generated after the table has been idle for 5 minutes. Again, the event is processed by the game official activity if the event causes the system to close a table that the corresponding game official actor is residing at. Both the actions for opening and closing a table are part of GameTableView, which is shown in figure 7.30.

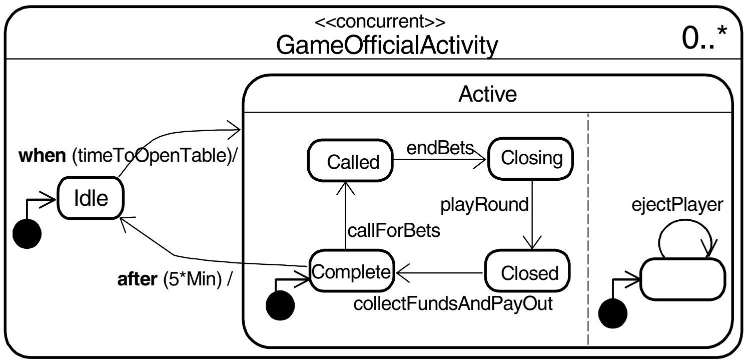

Figure 7.29: Auto-concurrent game official activity for VirtualRoulette

To clarify the sequencing between the opening and closing of a table, and the protocol between the clients and the game official during a game round, the collaboration that centers around a game table is described using a view-state called GameTableView. The view-state is an auto-concurrent state and it is shown in figure 7.30. It highlights the collaboration between the clients and the game official, and it shows the transitions that relate to the opening and the closing of the corresponding table. For example, one could imagine that the condition timeToOpenTable becomes true when the system determines that there are a certain number of clients waiting to gamble.

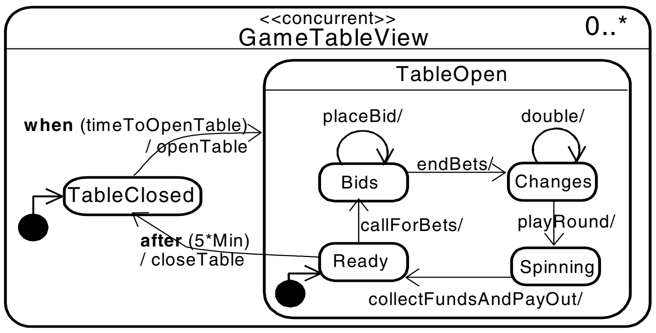

Figure 7.30: Auto-concurrent game table view activity for VirtualRoulette

Bringing all these elements together, the Protocol Model for VirtualRoulette is shown in figure 7.31.

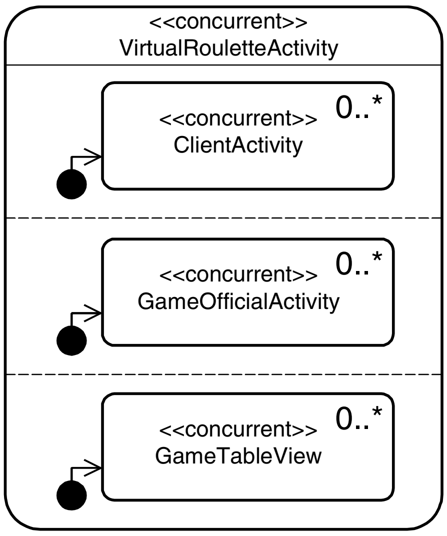

Figure 7.31: Protocol Model for VirtualRoulette

Chapter 8:
ANZAC Mapping Technique

This chapter presents the ANZAC mapping technique, which provides a systematic procedure for going from the ANZAC use case descriptions to the ANZAC specification for a system.

8.1 Introduction

The work of this thesis has focused on the development of an approach that can be used to precisely and concisely communicate to the development team the required behavior of the future system. With respect to this objective, two distinct specifications were perceived to be necessary. These specifications are referred to as the Behavioral Stakeholders Contract and the Behavioral Design Contract.

The Behavioral Design Contract is initially described by refining the Behavioral Stakeholders Contract. Therefore, it is necessary to establish a means for refining the latter specification to the former one. Such a refinement activity should also ensure that traceability between the two specifications is established, so that changes made to one can be methodically propagated to the other one.

In this chapter, I present the ANZAC mapping technique, which provides a systematic procedure for refining ANZAC use case descriptions to an ANZAC specification. The ANZAC mapping technique involves message identification from use case descriptions, collecting system responsibilities, defining the system's input protocol with its environment, formalizing the system's responsibilities, and performing checks on the resulting ANZAC specification.

In section 8.2, I describe the characteristics of ANZAC use case descriptions and the ANZAC specification that facilitate the mapping between them. In section 8.3, I describe the ANZAC mapping technique in a step-by-step manner. Note that I defer illustrating the ANZAC mapping technique on an example until the next chapter (Chapter 9).

8.2 Relating the ANZAC Use Case Descriptions to the ANZAC Specification

In this section, I describe the similarities of ANZAC use case descriptions and the ANZAC specification. Also, I discuss some of the benefits of mapping ANZAC use case descriptions to an ANZAC specification.

8.2.1 Noteworthy Characteristics of ANZAC Use Case Descriptions

An ANZAC use case description provides a plan of the possible interactions between the system and the actors in its environment in the form of a set of scenarios. These interactions form an informal statement of the system's interface to its environment and the responsibilities that it has in reacting to requests and notifications made by actors. ANZAC use case descriptions state the system's responsibilities without mentioning any of the internal details of how the software fulfills these responsibilities, i.e., they are black-box descriptions.

8.2.2 Noteworthy Characteristics of an ANZAC Specification

An ANZAC specification describes a system by offering three distinct but complementary views of it. The three views are described by the Concept Model, the Operation Model and the Protocol Model. The Concept Model describes the (abstract) state space of the system. The Operation Model states the responsibilities and assumptions that the system has in performing each operation. The Protocol Model specifies which operations are performed by the system in reaction to stimuli received over time. An ANZAC specification is a black-box description, where the internal details of the system in focus are left unmentioned.

8.2.3 Relating the Use Case Descriptions to Each Model of the ANZAC Specification

ANZAC use case descriptions provide an informal description of each responsibility that is placed on the system from interactions with its environment. These responsibilities can then be grouped together to form the description of each system operation. These descriptions can then become a basis for refinement into the Operation Model of the ANZAC specification.

ANZAC use case descriptions provide a good starting point for defining the Protocol Model of the ANZAC specification, because the set of scenarios described by each use case description provides a basis for specifying the temporal ordering of operations and for establishing the system's activities.

It is important to mention that the initial construction of the Concept Model is not performed by refinement from the ANZAC use cases. Rather, a Concept Model is initially derived from a domain model, which describes the concepts of the application domain. This "first-cut" of the Concept Model is refined with the development of the Operation Model, where the Concept Model is kept up-to-date with all the modifications that are necessary to keep synchronized with the Operation Model.

8.2.4 Benefits of Relating ANZAC Use Case Descriptions to an ANZAC Specification

The benefits of refining the use case descriptions into an ANZAC specification are the following:

- If care is taken in the refinement activity, then the responsibilities of the system that are described by an ANZAC specification can be traced to high-level goals of stakeholders. In such cases, the ANZAC specification can be justified against the goals and interests of the stakeholders. This justification allows the stakeholders greater confidence that the system under design is in fact appropriate.
- In the reverse direction, if a high-level goal of a stakeholder can be refined into a set of realistic system responsibilities and interactions with its environment, then the goal is confirmed to be viable, which is a form of goal validation.
- The refinement activity promotes iteration, because it forces specifiers to review and remove some of the ambiguities and imprecisions of the ANZAC use case descriptions, leading to use cases that have a more consistent level of detail and precision.
- The refinement activity drives the specifiers to describe more concretely the concurrent behavior of the system and the timing constraints that are needed.
- The refinement activity allows specifiers to formalize and prioritize development increments. This information can then be distributed among the development team, providing a guide for parallel development.
- The refinement activity offers the opportunity to define traceability links between ANZAC use case descriptions and an ANZAC specification. This means that changes to one of them can be propagated to the other one in a systematic manner.

8.3 ANZAC Mapping Technique

In this section, I present the ANZAC mapping technique. It is explained as a series of steps taking one from a set of ANZAC descriptions to an ANZAC specification. However, it is important to note that the process for applying the ANZAC mapping technique may be non-linear, i.e., steps may be repeated, steps may be performed out of order, etc.

A Note on Increments

If an incremental development process is applied to the ANZAC approach, then the ANZAC mapping technique would be applied to only a subset of the ANZAC use case descriptions for the system. In such cases, the use cases descriptions that are treated as input for the mapping should include at least some user-goal and sub-function goal level use case descriptions. Otherwise, the validation obtained from stakeholders with respect to the system operations would be questionable[1] and most of the work would be still left to do in the development of the ANZAC specification.

A general rule of thumb for finding a sufficient set of use cases is to ensure that the lowest goal level of the ANZAC use case descriptions have a granularity of step that is smaller than a single application-logic interaction between an actor and the system. (Note that the stress is on application-logic level, as opposed to presentation-logic level, in the case that the actor is a human, for example.)

8.3.1 ANZAC Mapping Technique Step by Step

It is assumed in the following description of the ANZAC mapping technique that the terms and concepts discussed in the chapters describing each of the models of an ANZAC specification are familiar to readers.

Step 1: From the ANZAC use case descriptions, identify incoming and outgoing messages and internal triggers, and rationalize the choices made

The aim of this step is to define all incoming and outgoing messages (relative to the system) and internal triggers from analyzing the use case descriptions. Step 1 consists of three substeps.

Step 1.1 Identify Messages and Internal Triggers

For each use case that is a leaf[2] in the use case hierarchy (tree) for the system, the steps of its main success scenario and extensions clauses are analyzed for the following three elements:

1. The ANZAC specification is less easily justified if the link between the operations and the stakeholder goals is vague or superficial.
2. Only “leaf” use cases (in the goal hierarchy) are considered, because they provide the most detailed view of the interactions between the system and actors.

- Input messages – they are derived from steps that describe an actor making a request or communicating a notification to the system.
- Output messages – they are derived from steps that describe the system making requests or communicating notifications to one or more actors.
- Internal triggers – they are derived from steps that describe system-instigated actions.

For each system-instigated action candidate found, an internal trigger definition is created, which includes a description of the trigger-condition written in OCL or natural language. For each message candidate found, a message definition is created, where the message's name matches the intention of the request or notification, and its parameters represent the necessary information to communicate to its target. All parameter types must be part of the Concept Model.

In general, input and output messages should not be difficult to find if the use case has been defined at an appropriate level of detail. Input and output messages that have a user interface flavor should be avoided, i.e., messages should represent application-logic level requests and notifications.

Internal triggers are often more difficult to extract from use cases. In general, they are most easily found after all input messages have been accounted for. At this moment, the use case can be checked to see that all system actions have corresponding messages that trigger them. If a "message-less" action is found, either a message has been missed (and a new one should be defined), or the action is system-instigated and therefore the condition that triggers the action should be defined.

Step 1.2 Rationalize Messages and Internal Triggers

Once all input and output messages and internal triggers have been found, each one should be rationalized as a distinct element with respect to the others, where the three groups are dealt with separately. This means that elements that are judged to be similar or the same are unified into a single element. Furthermore, for certain kinds of systems, there may be a large number of output messages, of which a large percentage of these are error messages (exceptions). In such cases, it may be worthwhile to group related messages and differentiate them by parameterization to reduce the clutter and enhance the clarity of the description.

Step 1.3 Annotate Use Case Description with Messages and Internal Triggers

For each use case that is analyzed, each step of the use case is annotated with all messages and internal triggers that are judged to be related to that step. Note that this does *not* mean that messages/internal triggers are only placed on those steps that led to the uncovering of the message/internal trigger, but rather that they are added to each and every step that "use" them. The annotation has the following (EBNF) form:

```
Annotation ::= "[" Body+ "]"
Body ::=   "in" "->" InputMessageName ("," InputMessageName)* ";"
           | "out"  "->" OutputMessageName ("," OutputMessageName)* ";"
           | "cond" "->" TriggerCondition ("," TriggerCondition)* ";"
```

For example, by analyzing the following step (from a use case for a Red Bull vending system), a RequestRedBull message is identified as incoming.

1. User requests System for a Red Bull

The resulting annotation to the use case step, with the RequestRedBull message, is the following one:

1. User requests System for a Red Bull [in -> RequestRedBull]

Result of Step 1

The result of this first step in the ANZAC mapping technique is a set of annotated use case descriptions, and message and internal trigger definitions. In addition to these ones, it can be useful to provide an overview of all messages exchanged between the system and its actors (i.e., those messages found after applying this step) with a specification-level UML collaboration diagram, which is shown without sequence numbers (ordering is dealt with in step 3).

Step 2: Collect system responsibilities related to each input message and internal trigger

The aim of this step is to provide a rough overview of the responsibilities that are placed on the system with respect to the reception of input messages or internal triggers (found by applying step 1).

For each input message, all steps of (annotated) use cases that indicate any actions performed by the system in response to this message are collected and related to the message definition as an informal comment. Rather than repeating the description given in the use case, the comment simply cross-references the steps of use cases that describe the corresponding responsibilities, i.e., all system responsibilities that are caused by this message. These cross-references support traceability between the use case descriptions and the ANZAC specification. Note that the actions that relate to a particular message are often part of *different* use case steps to those ones that include the message as an annotation, i.e., the "cause" may be found in a different part of the use case to the "effect". Therefore, the activity of collecting the responsibilities of the system is more complicated than just collecting all steps that have the message as an annotation.

Similarly, for each internal trigger, the step of the use case that describes the related system-instigated action is related to the internal trigger definition as an informal comment (in the form of a cross-reference to the use case step). Note that usually an internal trigger has only one related use case step.

Result of step 2

The result of this second step in the ANZAC mapping technique is a set of annotated input message and internal trigger definitions.

Step 3: Define the system's input protocol with its actors and establish operation signatures and responsibilities

The aim of this step is to define the Protocol Model for the system using the set of annotated input message and internal trigger definitions, which are found by applying step 2, and to establish the informal responsibilities of operations. Step 3 consists of five sub-steps.

Step 3.1 Map Input Messages and Internal Triggers to Events

All input messages and internal triggers are mapped to events. Each (input) message is mapped to a signal or call event with the same signature (i.e., same name and parameter list). Also, each internal trigger is mapped to a time or change event. Thus, each event corresponds to the reception of a message sent by an actor to the system or of an internal trigger generated by the system.

Input messages are mapped to signal events by default. However, if there is a specific need or requirement for synchronous communication, then the corresponding message is mapped to a call event. An internal trigger is mapped to a time event if its trigger-condition denotes a deadline expiration, otherwise it is mapped to a change event, where the trigger-condition becomes the condition associated to the change event.

Step 3.2 Apply Divide-By-Actor Technique

Once all events have been defined, the divide-by-actor technique is applied to the system. This technique is used to describe the input protocol between the system and each actor type as a separate activity. Each activity is described by a composite state, which is referred to as an actor-activity-state. By default, the divide-by-actor technique follows the inherent concurrency guideline, i.e., if there are no explicit concurrency requirements. This guideline is derived from the fact that each actor instance is a concurrent source of stimuli that can interact with the system independently of all other actor instances. The guideline states that the interaction that an actor has with the system may take place concurrently to the interactions that other actors have with the system, but a single actor (instance) can not interact with the system in parallel (in general).

For each actor, an actor-activity-state expresses the actor's (acceptable) input protocol with the system in terms of the temporal constraints on the ordering of events. And it defines the system's reaction to these events in all circumstances. The transitions in an actor-activity-state, which define the ordering constraints, include an event placeholder and an operation (the action that is triggered on firing of the transition).

An operation signature is defined for each transition. The name of the operation should always represent the effect of the responsibilities that it encapsulates[1]. Note that an operation that is associated to an event may only encapsulate some of the responsibilities that were found for the corresponding message in step 2. In such cases, the complete set of

1. Page-Jones provides a good discussion on how responsibilities should be distributed among operations and how operations should be appropriately named [P-J99].

responsibilities are split across multiple operations, where the decision of which operation to execute is based on the current state of the system.

The responsibilities of the operations are informally noted by cross-referencing the corresponding use case steps. In the case that the cross-reference to the use case step is not sufficiently explicit, the responsibilities of the operations may be reformulated in an informal description in addition. In formal development settings, it may be worthwhile to use an Operation Schema by directly filling in the Operation, Description and Use Cases clauses with this information.

If the protocol between the system and an actor is tightly coupled to the interaction of other actors with the system, then synchronization transitions can be added to enhance the description of the protocol for the actor. Note that synchronization transitions are operationless, i.e., they only contain event placeholders.

Step 3.3 Apply Divide-By-Collaboration Technique

Once the actor-activity-states have been established, the divide-by-collaboration technique is applied to the system. This technique is used to describe the input protocol between the system and the actors for each meaningful collaboration. Each collaboration is described by a composite state, which is referred to as a view-state. This technique can be particularly useful for capturing different aspects of the concurrent behavior of the system.

Before tackling the description of a particular collaboration, it is often desirable to construct view-states for those use cases that describe a certain aspect of the collaboration. These view-states can then be composed to form a description of the complete collaboration. Also, some view-states that are described using the divide-by-collaboration technique may be used as alternatives to actor-activity-states. In particular, a view-state is allowed to replace a set of actor-activity-states if it observes the concurrent nature of the system and it includes all operations mentioned in the actor-activity-states.

All events and operations that relate to internal triggers are described by view-states. The trigger-conditions of the internal trigger definitions are used to define when change and time events are generated. Moreover, operations are attached to time or change event transitions like for the other kinds of events. The responsibilities of the operation correspond to those ones described in the internal trigger definition.

As the protocol for each collaboration is progressively described using the divide-by-collaboration technique, the operations that are associated to transitions are given signatures. Similarly to the divide-by-actor technique, the responsibilities of the operations are noted by cross-references to the corresponding use case steps.

Step 3.4 Ensure Consistency between Actor-Activity-States and View-States

The Protocol Model for the system is defined by the combination of all actor-activity-states and view-states that result from the application of the divide-by-actor and divide-by-collaboration techniques. Care needs to be taken that the concurrent behavior defined by the resulting Protocol Model matches the intended one, and that at most one operation is triggered by an event occurrence.

Step 3.5 Add Timing Constraints

Once the Protocol Model is established, timing constraints are added. Timing constraints are derived from requirements that relate to response time, waiting time, arrival rate, the number of events processed in some interval of time, etc. Timing constraints will either lead to additional system invariants, the addition of guards on transitions, or additional time or change events, which trigger operations that respond to passed deadlines.

Timing requirements are informally mentioned in the "Timing Requirements" clause of an ANZAC use case description. Guidelines for specifying timing constraints on the Protocol Model can be found in Chapter 7.

Result of Step 3

The result of this third step in the ANZAC mapping technique is the Protocol Model for the system and a set of informal operation descriptions.

Step 4: Formalize the system's responsibilities by an Operation Model

The aim of this step is to define the Operation Model using the set of informal operation descriptions, which are found by applying step 3. Step 4 consists of two sub-steps.

Prerequisite of Step 4

A prerequisite for this step is that the Concept Model for the system is in a form that can be used as the reference for types in Operation Schemas. Note that the Concept Model will almost always undergo changes as each operation is described by an Operation Schema, i.e., new entities/relations will be added and others removed to take into account what is needed in the descriptions of operation effects.

The "All at Once" Approach and the "Atomic/Non-Atomic Split" Approach

The sub-steps given below take the approach of defining directly the synchronization constraints in the Operation Schemas for concurrent operations; I refer to this approach as the "all at once" approach. However, an equally viable approach is to first develop the Operation Schemas as though they are executed atomically (instantaneous-operation assumption) and then refine them to descriptions of truly concurrent operations; I refer to this approach as the "atomic/non-atomic split" approach. The advantage of the second approach is that the complexity of developing Operation Schemas is reduced, because you first concentrate on describing the functionality, and only after you have captured the functionality do you go about introducing synchronization information.

The "all at once" approach proposed below can be seen as the preferred approach for advanced specifiers, and the "atomic/non-atomic split" approach can be seen as the preferred approach for specifiers that are not so familiar with the techniques for describing concurrent operations, given in Chapter 6.

Step 4.1 Basic Structure of Operation Schema

The operation signatures and the references to their responsibilities (in the use cases), obtained in step 3, are used as the basis for constructing an Operation Schema for each operation. Given this information, the Operation and Use Cases clauses can be directly filled in. In consulting the referenced use case steps, the Description and Messages clauses usually follow in a straight-forward manner.

If the Protocol Model for the system contains concurrent states, then the shared resources for operations will need to be found. Note that if the Scope clauses of all the Operation Schemas are defined then the shared-clause algorithm (Chapter 6) can be used to calculate the Shared clause for each Operation Schema. If shared resources are found, then the Operation Schema should be described using the techniques described in Chapter 6 for concurrent operations.

Step 4.2 Describe the Pre- and Postcondition Assertions

The Pre clause is developed by reflecting on what assumptions are made about the parameters of the operation and state of the system before the operation's execution. These assumptions are described in terms of assertions.

The Post clause is developed by consulting the relevant use cases (referenced by the Use Cases clause) and the Concept Model. The corresponding steps in the use cases provide information about the operation's responsibilities. These responsibilities need to be transformed into assertions about the effect of the operation on the state of the system and the messages that are output with the operation's execution. All assertions are made on entities and relations that specified by the Concept Model, or about messages that are declared in the Messages clause.

Result of Step 4

The result of this fourth step in the ANZAC mapping technique is an established Operation Model and Concept Model.

Step 5: Perform checks on the resulting ANZAC specification

The aim of this step is to check the resulting ANZAC specification for correctness, consistency and completeness, and to ensure that it has been correctly refined from the ANZAC use case descriptions. Each model of the ANZAC specification should be correct in terms of its syntax and type usage; consistent with respect to itself, the other models, and the use case descriptions; and complete with respect to the other models and the use case descriptions.

In addition to standard syntax and type checks (which are covered by the respective chapters of the three models), the following checks should be performed to each model:

- Operation Model:
  - The output messages that are declared in the Messages clause of each Operation Schema match those identified in step 1 of the mapping technique.

- All classes, attributes and associations that are used by the Operation Schemas are defined in the Concept Model.
- The Use Cases clause of each Operation Schema references actual steps of use cases.
- All Parameterized Predicates and Functions used in the Operation Schemas are defined.
- The Operation clause of each Operation Schema declares parameters that are mentioned in the signature of the triggering event(s).
- The resources that are mentioned in the Shared clause of each Operation Schema are shared with operations that may execute concurrently to the respective operation (according to the Protocol Model).

• Protocol Model:
- Each input message and internal trigger (found by applying step 1) has one corresponding event type, and this event is associated to one or more transitions in the Protocol Model.
- No events other than those ones that have been defined in step 3 are mentioned in the Protocol Model.
- All operations shown in the Protocol Model are described by Operation Schemas.

• Concept Model:
- The Concept Model does not include any concepts that are not mentioned in the Operation Model[1].
- All invariants defined by the Concept Model (which includes multiplicities) are not contradicted by the Pre and Post clauses of the Operation Schemas.

As a general check, the traceability between the Operation and Protocol Model and the ANZAC use case descriptions should be examined. In particular, it should be ensured that the system's responsibilities, which are described by steps of use case descriptions, are correctly represented by the Operation Schemas. Also, the ordering of messages implied by the use cases should be correctly described by the Protocol Model.

1. This is an important check, because the Concept Model should strictly only describe the state space of the system, and not just any concept that is related to the system.

Chapter 9:
Elevator Case Study

This chapter applies the ANZAC approach to an elevator control system. It takes the reader progressively through the development of the system using the ANZAC approach, offering a full treatment of it. In particular, it presents the ANZAC use case descriptions and the ANZAC specification for the elevator control system, and it illustrates the steps taken to map the first to the second one using the ANZAC mapping technique. The concurrent behavior and timing requirements of the elevator control system are also addressed.

9.1 Introduction

The elevator control case study is concerned with the development of an embedded system for controlling the movement of a single elevator cabin and the opening of the cabin's door. The system, which I refer to as ElevatorControl, bases its decisions for controlling the cabin and door on the requests made by the users of the elevator and information it receives from the door, sensors and motor controllers. Apart from offering a reasonably complete illustration of the ANZAC approach, this case study offers a different challenge to some of the examples given in previous chapters, because ElevatorControl communicates with controllers that have predefined interaction protocols.

9.1.1 Problem Statement for the ElevatorControl system

The ElevatorControl system is responsible for controlling an elevator cabin for the purpose of serving requests made by elevator users. A user requests for the cabin at a particular floor (pick up) or s/he requests the cabin to go to a particular floor (drop off); they are referred to as external requests and internal requests, respectively. The system does not treat user requests directly but rather receives them via a request intermediary, i.e., the request intermediary communicates to the system the requests that originate from floor and cabin buttons. There is one request intermediary for all internal requests and one for all external requests (i.e., two in all). Furthermore, requests are direction-less (in the case of external requests), they are definitive, i.e., they cannot be cancelled, and they should eventually be served. Also, once a request has been served, the system should inform the appropriate request intermediary of this fact.

The system may ask the cabin door to open. The cabin door[1] is responsible for closing itself and it informs the system when it has just closed. This means that the system is not responsible for telling the door to close. In the case that the door is already open and it is told to reopen, the door will continue to stay open for a new time period. The door is guaranteed to stay open for at least 5 seconds.

The arrival of the cabin at a floor is detected by a sensor (there is a sensor for each floor). On detection of the cabin, the sensor informs the system of this situation. It takes at least 1.5 seconds and at most 2 seconds to go from one floor to another one, i.e., the time between sensor events. The system may ask the cabin motor to go up, go down or stop. The motor will stop the cabin correctly at the approaching floor, given sufficient notice. Thus, all complexities involved in calculating how to stop (deaccelerations, etc.) is dealt with by the motor and is not a responsibility of the system. Once the motor has stopped the cabin, it informs the system that it is stopped.

In stopping the cabin, the system must give the motor sufficient time to stop the cabin. In terms of timing requirements, "sufficient time" means that the system must decide within 0.1 seconds if it is to stop the cabin.

9.2 ANZAC Use Case Descriptions

In this section, I develop the ANZAC use case descriptions for the ElevatorControl system by applying the ANZAC approach and given the problem statement.

For size reasons, I do not discuss the approach taken to develop the use cases: I just concentrate on describing the results. Also, I consider only a single user-goal level use case for the ElevatorControl system (which is refined into two sub-functional level use cases): the goal of a User actor to take the cabin from one floor to another, which is called Take Elevator. This use case is shown in figure 9.1. The use case's primary actor is User and its secondary actors are not visible at this level of description.

1. Only one door is mentioned because logically an elevator can be seen as having a single door consisting of two parts (i.e., the cabin door and the floor door) that are always synchronized.

Use Case: Take Elevator
System: ElevatorControl
Level: User-Goal
Intention in Context: The User intents to go from one floor to another using the elevator. The System has a single cabin that may service many users at any one time. The Users are predominantly office workers that are taking the elevator for the purpose of: going to and coming from work.
Primary Actor: User
Main Success Scenario:
1. User enters cabin
2. User exits cabin at destination floor
Extensions:
1a. User fails to enter cabin; use case ends in failure
Other Concurrent Use Cases:
This use case can perform concurrently to possibly many other "Take Elevator" use cases.

Figure 9.1: Take Elevator use case for ElevatorControl system

The Take Elevator use case has two steps in its main scenario, which are both sub-use cases[1], and it has a single extension step, which is taken on the failure of a performance of the Enter Cabin (sub-) use case. The Take Elevator use case provides the context of the users that are taking the elevator (Intention in Context clause), which is useful to know when deciding which scheduling policy might be appropriate, for example. Also, note that no secondary actors appear at this (quite abstract) level of description.

Enter Cabin use case

The Enter Cabin use case captures the goal of the user to enter the elevator cabin at the requested floor. Note that this is a sub-goal of the user's goal to take the elevator from one floor to another. However, no mention of this fact is made in the Take Elevator use case so as to promote reuse. Thus, each level in the use case hierarchy is isolated from the upper level. The Enter Cabin use case has six main scenario steps. The third and fourth steps are related: the system is informed about the floors that the cabin is arriving at (step 3); using this information, the system can determine that the cabin has reached the source floor of the request (the one where the user is waiting), leading to the system stopping the cabin at that floor (step 4). Note that this use case defines some timing requirements, which are taken verbatim from the problem statement and associated with the appropriate steps. Also, note that the Motor, Door and Sensor secondary actors are now visible. However, the intermediary role of the ExtReqSource is still unmentioned by the use case.

1. The underlined text in these two steps signifies a hyperlink to the corresponding descriptions of the sub-use cases (symbolic in this case).

Use Case: Enter Cabin
System: ElevatorControl
Level: Sub-Function
Intention in Context: The User intents to enter the elevator cabin at a certain floor.
Primary Actor: User
Main Success Scenario:
1. User makes a request to System for cabin
2. System requests Motor to move towards source floor; Motor goes in direction of source floor
Step 3 is repeated until System determines that the source floor of the User has been reached
3. Sensor informs System that cabin has reached a certain floor
4. System requests Motor to stop cabin; Motor stops cabin and informs System that it is stopped;
5. System requests Door to open, and informs User that request has been fulfilled; Door opens
6. User enters cabin at source floor

Extensions:
2a. System is serving another request:
  2a.1. System schedules the request; use case continues at step 2 (when System decides to serve this request).
2b. System determines that cabin is already at requested floor; use case continues at step 5[1].
(2-6)a. User walks away[2]; use case ends in failure

Timing Requirements:
Step 3: “It takes at least 1.5 seconds and at most 2 seconds to go from one floor to another one”.
Step 4: “the system must decide within 0.1 seconds if it is to stop the cabin”.

Notes:
[1] It is possible that the door may already be open (assumption: requesting door to open while it is already open will make it stay open for a new time period).
[2] This is not detectable by the System. The only way the System could determine that the user has left is if it times-out on waiting for a request.

Figure 9.2: Enter Cabin use case for ElevatorControl system

The Enter Cabin use case has three extension steps. The first two extension steps are alternatives to step 2. The first extension step (2a) is taken if the system is already busy with another request. In this case, the system schedules the request for later. This extension step rejoins the main scenario at step 2, which indicates that the request will eventually be served by the system. The second extension step (2b) is taken if the request made by the user is for the same floor as the one where the cabin is stationed. In this case, the use case joins the main scenario where the system asks the door to open. The third extension step (2-6 a) is an alternative to any step between 2 and 6 inclusive. The step is used to show the scenario when the user simply walks away. Although this step is not directly detectable by the system, it is worth mentioning because it is a fact of the environment that may at some point be relevant to the system. For example, this point may deter any decisions that would force the cabin to wait for the user’s internal request (after the external request has been serviced).

Exit Cabin use case

The Exit Cabin use case captures the goal of the user to exit the cabin at his/her destination floor, which is a sub-goal of taking the elevator from one floor to another (Take Elevator use case). The Exit Cabin use case has seven main scenario steps[1]. The first two can be performed in any order, because the door may close before or after the user makes a request. Steps 3-6 are similar to those found in the Enter Cabin use case (steps 2-5). Note that this use case does not cover the case when a user does not make a request. In this case, the user can be seen as a passive rider, who is taken "on the back" of another user who makes the request. This case is not deemed sufficiently noteworthy to mention in the use case. Also, this use case defines some timing requirements, which are taken verbatim from the problem statement and associated with the appropriate steps. Note that there is some overlap with those defined for the Enter Cabin use case.

The Exit Cabin use case has five extension steps. The first one (2a) is an alternative to step 2. It is taken if the request made by the user is the same floor as the one where the cabin is currently stationed. The second extension step (2-6 ||a) takes place if the user makes an additional request to the system in parallel to steps 2, 3, 4, 5 or 6. In this case, the system schedules the request and continues the main scenario normally. The third extension step (3a) is an alternative to step 3. It is taken if the system decides that another request has priority and it serves that one first (i.e., it does not serve the user's request straightaway). In this case, the system schedules the request for later. This extension step rejoins the main scenario at step 3. The fourth extension step (3b) is taken if the system is already busy with another request (e.g., the user was too slow in making a request). In this case, the system schedules the request for later. This extension step rejoins the main scenario at step 3, where the request will eventually be served by the system, according to the main scenario. The fifth extension step (7a) is an alternative to step 7. It is taken if the user previously chose to go to another (additional) destination. In this case, the use case continues at step 2.

As highlighted by the third note in the Notes clause of the Exit Cabin use case, the system can not know which requests are made by which User actors. This point is worth mentioning, however, because it may make the stakeholders reconsider their requirements, e.g., should a user be allowed to cancel requests, would it be interesting to have "personalized" requests for VIP riders, etc.

1. Note that it would have been possible to group these steps into three sub-goals of the user: user makes request, user waits for floor and user leaves cabin, and describe use cases for each one. This approach has not been taken for space reasons, and because it is not always profitable to have too many small use cases.

Use Case: Exit Cabin
System: ElevatorControl
Level: Sub-Function
Intention in Context: The User intents to leave the elevator cabin at a certain floor.
Primary Actor: User
Main Success Scenario:
Steps 1 and 2 can happen in any order
1. User makes a request to System to go to a floor
2. Door times-out on open period and closes itself, and informs System that it is closed
3. System requests Motor to move towards destination floor; Motor goes in direction of destination floor

Step 4 is repeated until System determines that the destination floor of the User has been reached

4. Sensor informs System that cabin has reached a certain floor
5. System requests Motor to stop cabin; Motor stops cabin and informs System that it is stopped;
6. System requests Door to open, and informs User that request has been fulfilled; Door opens
7. User exits cabin at destination floor

Extensions:
2a. System determines that cabin is already at requested floor; use case continues at step 6[1].
(2-6)||a. User requests System to go to a different floor[2]:
  (2-6)||a.1. System schedules the request; use case joins main scenario.
3a. System decides to serve another request:
  3a.1. System schedules the request; use case continues at step 3 (when System decides to serve this request).
3b. System is serving another request (step 2 occurred before step 1):
  3b.1. System schedules the request; use case continues at step 3 (when System decides to serve this request).
7a. User has additional requests pending[3]: use case continues at step 2.
Timing Requirements:
Step 4: “It takes at least 1.5 seconds and at most 2 seconds to go from one floor to another one”.
Step 5: “the system must decide within 0.1 seconds if it is to stop the cabin”.
Step 6: “The door is guaranteed to stay open for at least 5 seconds”.
Notes:
[1] It is possible that the door may already be open (assumption: requesting door to open while it is already open will make it stay open for a new time period).
[2] There is no canceling of requests.
[3] System cannot make a difference between a single user making many requests and many users each making a request.

Figure 9.3: Exit Cabin use case for ElevatorControl system

9.3 ANZAC Specification

In this section, I describe the ANZAC specification for the ElevatorControl system. In section 9.3.1, I present the Concept Model for the ElevatorControl system. From thereon, I describe the steps taken to derive and refine the ANZAC specification from the ANZAC use case descriptions for the ElevatorControl system. These steps follow the ANZAC mapping technique described in Chapter 8.

9.3.1 Concept Model

In this subsection, I describe the Concept Model for the ElevatorControl system. Note that the initial construction of the Concept Model is not part of the ANZAC mapping technique, because the Concept Model is not directly derived from the ANZAC use case descriptions. Instead, an initial Concept Model is usually derived from a description of the concepts of the application domain[1]. Note that in this thesis I do not address this aspect. An example of a domain model that was used to derive a “first-cut” of the Concept Model for the ElevatorControl system can be found in [Se02b].

As part of the mapping activity, the “first-cut” of the Concept Model is refined and changed as the Operation Model is developed. Entities and relations, which represent concepts of the problem domain, are introduced on a “need to know” basis, which means that each entity/relation must be justified from the perspective that it allows the system to retain the essential information that it needs to meet its responsibilities. Hence, the Concept Model is tightly coupled to the Operation Model. In fact, the Concept Model can not be considered stable until all functionality (of the increment) has been investigated with the Operation Model.

The Concept Model that I present in this subsection shows the synthesis of all the modifications made during the development of the Operation Model (section 9.3.5). Therefore it can be seen as a result of step 4 of the ANZAC mapping technique applied to the ElevatorControl system. I show it as a precursor to the application of the ANZAC mapping technique, however, because it is used in the explanation of the Protocol and Operation Models.

The Concept Model for the ElevatorControl system is shown in figure 9.4. It defines the (abstract) state space of the ElevatorControl system in the form of classes, associations between these classes, constraints on the possible number of instances of classes and associations, and other invariants. Inside the system, there are three (normal) classes, Cabin, Floor, and Request, and five <<rep>> classes, Motor, Door, *ReqSource*, IntReqSource, ExtReqSource. *ReqSource* is an abstract class, and Cabin, Motor, Door, IntReqSource and ExtReqSource have only a single object in the system (which matches the fact that the system only interacts with a single instance of these actors). Cabin has two attributes: movement, which indicates whether the cabin is going up, down, stopping, or stopped; and doorState, which indicates whether the cabin door is open or closed. Floor has a single (frozen) attribute called num, which indicates the level in the building within which it is contained[2].

The system has five (non-composition) associations: IsFoundAt links a cabin to its current floor; HasCurrentRequest links the cabin to the current request that it is serving, if there is one; HasTargetFloor links request objects to their (frozen) target floor (source or destina-

1. Such a description of the domain is also important to set a common vocabulary across the project.
2. Note that for convenience, integer was chosen as the type of num. However, this may need to be made more general to cover more “exotic” numbering schemes.

tion); and HasSource links a request object to the (frozen) actor rep object that represents the actor that made the request.

The system has two invariants: noMovingWithDoorOpen, which states that if the cabin door is open then the lift must be stopped and if the cabin is moving then the door is closed; and strictlyIncreasingFloorNumbers, which states that for each floor there is exactly one floor above it and one floor below it, unless it is the top or bottom floor, respectively. These two invariants, written in OCL, are shown below.

```
context Cabin inv noMovingWithDoorOpen:
self.doorState = DoorState::opened implies self.movement = Movement::stopped &
self.movement <> Movement::stopped implies self.doorState = DoorState::closed;

context ElevatorControl inv strictlyIncreasingFloorNumbers:
self.floor->forall (f1 | -- for each floor in the system
   (self.floor->one (f2 | f2.num = f1.num + 1) or self.floor.num->max () = f1.num)
   -- there exists exactly one floor higher than it or it is the top floor
   and (self.floor->one (f2 | f2.num = f1.num - 1) or self.floor.num->min () = f1.num)
   -- and there exists exactly one floor lower than it or it is the bottom floor
   ) &
self.floor->forall (x, y | x.num = y.num implies x = y); -- all numbers are unique
```

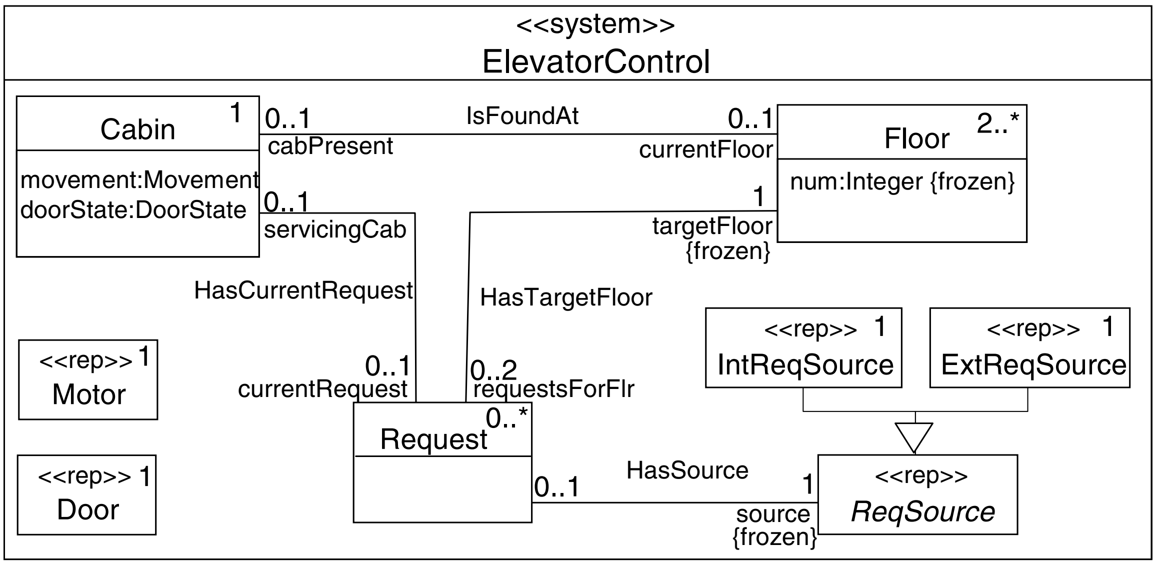

Figure 9.4: Concept Model for ElevatorControl system

It is important to note that the classes and associations specified by the Concept Model are not the only possible choice—there are in fact many possible ways to represent the state space of the ElevatorControl system. I now highlight some of the reasoning behind the choices taken.

I chose to represent floors and requests as objects. In the case of floors, an alternative would have been to represent a floor just as a number. In this case, the associations IsFoundAt and HasTargetFloor would not be needed, and the classes Cabin and Request would contain an additional attribute, which would represent the current floor and the target floor, respectively. The reason for representing a floor as an object was because it proves to be easier to define constraints—they can be defined by navigation in OCL and not by calculation, which would require additional (special-purpose) functions to be defined—and because it is conceptually cleaner to use objects and more visually appealing (since optimization is not an issue for this description).

In the case of requests, clearly the notion of request is needed by the ElevatorControl system, because the system must ensure that eventually services all of them—a user request can take significant time to service since they are only served one at a time, in most cases, and the cabin takes time to go from one floor to another. Alternatives to a class would have been to use an association class between Floor and ReqSource, which would itself have an association with Cabin (to represent the current request); or to define two attributes of cabin, one that contains a set of tuples with (floor number, instance of ReqSource), which represents all requests, and one that contains a single tuple with (floor number, instance of ReqSource), which represents the current request. The first option would be a possibility, although I prefer to avoid association classes when possible and it would not be as easy in general to maintain; this issue is discussed in the next paragraph. And, the second option is quite complicated, because the set of all request tuples and the current request tuple would need to be kept consistent, and it would be necessary to manipulate tuples, which would exclude the use of the current version of OCL (v1.4).

Using an explicit class Request has advantages if it became necessary to differentiate external requests from internal requests. For instance, if external requests were extended with the notion of direction, then only two subclasses would need to be added to the current model, one for internal requests and one for external requests. However, in the case of either alternative, it would require quite some changes, and the effect to the OCL constraints probably would require quite some rework.

9.3.2 Applying Step 1 of the ANZAC Mapping Technique

In this subsection, following step 1 of the ANZAC mapping technique, I describe how the messages exchanged between the system and the actors in its environment are identified from the ANZAC use case descriptions (given in section 9.2). Note that neither the Enter Cabin use case nor the Exit Cabin use case indicate any system-instigated actions; hence no internal triggers are provided for the ElevatorControl system.

The Enter Cabin and Exit Cabin use cases are used for extracting candidate input and output messages. Starting with the Enter Cabin use case, I analyze its main scenario for candidate communications that involve the system, either as sender or receiver.

- *Main scenario step 1*: The user makes a request to the system for the elevator cabin; this suggests an input message from a User actor. However, this use case abstracted away the two request intermediaries, one for all external requests and one for all internal requests, which intercept the requests from users and pass them onto the system (according to the problem statement). I refer to the first one as ExtReqSource and the second one as IntReqSource (they have facilitator personalities). The system (directly) interacts with these actors and not users. Therefore, according to the ANZAC specification, a request is defined as coming from one of these two actors.

  The input message from the ExtReqSource actor is given the following signature.

  ```
  ExternalRequest (f: Floor);
  ```

  The message takes a floor object as parameter, which indicates the source floor of the request. A floor object is passed rather than a (floor) number or some other identifier standing for a particular floor to ensure data abstraction.
- *Main scenario step 2*: The system sends out a request to the motor to go in a certain direction; this suggests a message being sent to the (singleton) Motor actor.

  The messages sent to the Motor actor, each indicating a possible direction to travel, are given the following signatures.

  ```
  GoDown ();
  GoUp ();
  ```

  Two different, parameter-less, messages have been defined rather than a single message with a parameter that would be used to differentiate the commands to the Motor. The reason for two versus one is not fundamental, but rather a matter of style: two messages with informative names are probably clearer than a single generic message (as long as there are not too many). Note that this style choice does not constrain the actual interaction protocol to follow suit, but only that it is a refinement of the (abstract) communication protocol defined by an ANZAC specification.
- *Main scenario step 3*: The sensor notifies the system that the cabin is arriving at a floor; this suggests an input message from a Sensor actor and the following signature is proposed.

  ```
  AtFloor (f: Floor);
  ```

  The message has a floor object as parameter, which indicates the floor at which the cabin is arriving.
- *Main scenario step 4*: The system sends out a request to the motor to stop and the motor responds by informing the system that it is stopped; this suggests a message being sent to the Motor actor and the following signature would seem appropriate,

  ```
  StopAtNextFloor ();
  ```

  and a message from the Motor actor with the following signature.

  ```
  CabinIsStopped ();
  ```

  Both messages are without parameters.

- *Main scenario step 5*: The system sends out a request to the door to open and informs the user that his/her request has been serviced; this suggests a message being sent to the Door actor and the following signature would seem appropriate,

  ```
  OpenDoor ();
  ```

  and a message sent to the ExtReqSource actor with the following signature.

  ```
  ServicedRequest (servicedFlr: Floor);
  ```

  The first message is without parameters. The second message has a floor object as parameter, which indicates the requested floor that was serviced.
- *Main scenario step 6*: There are no messages involving the system.

Now analyzing the extension steps of the Enter Cabin use case.

- *Extensions steps 2a and 2a.1, 2b, (2-6)a*: There are no messages involving the system.

In total, three input messages and five output messages were found. It is possible to show the mapping decisions directly on the corresponding use case. The annotated Enter Cabin use case is shown in figure 9.5. Note that the square bracket fields contain the defined messages and indicate the mode of the messages, i.e. *out* implies output message and *in* implies input message (relative to the system).

Use Case: Enter Cabin
Main Success Scenario:
1. User makes a request to System for cabin **[***in* **->** ExternalRequest**]**
2. System requests Motor to move towards source floor **[***out* **->** GoDown, GoUp**]**;
   Motor goes in direction of source floor

Step 3 is repeated until System determines that the source floor of the User has been reached

3. Sensor informs System that cabin has reached a certain floor **[***in* **->** AtFloor**]**
4. System requests Motor to stop cabin **[***out* **->** StopAtNextFloor**]**;
   Motor stops cabin and informs System that it is stopped **[***in* **->** CabinIsStopped**]**;
5. System requests Door to open **[***out* **->** OpenDoor**]**,
   and informs User that request has been fulfilled **[***out* **->** ServicedRequest**]**;
   Door opens
6. User enters cabin at source floor

Extensions:
2a. System is serving another request:
  2a.1. System schedules the request; use case continues at step 2 (when System decides to serve this request).
2b. System determines that cabin is already at requested floor; use case continues at step 5.
(2-6)a. User walks away; use case ends in failure

Figure 9.5: Annotated Enter Cabin use case for ElevatorControl system

Continuing now with the Exit Cabin use case, I analyze its main scenario for candidate communications that involve the system, either as sender or receiver.

- *Main scenario step 1*: The user makes a request to the system from inside the elevator; this suggests an input message from the IntReqSource actor and the following signature would seem appropriate.

```
InternalRequest (f: Floor);
```

  The message has a floor object as parameter, which indicates the destination floor of the User.
- *Main scenario step 2*: The door notifies the system of its closing; this suggests an input message from the Door actor and the following signature would seem appropriate.

  ```
  DoorIsClosed ();
  ```

  The message is without parameters.
- *Main scenario steps 3, 4, and 5*: These steps are identical to steps 2, 3, and 4, respectively, of the Enter Cabin use case and therefore the candidate messages have already been found.
- *Main scenario step 6*: The system sends out a request to the door to open and informs the user that his/her request has been serviced; the message for opening the door has already been defined by step 5 of the Enter Cabin use case, but the message for informing the user that his/her request has been serviced is sent to the IntReqSource actor (rather than the ExtReqSource actor as in step 5 of the Enter Cabin use case). At this stage, there is no conceptual reason or requirement (given by section 9.1.1) that would stop one from using the same message (type) to inform the user that his/her request has been serviced, both in the case of internal and external requests. Thus, no new message is defined.
- *Main scenario step 7*: There are no messages involving the system.

Now moving onto the extension steps.

- *Extensions steps 2a*: There are no messages involving the system.
- *Extensions steps (2-6)||a and (2-6)||a.1*: The user makes a request to the system from inside the elevator; this is the InternalRequest message captured in the analysis of step 1.
- *Extensions steps 3a and 3a.1, 3b and 3b.1, 7a*: There are no messages involving the system.

In total, two input messages and no output messages were found in addition to those found for the Enter Cabin use case.

As shown earlier for the Enter Cabin use case, it is possible to show the mapping decisions directly on the use case. The annotated Exit Cabin use case is shown in figure 9.6.

Use Case: Exit Cabin
Main Success Scenario:
Steps 1 and 2 can happen in any order
1. User makes a request to System to go to a floor **[***in* **->** InternalRequest**]**
2. Door times-out on open period and closes itself, and informs System that it is closed **[***in* **->** DoorIsClosed**]**
3. System requests Motor to move towards destination floor **[***out* **->** GoDown, GoUp**]**;
   Motor goes in direction of destination floor

Step 4 is repeated until System determines that the destination floor of the User has been reached
4. Sensor informs System that cabin has reached a certain floor **[***in* **->** AtFloor**]**
5. System requests Motor to stop cabin **[***out* **->** StopAtNextFloor**]**;
   Motor stops cabin and informs System that it is stopped **[***in* **->** CabinIsStopped**]**;
6. System requests Door to open **[***out* **->** OpenDoor**]**,
   and informs User that request has been fulfilled **[***out* **->** ServicedRequest**]**;
   Door opens
7. User exits cabin at destination floor

Extensions:
2a. System determines that cabin is already at requested floor; use case continues at step 6.
(2-6)||a. User requests System to go to a different floor **[***in* **->** InternalRequest**]**:
   (2-6)||a.1. System schedules the request; use case joins main scenario.
3a. System decides to serve another request:
   3a.1. System schedules the request; use case continues at step 3 (when System decides to serve this request).
3b. System is serving another request (step 2 occurred before step 1):
   3b.1. System schedules the request; use case continues at step 3 (when System decides to serve this request).
7a. User has additional requests pending: use case continues at step 2.

Figure 9.6: Annotated Exit Cabin use case for ElevatorControl system

Bringing together the input and output messages that were identified from the two use cases, there are five input messages and five output messages. Figure 9.7 shows all the input and output messages, defined above, with their corresponding actor (destination or source). Note that the ExtReqSource and IntReqSource actors have been generalized by the ReqSource actor. This conceptually allows one to send a ServicedRequest to a single actor (kind).

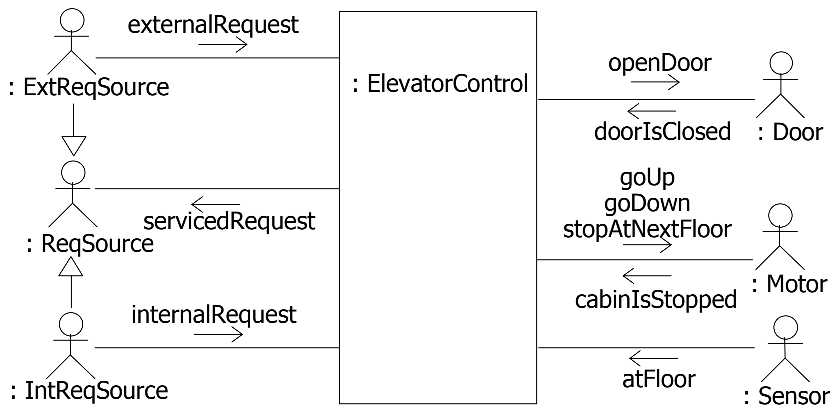

Figure 9.7: Overview of all input and output messages for ElevatorControl System

9.3.3 Applying Step 2 of the ANZAC Mapping Technique

In this subsection, following step 2 of the ANZAC mapping technique, I collect the system's responsibilities with respect to each of the five input messages (found during step 1, section 9.3.2). These responsibilities are simply noted as cross-references to the relevant steps of use case descriptions.

The result of performing step 2 of the ANZAC mapping technique on the ElevatorControl system is shown below.

```
ExternalRequest (f: Floor)
-- Enter Lift: Steps 2, 2a, 2b
```

The ExternalRequest message invokes actions by the system that are described in steps 2, 2a, and 2b of the Enter Lift use case.

```
InternalRequest (f: Floor)
-- Exit Lift: Steps 3, 2a, (2-6)||a, 3a, 3b
```

The InternalRequest message invokes actions by the system that are described in steps 3, 2a, (2-6)||a, 3a and 3b of the Exit Lift use case.

```
AtFloor (f: Floor)
-- Enter Lift: Steps 3, 4
-- Exit Lift: Steps 4, 5
```

The AtFloor message invokes actions by the system that are described in steps 3 and 4 of the Enter Lift use case and steps 4 and 5 of the Exit Lift use case.

```
DoorIsClosed ()
-- Exit Lift: Steps 3, 3a
```

The DoorIsClosed message invokes actions by the system that are described in steps 3 and 3a of the Exit Lift use case.

```
CabinIsStopped ()
-- Enter Lift: Steps 5
-- Exit Lift: Steps 6
```

The CabinIsStopped message invokes actions by the system that are described in step 5 of the Enter Lift use case and step 6 of the Exit Lift use case.

9.3.4 Applying Step 3 of the ANZAC Mapping Technique

In this subsection, following step 3 of the ANZAC mapping technique, the Protocol Model for the ElevatorControl system is established. In particular, I define the input protocol of the system, given the input messages and the related responsibilities of the system identified in section 9.3.3.

In this subsection, I make the assumption that all operations execute instantaneously (i.e., in zero time), which means that only one operation can ever execute at any one time, and hence, there can be no interference between operations. The reason I take this approach is because I want to show that the instantaneous-operation assumption can also effect the validity of the Protocol Model, and not just the Operation Model. A version of the Protocol Model that does not assume operations are instantaneous is given in section 9.4. Also, I defer the specification of timing constraints until section 9.4 as well (since it seems pointless to describe timing constraints on instantaneous operations).

Following step 3 of the ANZAC mapping technique, each input message is mapped to an event with the same signature[1]. Signal events are used rather than call events, because all messages are asynchronously sent by actors (i.e., the calling actors do not wait). The resulting event definitions are not shown here, due to their similarity to the message definitions.

The divide-by-actor technique is applied to find the input protocol that the system has with each actor. The actor-activity-state for the ExtReqSource, IntReqSource, Door, Motor and Sensor actors are shown in figure 9.8. Each actor-activity-state only involves a single (self) transition, which is quite uninformative as to the input protocol that the system has with the actors.

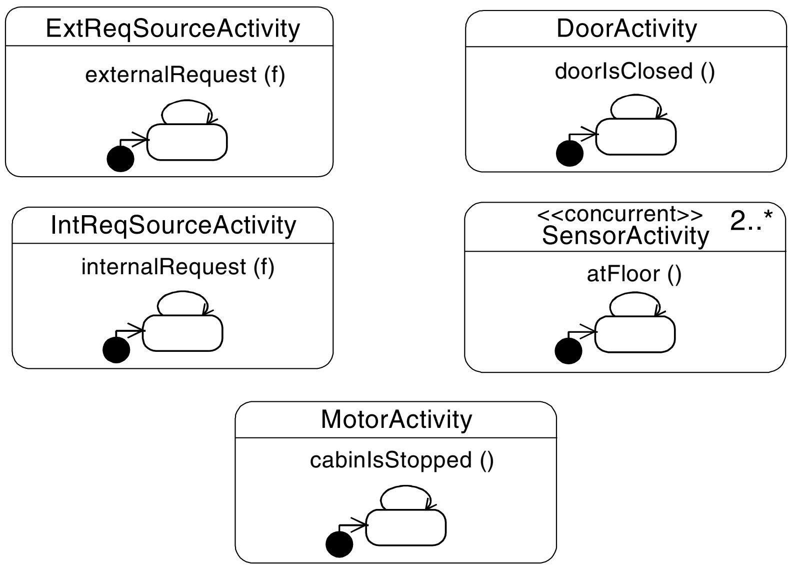

Figure 9.8: Activities for each actor of the ElevatorControl system

Applying the divide-by-collaboration technique, it is necessary to look for collaboration activities that the system is involved in with the actors. According to this technique, each collaboration activity that is found is described using a view-state.

Probably the most intuitive choice for a collaboration activity would be the one that surrounds the movement of the cabin and the opening of the door, since this activity is what drives the interaction of the system with the Motor, Sensors, and Door. To understand the input protocol associated to these interactions, it is useful to revisit the Enter Cabin and Exit Cabin use cases. I describe the protocols that can be extracted from these two use cases and

1. Recall that on reception of a message, the system generates an event with the same signature as the message.

express them in terms of the effects that the set of scenarios, described by the use cases, have on the state of the cabin, both in terms of the change to the state of the cabin's movement and the state of the cabin's door.

View of Enter Cabin use case

Analyzing the (annotated) Enter Cabin use case (shown in figure 9.5), the use case starts by the user making a request to the system for the cabin (step 1). This request is represented by an externalRequest event. In processing this externalRequest event, the system deals with it in one of three ways.

- The system is able to serve the request directly; so it requests the motor to move the cabin towards the source floor (step 2).
- The system is able to serve the request directly but the request is at the same floor; so the system requests the door to open (step 2b).
- The system is serving another request; so the request is scheduled to be served later (step 2a).

If one was to describe this behavior in terms of changes in state to the cabin movement and door state, then the first situation would cause a transition from the "cabin is stopped" state to the "cabin is moving" state and door would stay in closed state; the second situation would cause a transition from the "door is closed" state to the "door is open" state or no transition (since door is already open), and no change to the movement of the cabin; and the third situation would not cause a change in the movement of the cabin.

Given the first situation (following the main scenario), where the cabin is set into motion, the system will receive an atFloor event as the cabin arrives at each floor (step 3). For each atFloor event received, the system deals with them in one of two ways.

- The system determines that the source floor is not the one arriving; so the system lets the cabin continue on.
- The system determines that the source floor is the one arriving; so the system requests the motor to stop (step 4).

In terms of state transitions: the first situation would mean that the cabin stays in the "cabin is moving & door is closed" state; and the second situation would cause a transition from the "cabin is moving" state to the "cabin is stopping" state and no change to the door state.

Given the second case, where the cabin is stopping, the system will receive a cabinIsStopped event from the motor when the cabin reaches the corresponding floor. This event would cause a transition from the "cabin is stopping" state to the "cabin is stopped" state. Furthermore, the system will request the door to open causing a transition from the "door is closed" state to the "door is open" state and inform the user that the request has been fulfilled (step 5), and the user will enter the cabin (step 6). Neither of these circumstances cause a change in the state of the cabin.

The view-state that describes the input protocol of the system from the perspective of the Enter Cabin use case is shown in figure 9.9. The EnterCabinUCView state has three orthogonal regions. The externalRequest transition is modeled in an orthogonal region, because the user can make his/her request when the cabin and door are in any state. The door state and the cabin movement are modelled as orthogonal regions because they address two different concepts. The six transitions shown in the EnterCabinUCView state include three synchronization transitions and two guards that refer to the state of the other region. Note that the EnterCabinUCView state is only a partial activity because it refers to strictly only the information that was provided by the use case, e.g., there is no transition from Open to Closed.

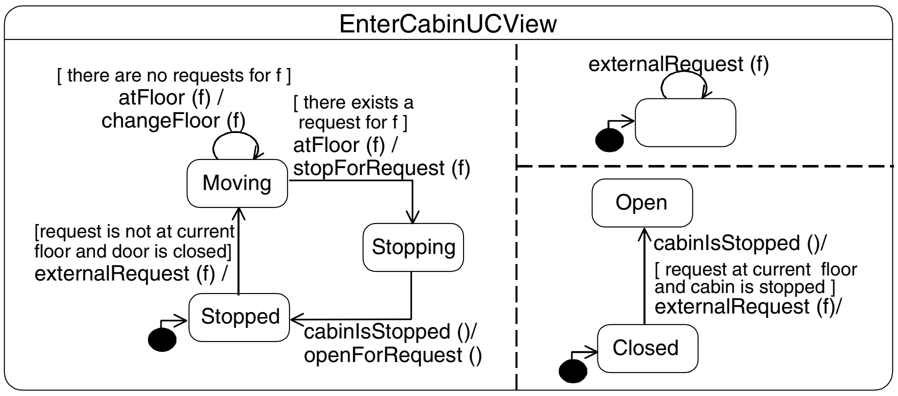

Figure 9.9: View of protocol described by Enter Cabin use case

The choice of operations shown on the transitions of the EnterCabinUCView state reflect the responsibilities that the system has in those situations. Note that the single label on the externalRequest transition refers to both the event and operation name. Also, recall that all operations are executed atomically (and instantaneously) with respect to others.

View of Exit Cabin use case

The view-state that describes the input protocol of the system from the perspective of the Exit Cabin use case is shown in figure 9.10. It is similar to the EnterCabinUCView. There are two points that are noteworthy.

First, the internalRequest transitions matches closely the externalRequest transition of EnterCabinUCView, except, in addition to being orthogonal, it is concurrent to the other two regions. The reason that it is specified as a concurrent region is because the use case states that a user can make multiple requests (step (2-6)||a), where the user can act concurrently to the Door, Sensors and Motor (following the inherent concurrency guideline).

Second, two transitions from Open to Closed and from Stopped to Moving are in addition to those shown for EnterCabinUCView. These transitions take into account the doorIsClosed event, which is derived from step 2. The two transitions cover the following three cases:

- The door closes after the user issues a floor request: the transitions to Moving and Closed are taken, either to service the request of the user or another user (step 3 or 3a).
- The door closes before the user issues a floor request (internalRequest event) and there are no requests pending: only the transition to Closed is taken (step 2 where it occurs before step 1). If the elevator is still idle when the user makes the request, then the request will be served directly (step 3).
- The door closes before the user issues a floor request and there are requests pending: the transitions to Moving and Closed are taken to service the request of another user (step 2 where it occurs before step 1). This means that when the user makes the request it will be scheduled (step 3b).

Note that these three cases highlight that the doorIsClosed event is used to trigger the system into serving the next request, which covers the cases when there are pending requests and there are none. The choice of relating this responsibility to the reception of the doorIsClosed event was a design decision. Such a decision was required for the description to be functionally complete. The choice matches naturally the reactive nature of the system, i.e., actions are triggered by stimuli from the system's environment. The alternative choice would have been to periodically check if the cabin is in an idle state, and if so, trigger it into serving any pending requests. However, this second choice would create additional overheads (e.g., a timer activity, checks made periodically that may often be unnecessary, etc.), and there was certainly no explicit requirement for such a mechanism (according to the use case descriptions and the problem statement).

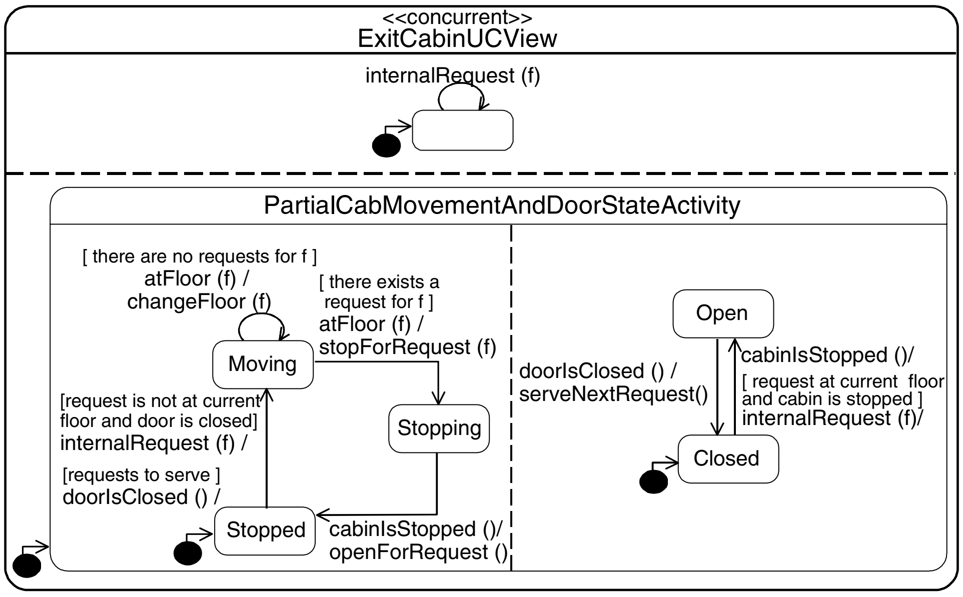

Figure 9.10: View of protocol described by Exit Cabin use case

View of Cabin Movement and Door State

The complete view of the cabin movement and door state can be constructed by composing the partial views given by EnterCabinUCView (figure 9.9) and ExitCabinUCView (figure 9.10). However, it is important to verify that the combination of these two views cover all cases of the collaboration, due to the incomplete nature of use cases. In this regard, there is one case that is not mentioned by either of the two use cases that relates to the state of the cabin: after the user exits the cabin, the door closes; at this point, the elevator may simply become idle, having no pending requests. This case, however, is equivalent to the case where the door closes before the user issues a floor request and there are no requests pending. Thus, it is covered in ExitCabinUCView by the transition from Open to Closed.

Figure 9.11 shows the CabMovementAndDoorStateView, which represents the complete view of the cabin movement and door state. The externalRequest and internalRequest regions have been factored out into separate concurrent state (a different part of the Protocol Model), since externalRequest and internalRequest events can be received in any state of CabMovementAndDoorStateView, but more importantly because they represent requests from sources that operate concurrently to this activity (ExtReqSource and IntReqSource), following the divide-by-actor technique.

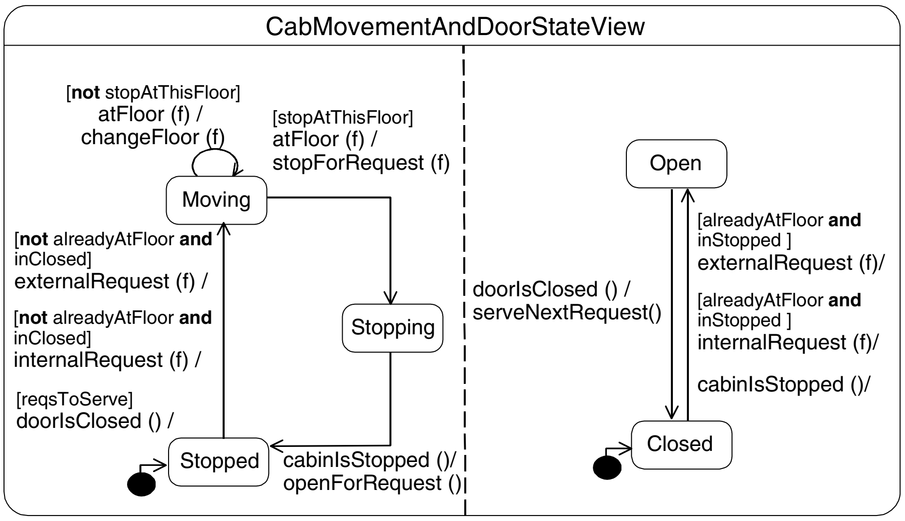

Figure 9.11: CabMovementAndDoorStateView (Version 1)

The guards of CabMovementAndDoorStateView are expressed in OCL. The declaration of the expression-substitutions that are used in the guards are given below.

```
Aliases:
stopAtThisFloor: Boolean Is self.makeStopAccordingToPolicy_f (self.cabin, f);
reqsToServe: Boolean Is self.request->notEmpty ();
alreadyAtFloor: Boolean Is self.cabin.currentFloor = f;
inClosed: Boolean Is self.oclInState (Closed);
inStopped: Boolean Is self.oclInState (Stopped);
```

The expressions are defined in terms of the Concept Model (section 9.3.1) and the CabMovementAndDoorStateView state itself. All five expressions are defined in the context of the system, i.e., self is equivalent to the system, and the first and third expressions make use of a floor object f, which is a parameter of the two corresponding events with which the expressions are guards.

The boolean expression stopAtThisFloor evaluates to true if there are requests that should be serviced at floor f, according to the current stopping policy of the elevator, false otherwise. The application of the stopping policy is encapsulated by the Function makeStopAccordingToPolicy_f, which is defined in Appendix C. The boolean expression reqsToServe evaluates to true if the system has pending requests, otherwise it is false. The boolean expression alreadyAtFloor evaluates to true if the cabin is currently at the same floor as the floor parameter f, otherwise it is false. The boolean expression inClosed evaluates to true if the ElevatorControlActivity state machine is in the Closed state (CabMovementAndDoorStateView), otherwise it is false. The boolean expression inStopped evaluates to true if the ElevatorControlActivity state machine is in the Stopped state (CabMovementAndDoorStateView), otherwise it is false.

Protocol Model for the ElevatorControl system

The Protocol Model for the ElevatorControl system is shown in figure 9.12. It consists of three concurrent states: ExtReqSourceActivity, IntReqSourceActivity and CabinAndDoorMovementView.

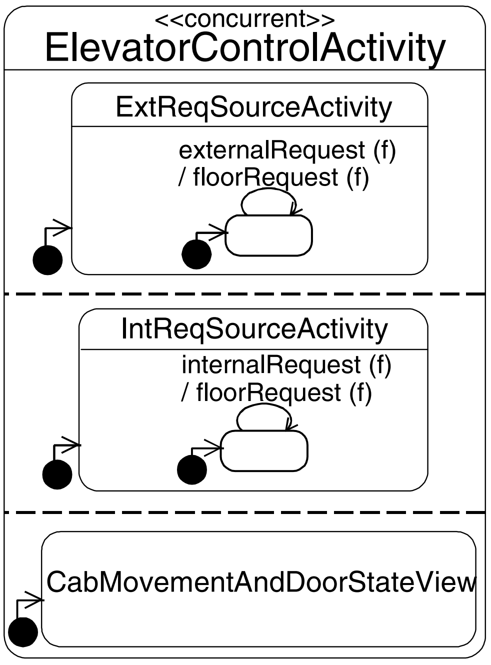

Figure 9.12: ElevatorControl's Protocol Model

The ExtReqSourceActivity state models the input protocol that the system has with the actor ExtReqSource, i.e., any number of external request messages can be received over time. The ExtReqSourceActivity state is concurrent to the IntReqSourceActivity and CabinAndDoorMovementViews. The IntReqSourceActivity state is the internal request equivalent to ExtReqSourceActivity. It is concurrent to the ExtReqSourceActivity and CabinAndDoorMovementViews. Note that both ExtReqSourceActivity and IntReqSourceActivity are actor-activity-states that are a result of the divide-by-actor technique. The floorRequest operation generalizes both the externalRequest and internalRequest operations defined by the EnterCabinUCView and ExitCabinUCView views, because it was judged that they could be both served by the same operation (if the two operations were generalized).

The CabMovementAndDoorStateView models the input protocol that relates to the cabin movement and door state—it is shown in full in figure 9.11. Note that this view-state was provided as an alternative to the actor-activity-states for the Door, the Motor and the Sensors.

Operation signatures and responsibilities

The operations that are associated to each transition have the following signatures and responsibility allocations:

```
floorRequest (f: Floor)
-- Enter Lift: Steps 2, 2a, 2b
-- Exit Lift: Steps 3, 2a, (2-6)||a, 3a, 3b
```

The floorRequest operation is described by steps 2, 2a, and 2b of the Enter Lift use case and steps 3, 2a, (2-6)||a, 3a and 3b of the Exit Lift use case. Note that these responsibilities are the combination of those found for the ExternalRequest and InternalRequest messages.

```
changeFloor (f: Floor)
-- Enter Lift: Steps 3
-- Exit Lift: Steps 4
```

The changeFloor operation is described in step 3 of the Enter Lift use case and step 4 of the Exit Lift use case.

```
stopForRequest (f: Floor)
-- Enter Lift: Steps 4
-- Exit Lift: Steps 5
```

The stopForRequest operation is described in step 4 of the Enter Lift use case and step 5 of the Exit Lift use case. Note that the stopForRequest and changeFloor operations together account for the responsibilities related to the AtFloor message.

```
serveNextRequest ()
-- Exit Lift: Steps 3, 3a
```

The serveNextRequest operation is described in steps 3 and 3a of the Exit Lift use case. This operation accounts for the responsibilities related to the DoorIsClosed message.

```
openForRequest ()
-- Enter Lift: Steps 5
```

-- Exit Lift: Steps 6

The openForRequest operation is described by step 5 of the Enter Lift use case and step 6 of the Exit Lift use case. This operation accounts for the responsibilities related to the CabinIsStopped message.

9.3.5 Applying Step 4 of the ANZAC Mapping Technique

In this subsection, following step 4 of the ANZAC mapping technique, I formalize the system's responsibilities by developing the Operation Model for the ElevatorControl system. Step 4 offers two approaches for describing concurrent operations: the "all at once" approach and the "atomic/non-atomic split" approach. The second approach is taken in this chapter, because it makes it easier to present the Operation Schemas, and it allows one to make a comparison between the sequential and concurrent version.

As a consequence of the "atomic/non-atomic split" approach, in this subsection I develop the Operation Schemas for operations that are assumed to execute atomically with respect to other operations. The Operation Schemas for the "non-atomic" versions are given in section 9.4 (i.e., the "non-atomic" part of step 4).

The Operation Model for the ElevatorControl system is described by five Operation Schemas for the operations taken from section 9.3.4 (step 3 of the ANZAC mapping technique): floorRequest, changeFloor, stopForRequest, openForRequest and serveNextRequest. The circumstances under which these operations are triggered is defined by the Protocol Model. The responsibilities of each operation are derived, and refined, from the steps of the (annotated) Enter Cabin and Exit Cabin use cases.

I now describe each Operation Schema for these five operations, and I discuss how they relate to the ANZAC use cases, the Concept Model, and the Protocol Model.

floorRequest

The floorRequest operation is called whenever an internal or external request is made on the system by the ExtReqSource or IntReqSource actor, respectively. The responsibilities of this operation are derived from steps 2, 2a and 2b of the Enter Cabin use case and from steps 3, 2a together with 6, (2-6)||a, 3a and 3b of the Exit Cabin use case. The floorRequest operation is responsible for deciding whether to reject the request, serve the request by opening the door, serve the request by sending the cabin in the direction of the request, or store the request for later. The Operation Schema for the floorRequest operation is shown in figure 9.13.

The Messages clause indicates that the actors: Motor, Door and ReqSource, may be sent messages. According to the generalization relationship from ReqSource to ExtReqSource and IntReqSource (shown in figure 9.7), both the IntReqSource and ExtReqSource actors can receive the messages that ReqSource can accept.

The New clause declares a new object and four message instances. The new object newReq becomes part of the system state population if the request can not be served immediately, so as to ensure that it will not be forgotten.

The Aliases clause provides four expression-substitution declarations: cab, notAlreadyRequested, alreadyAtFloorAndStopped, and elevatorIsIdle. cab is an expression that denotes a reference to the only cabin object in the system. notAlreadyRequested is a boolean expression that results in true if the given floor has already been requested by the client (denoted by the pre-defined identifier sender in the expression). Note that the client is either the ExtReqSource or IntReqSource actor. alreadyAtFloorAndStopped is a boolean expression that results in true, if the cabin is at the same floor as the one given as parameter, and the cabin is stopped. elevatorIsIdle is a boolean expression that results in true if the cabin is stopped and the door is closed.

```
Operation: ElevatorControl::floorRequest (f: Floor);
Description: A floor request (external or internal) is made to system, where it must decide to
reject it, serve it (open door, or send cabin to floor), or store it for later;
Use Cases: EnterCabin::{2; 2a; 2b;}; ExitCabin::{3; 2a; 6; (2-6)||a; 3a; 3b;};
Scope: Cabin; Floor; Request; HasTargetFloor; HasCurrentRequest; IsFoundAt; HasSource;
Messages: Motor::{GoUp; GoDown;}; Door::{OpenDoor;}; ReqSource::{ServicedRequest;};
New: newReq: Request; up: GoUp; down: GoDown; open: OpenDoor;
servicedReq: ServicedRequest;
Aliases:
  cab: Cabin Is self.cabin;
  notAlreadyRequested: Boolean Is f.requestsForFlr@pre.source->excludes (sender);
  alreadyAtFloorAndStopped: Boolean Is
    cab.currentFloor@pre = f and cab.movement@pre = Movement::stopped;
  elevatorIsIdle: Boolean Is cab.movement@pre = Movement::stopped and
    cab.doorState@pre = DoorState::closed;
Pre: true;
Post:
  if notAlreadyRequested then
    if alreadyAtFloorAndStopped then
      cab.doorState = DoorState::open &
      requestDoorToOpen_p (open) & -- request door to open even if it is already open
      informSourceThatReqServiced_p (sender, serviceReq, f)
    elsif elevatorIsIdle then
      storeRequest_p (f, newReq, sender) & -- request is persisted
      serveRequest_p (cab, f, newReq, up, down) --req is made current, cab sent in its direction
    else
      storeRequest_p (f, newReq, sender)
    endif
  endif;
```

Figure 9.13: Operation Schema for floorRequest

The Pre clause is true and thus there is no precondition on the operation, i.e., the operation can make not assumptions about the state of the system or parameters before it is executed (other than the invariants and well-formedness rules on parameters).

The Post clause asserts that in the case that the requester has already made the same request then the operation did nothing. Otherwise, if the cabin is already at the requested floor and it is not moving, then the system requested the door to open and informed the requester that its request has been served. Otherwise, if the elevator was idle then the system scheduled the request (stored it) and started to serve it by sending the cabin in the direction of the request. Otherwise, the system scheduled the request (to serve later). Note that this operation only has the right to serve the request if the cabin is idle (stopped and door closed). This means that even if the elevator has no requests pending, but its door is still open, then it would be required to store the request, because the system is not "allowed" to tell the door to close, i.e., it is not part of its responsibilities.

The Post clause makes use of four Parameterized Predicates (shown with "_p" suffix). The definitions of these four Parameterized Predicates are given in Appendix C, along with all Parameterized Predicates and Functions that are used by the Operation Schemas of the elevator case study. The requestDoorToOpen_p Parameterized Predicate asserts that the system sent an "open door" message to the Door. The informSourceThatReqServiced_p Parameterized Predicate asserts that the system sent a "serviced your request" message to the appropriate ReqSource. The storeRequest_p Parameterized Predicate asserts that the request has become part of the system state population. The serveRequest_p Parameterized Predicate asserts that the system has made the request the current one and that it has sent a "go up" or "go down" message to the motor, depending on the current position of the cabin and the requested floor.

changeFloor

The changeFloor operation is called as a result of a Sensor notifying the system that the cabin is approaching a certain floor (denoted by the parameter) and there are no appropriate requests for this floor. The responsibility of this operation is derived from step 3 of the Enter Cabin use case and from step 4 of the Exit Cabin use case. The changeFloor operation is responsible for updating the system's state on the current position of the cabin, given the floor that is to be reached. The Operation Schema for the changeFloor operation is shown in figure 9.14.

The Aliases clause provides two expression-substitution declarations: cab, noReqsForThisFloor. noReqsToStopFor is a boolean expression that results in true if the Function makeStopAccordingToPolicy_f is false, given the cabin and approaching floor as parameters; noReqsToStopFor results in false otherwise. makeStopAccordingToPolicy_f is a Function (declared in Appendix C) that results in true if there are one or more requests for the approaching floor that meet the criteria of the elevator's stopping policy. For example, if it was decided that the elevator is in "express" mode then this Function would prohibit "pick-ups" on the way to the destination. By encapsulating this decision in a Function, the Operation Schema need not be changed to take into account a different stopping policy, both in terms of static and dynamic changes to policies. Thus, the use of this Function reveals a variation point to designers.

```
Operation: ElevatorControl::changeFloor (f: Floor);
Description: The system records that cabin has changed floors;
Use Cases: EnterCabin::{3;}; ExitCabin::{4;};
Scope: Cabin; Floor; Request; IsFoundAt; HasTargetFloor; HasCurrentRequest;
Aliases:
  cab: Cabin Is self.cabin;
  noReqsToStopFor: Boolean Is not self.makeStopAccordingToPolicy_f (cab, f);
Pre:
  noReqsToStopFor &
  self.request->notEmpty () &
  self.cabinMovedByOneFloor_p (cab, f) &
  cab.doorState = DoorState::closed; -- redundant due to invariant
Post:
  cab.currentFloor = f;
```

Figure 9.14: Operation Schema for changeFloor

The Pre clause asserts that this operation is assumed to be called in the following case: there are no appropriate requests to stop for at the given floor (line 1); there are pending requests, because the cabin is moving to serve one of them (line 2); the cab is moving, and the approaching floor is one higher or one lower than the previous floor visited, given that the cabin is moving in an upward or downward direction, respectively (line 3); and the door is closed (line 4). The third assertion is encapsulated by the cabinMovedByOneFloor_p Parameterized Predicate. The Post clause asserts that the IsFoundAt association has a link between the cabin and the supplied floor, f, as a result of the operation.

stopForRequest

The stopForRequest operation is the complement to the changeFloor operation. It is called as a result of a Sensor notifying the system that the cabin is approaching a certain floor (denoted by the parameter) and there are appropriate requests for this floor. The responsibility of this operation is derived from step 4 of the Enter Cabin use case and from step 5 of the Exit Cabin use case. The stopForRequest operation is responsible for updating the system's state on the current position and movement status of the cabin, and for stopping the cabin. The Operation Schema for the stopForRequest operation is shown in figure 9.15.

The Aliases clause provides two expression-substitution declarations: cab and stopAtThisFloor. stopAtThisFloor is a boolean expression that is equivalent to the result of a call to the makeStopAccordingToPolicy_f Function. The Messages clause indicates that the Motor actor may be sent messages of type StopAtNextFloor. Also, a single message instance of type StopAtNextFloor is declared in the New clause.

Operation: ElevatorControl::stopForRequest (f: Floor);
Description: The cabin has reached a floor that has been requested and is deemed worthy to stop for: it should stop the cabin;
Use Cases: EnterCabin::{4;}; ExitCabin::{5;};
Scope: Cabin; Floor; Request; IsFoundAt; HasTargetFloor; HasCurrentRequest;
Messages: Motor::{StopAtNextFloor;};
New: stop: StopAtNextFloor;
Aliases:
  cab: Cabin **Is** self.cabin;
  stopAtThisFloor: Boolean **Is** self.makeStopAccordingToPolicy_f (cab, f);
Pre:
  stopAtThisFloor &
  self.cabinMovedByOneFloor_p (cab, f) &
  cab.doorState = DoorState::closed; -- redundant due to invariant
Post:
  cab.currentFloor = f & -- new current floor for the cabin
  cab.movement = Movement::stopping &
  requestCabinToStop_p (stop);

Figure 9.15: Operation Schema for stopForRequest

The Pre clause asserts that there are appropriate requests for this floor to make it necessary to stop the cabin. And, the last two expressions are the same as the ones for the Operation Schema of the changeFloor operation. The Post clause asserts that, as a result of the operation, the IsFoundAt association has a link between the cabin and the supplied floor, f, the movement attribute of cabin has the (enumerated) value stopping, and the motor has been requested to stop (which is encapsulated by the requestCabinToStop_p Parameterized Predicate).

openForRequest

The openForRequest operation is called as a result of the Motor notifying the system that the cabin is stopped. The responsibility of this operation is derived from step 5 of the Enter Cabin use case and from step 6 of the Exit Cabin use case. The openForRequest operation is responsible for opening the cabin door and informing the relevant requesters that their requests have been serviced. The Operation Schema for the openForRequest operation is shown in figure 9.16.

The Aliases clause provides two expression-substitution declarations: cab, and reqsForThisFloor. reqsForThisFloor denotes all requests for the current floor of the cabin. The Messages clause indicates that the Door and ReqSource actors may be sent messages of type OpenDoor and ServicedRequest, respectively. And, two message instances are declared in the New clause.

```
Operation: ElevatorControl::openForRequest ();
Description: The cabin has stopped for a request: it should open the door and inform each
corresponding requester that the request has been serviced;
Use Cases: EnterCabin::{5;}; ExitCabin::{6;};
Scope: Cabin; Floor; Request; IsFoundAt; HasCurrentRequest; HasTargetFloor; HasSource;
Messages: Door::{OpenDoor;}; ReqSource::{ServicedRequest;};
New: open: OpenDoor; servicedReq: ServicedRequest;
Aliases:
   cab: Cabin Is self.cabin;
   reqsForThisFloor: Set (Request) Is cab.currentFloor.requestsForFloor@pre;
Pre:
   -- self.makeStopAccordingToPolicy_f (cab, cab.currentFloor) would also be part of the precondition if the
   -- stopping policy could not be changed dynamically
   cab.movement = Movement::stopping &
   cab.doorState = DoorState::closed; -- redundant due to invariant
Post:
   cab.movement = Movement::stopped & -- cabin is no longer moving
   cab.doorState = DoorState::open &
   requestDoorToOpen_p (open) &
   self.request->excludesAll (reqsForThisFloor) & -- removed request(s) for this floor
   serviceReq.oclIsNew ((servicedFlr => cab.currentFloor)) & -- indicates which floor is serviced
   reqsForThisFloor.source.sentOneToAll (serviceReq); -- sent to extReqSrc and/or intReqSrc
```

Figure 9.16: Operation Schema for openForRequest

The Pre clause asserts that the cabin is the stopping state and the door is closed immediately before the operation is executed. Note that the first line of the Pre clause is a commented expression that, if uncommented, would assert that there is at least one appropriate and pending request for this floor (according to makeStopAccordingToPolicy_f). This assertion provides additional context to the operation, highlighting that the operation is performed as the result of a stopForRequest operation. If it was to be asserted, it would be assumed that the stopping policy of the elevator, which is encapsulated by makeStopAccordingToPolicy_f, is the same as the one that was used in the preceding stopForRequest operation. However, this assumption may not hold in a system that is able to dynamically change the stopping policy of the elevator. It would seem unreasonable to restrict an implementation to a static stopping policy, and therefore, it is not part of the operation's precondition, but rather shown as a comment to provide additional context.

The Post clause asserts that, as a result of the operation, the movement attribute of the cabin is equivalent to stopped (line 1), the doorState attribute of the cabin is equivalent to open (line 2), the system requested the door to open (line 3), all requests for this floor are removed (line 4), and the system sent a message to the relevant requesters, informing them that their requests have been serviced (line 6).

serveNextRequest

The serveNextRequest operation is called as a result of the Door notifying the system that it is closed. The responsibility of this operation is derived from step 3 and 3a of the Exit Cabin use case. The serveNextRequest operation is responsible for requesting the cabin to go in the direction of the most eligible request that is still pending service. The Operation Schema for the serveNextRequest operation is shown in figure 9.17.

```
Operation: ElevatorControl::serveNextRequest ();
Description: The door is now closed and there are requests to serve, so the system serves the
most eligible request;
Use Cases: ExitCabin::{3; 3a;};
Scope: Cabin; Floor; Request; IsFoundAt; HasTargetFloor; HasCurrentRequest;
Messages: Motor::{GoUp; GoDown;};
New: up: GoUp; down: GoDown;
Aliases: cab: Cabin Is self.cabin;
Pre:
   cab.doorState = DoorState::open &
   cab.movement = Movement::stopped &
   self.request->forall (r | r.targetFloor <> cab.currentFloor); -- no requests for the current floor
Post:
   cab.doorState = DoorState::closed &
   if self.request->notEmpty () then
      cab.currentRequest = self.calcNextRequest_f (cab) & -- returns most eligible req
      requestCabinToMove_p (cab, cab.currentFloor, cab.currentRequest.targetFloor,
                              up, down)
   endif;
```

Figure 9.17: Operation Schema for serviceNextRequest

The Messages clause indicates that the Motor actor may be sent messages. Two message instances are declared in the New clause. The Pre clause asserts that the door is open (line 1), the cabin is stopped (line 2) and there are no requests pending for the current floor (line 3) immediately before the operation is executed.

The Post clause asserts that as a result of the operation: the door is closed, the system requested the motor to move the cabin in the direction of the most eligible request if there is at least one pending request. The "most eligible" request is calculated by the Function calcNextRequest_f (see Appendix C for its declaration). Depending on the algorithm chosen, the most eligible request may just be the request that is already the current one, e.g., if a pick-up was made on the way to the destination[1]. Again, by encapsulating this decision in a Function, the Operation Schema need not be changed to take into account a different scheduling policy. Thus, the use of this Function reveals a variation point to designers.

9.3.6 Applying Step 5 of the ANZAC Mapping Technique

Step 5 of the ANZAC mapping technique involves performing checks on the three models of the ANZAC specification that result from the four previous steps. All results from performing these checks have been fed back into the models shown so far. Consequently, I do not show the results in this subsection.

1. It would also seem reasonable to give priority to internal requests over external requests.

9.4 Addressing Concurrent Behavior in the ElevatorControl System

In this section, I describe the Protocol and Operation Model for the ElevatorControl system by considering that operations do take time to execute. As a result, the Operation Model, shown in section 9.3, is changed to take into account the removal of the instantaneous-operation assumption, and the new version is presented. Interestingly, the Protocol Model also needs to be changed because some of the guards are no longer valid if operations can execute in true concurrency.

9.4.1 Protocol Model (Revised Version)

In this subsection, I describe the revised Protocol Model for the ElevatorControl system that takes into account that the removal of the instantaneous-operation assumption.

By glancing at the Protocol Model for the ElevatorControl system, shown in figure 9.12, there seems to be no need to modify it, because it already accounts for "real" concurrency due to its use of concurrent states. However, by looking closely at the CabMovementAndDoorStateView (figure 9.11), it is possible to observe that some guards lead to race conditions; hence, in some cases, the Protocol Model does not describe correct system behavior. In particular, the CabMovementAndDoorStateView state (figure 9.11) uses guards that make the assumption that operations are executed atomically with respect to their evaluation, i.e., the guard is evaluated and the operation performed as an atomic unit from the perspective of the exterior (i.e., other operations).

To illustrate the problem of racing, consider the CabMovementAndDoorStateView state (figure 9.11) in the following situation: its internal states Stopped and Open are active, a doorIsClosed event is the current event, and the guard [**not** reqsToServe] evaluates to true. If, during the serveNextRequest operation (i.e., the operation attached to the transition), but before the serveNextRequest operation updates the doorState attribute of the cabin, a floorRequest operation is executed completely (as a transition in the ExtReqSourceActivity or IntReqSourceActivity states), then the request will unexpectedly be queued. Due to the reactive nature of the system and the assumptions made on servicing requests, an additional floorRequest operation will have to be performed before the system "realizes" that this request is pending.

In general, guards that make assertions about shared resources are not usually appropriate unless the whole transition is one atomic step (i.e., operations are performed atomically with respect to other operations). The revised CabMovementAndDoorStateView is shown in figure 9.18. There are two noteworthy differences to the one shown in figure 9.11. There are no guards shown, and the atFloor event is mapped to the operation decideOnStop.

It is important to emphasize that the guards in CabMovementAndDoorStateView are *not* shown rather than non-existent (i.e., true by default). If there were no guards, then the two regions would be non-deterministic. However, all the information necessary to decide which transition is taken is given by the Operation Schema of the corresponding operation (i.e., the one attached to the transition). In fact, the target of a transition is "calculated" in a post-operation manner, rather than in a pre-operation manner, i.e., the guard is evaluated after the operation is executed. The implication of a post-operation guard evaluation is that the event triggers a single operation; hence the need to map the atFloor event to only one operation.

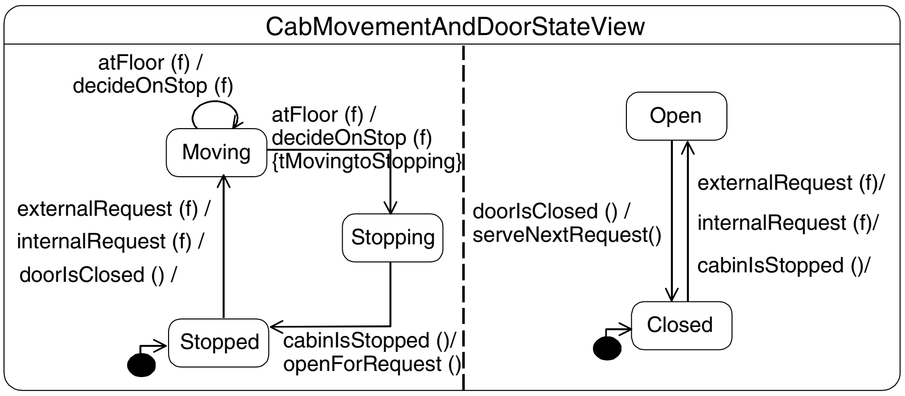

Figure 9.18: CabMovementAndDoorStateView (revised version)

The revised Protocol Model for the ElevatorControl system is the same one as shown in figure 9.12, except the revised CabMovementAndDoorStateView, shown in figure 9.18, is substituted for the former one (figure 9.11).

Timing Constraints

The problem statement implies that there are three timing invariants: "The door is guaranteed to stay open for at least 5 seconds", "It takes at least 1.5 seconds and at most 2 seconds to go from one floor to another one", and "the system must decide within 0.1 seconds if it is to stop the cabin". These timing constraints are specified with the following three invariants, respectively.

The invariant below states that the CabMovementAndDoorStateView state will stay in its Open state for at least 5 seconds. Note that since unique state names are used, it is unnecessary to fully qualify them, e.g., ElevatorControlActivity::CabMovementAndDoorStateView::Open.

```
context: ElevatorControl
inv: defined (Open.actTime > 5*Sec);
```

Recall that actTime is a time property of states, whose value is equal to the amount of time that the corresponding state has been active for. Also, note that the defined function has been used to ensure that the invariant is always defined—it is true whenever the expression is undefined, and equivalent to the truth of the expression otherwise. The use of this function covers the undefined problem on initialization of actTime and transDur (which is discussed in Chapter 7).

The invariant below states that the CabMovementAndDoorStateView state will stay in its Moving state for at least 1.5 seconds and at most 2 seconds. This constraint takes advantage of the fact that the atFloor self-transition leaves and then reenters the state every time that it is fired. Note that this constraint may not represent the exact timing requirement, because it does not take into account the time that it takes for the decideOnStop operation (self-transition) to execute. If the problem statement was clearer as to exactly what was meant by "go from one floor to another one", then it would be possible to be more precise with the constraint.

```
context: ElevatorControl
inv: defined (2*Sec > Moving.actTime > 1.5*Sec);
```

The invariant below states that the transition from Moving to Stopping (denoted by the label tMovingtoStopping) in the CabMovementAndDoorStateView state will take less than 0.1 seconds to complete. This corresponds to the fact that the system must take less than 0.1 seconds to decide if it is to stop the lift or not.

```
context: ElevatorControl
inv: defined (tMovingtoStopping.transDur < 0.1*Sec);
```

Recall that transDur is a time property of transitions, whose value is equal to the amount of time that it takes to go from source to target state; it is directly equivalent to the time it takes to execute the operation on the transition.

9.4.2 Operation Model (Concurrent Version)

In this subsection, I describe the revised Operation Model for the ElevatorControl system, which takes into account the removal of the instantaneous-operation assumption and addresses the other half of step 4's "atomic/non-atomic split" approach (ANZAC mapping technique), which was started in section 9.3.5.

According to the Protocol Model, there are four system operations to specify (one less than the sequential version). A floorRequest operation can be executed concurrently to another floorRequest operation and concurrently to one of the other three operations: decideOnStop, openForRequest, and serveNextRequest. I now present the Operation Schemas for each of these four operations, which makes use of the techniques described in Chapter 6 to clarify which resources are shared and the necessary synchronization constraints on the operations to ensure correctness.

floorRequest (concurrent version)

The Operation Schema for the floorRequest operation is shown in figure 9.19. It has similar content to the one shown in figure 9.13 (i.e., the atomic version): the same Operation, Description, Use Cases, Scope, Sends, and Pre clause. In describing it, I will concentrate on the difference to the atomic version.

```
Operation: ElevatorControl::floorRequest (f: Floor);
Description: A floor request (external or internal) is made to system, where it must decide to
reject it, serve it (open door, or send cabin to floor), or store it for later;
Use Cases: EnterCabin::{2; 2a; 2b;}; ExitCabin::{3; 2a; 6; (2-6)||a; 3a; 3b;};
Scope:
   Cabin; Floor; Request; HasTargetFloor; HasCurrentRequest; IsFoundAt; HasSource;
Shared:
   Cabin; Request; HasTargetFloor; HasCurrentRequest; IsFoundAt; HasSource;
Messages:
   Motor::{GoUp; GoDown;}; Door::{OpenDoor;}; ReqSource::{ServicedRequest;};
New:
   newReq: Request; up: GoUp; down: GoDown; open: OpenDoor;
   servicedReq: ServicedRequest;
Aliases:
   cab: Cabin Is self.cabin;
   notAlreadyRequested: Boolean Is f.requestsForFlr.source->excludes (sender);
   alreadyAtFloorAndStopped: Boolean Is
      cab.currentFloor = f and cab.movement = Movement::stopped;
   elevatorIsIdle: Boolean Is
      cab.doorState = DoorState::closed and cab.movement = Movement::stopped;
Pre: true;
Post:
❶ rely notAlreadyRequested then
❷    rely alreadyAtFloorAndStopped then
         cab.doorState@postAU = DoorState::open &
         requestDoorToOpen_p (open) & -- request door to open even if it is already open
         informSourceThatReqServiced_p (sender, serviceReq, f)
      fail
❸       rely elevatorIsIdle then
            storeRequestV2_p (f, newReq, sender)'ACU & -- request is persisted
            serveRequestV2_p (cab, cab.currentFloor@rd, f, newReq, up, down)'ACU
         fail
            storeRequestV2_p (f, newReq, sender)
         endre
      endre
   endre;
```

Figure 9.19: Operation Schema for floorRequest (concurrent version)

The Shared clause highlights that most resources that the Operation Schema uses are shared—this is mainly due to the fact that this operation can execute concurrently to other floorRequest operations and to all the other types of operations. The Post clause has a similar form to the one shown in figure 9.13. The rely-condition of the first rely expression, marked by ❶, will hold true in the case that the floor has not already been requested by the same source (ExtReqSource or IntReqSource actor). In the case that it does not hold, the operation does nothing. Note that even though the expression used in the rely-condition denotes use of the shared association HasSource, the subset of the association denoted by the expression can not be changed by other concurrent operations, because a request can not be made from the same ReqSource concurrently, according to the Protocol Model (a consequence of the inherent concurrency guideline). This highlights the limitation of making all tuples of an association a single resource, where sharing is defined at this level of granularity.

The second rely expression, which begins at ❷, asserts that if it can be relied upon that the cabin is at the requested floor and stays there and the cabin is stopped and it stays stopped, then the door is told to open and the calling actor is informed that the request has been serviced. Note that the second and third effects in the then-part of the rely expression are Parameterized Predicates.

In the case that the rely-condition at ❷ cannot be relied upon, then the fail-part of the rely expression must be realized instead, i.e., the third rely expression is asserted to hold. The rely-condition of the third rely expression, marked by ❸, asserts that if it can be relied upon that the door stays closed and the cabin stays stopped during the time that the effects are carried out, then the request is stored and it is made the current one, which involves telling the motor to go in the direction of the requested floor. If this condition can not be relied upon, the request is simply stored. Note that the two Parameterized Predicates in the then-part of the rely expression (❸) assert an atomic compound-effect.

Analyzing the rely conditions of this Operation Schema, it is clear that an implementation that waited on the first two conditions (❶ and ❷) to become true may not terminate. Looking closer at the issue, in the case of ❶, if the rely-condition is not initially true, then it will never become true, because a request can not be made from the same ReqSource concurrently, as stated earlier. In the case of ❷, if the rely-condition is not initially true, then it may never become true, because it is possible that the other ReqSource actor may never request for the same floor, which would be the only way that the condition could become true. As a consequence, an implementation of the rely expressions at ❶ and ❷ should use immediate semantics, or at least wait semantics with time-out, for it to be able to guarantee termination of the operation—a condition that must be satisfied by all implementations of an Operation Schema.

decideOnStop (concurrent version)

The Operation Schema for the decideOnStop operation is shown in figure 9.20. It can be seen as the combination of the changeFloor and stopForRequest operations that are shown in figure 9.14 and figure 9.15, respectively. It has similar content to them both. I will concentrate on the differences that were introduced to take into account non-atomicity of execution.

The Post clause asserts that the currentFloor of the cabin is updated atomically and that if it can be relied upon that there are appropriate requests to stop for (according to the truth of the Function makeStopAccordingToPolicy_f), then the motor is requested to stop, otherwise the cabin is allowed to continue. The body of the rely expression contains two expressions, one of which is a Parameterized Predicate.

```
Operation: ElevatorControl::decideOnStop (f: Floor);
Description: The cabin has reached a floor: the system must decide whether there is a request
that is deemed worthy to stop for, in which case the system will stop the cabin;
Use Cases: EnterCabin::{3; 4;}; ExitCabin::{4; 5;};
Scope: Cabin; Floor; IsFoundAt; HasTargetFloor; HasCurrentRequest;
Shared: Cabin; IsFoundAt; HasTargetFloor; HasCurrentRequest;
Messages: Motor::{StopAtNextFloor;};
New: stop: StopAtNextFloor;
Aliases:
  cab: Cabin Is self.cabin;
  stopAtThisFloor: Boolean Is self.makeStopAccordingToPolicy_f (cab, f);
Pre:
  self.cabinMovedByOneFloor_p (cab, f) &
  self.request->notEmpty () &
  cab.doorState = DoorState::closed; -- redundant due to invariant
Post:
  cab.currentFloor@postAU = f & -- new current floor for the cabin
  rely stopAtThisFloor then
    cab.movement@postAU = Movement::stopping &
    requestCabinToStop_p (stop)
  endre;
```

Figure 9.20: Operation Schema for decideOnStop (concurrent version)

Analyzing the rely-condition, if the condition is initially false, then it will only become true if an additional request is made that is judged appropriate to stop for by the makeStopAccordingToPolicy_f Function. Thus, once it becomes true it will stay that way, because only this operation is allowed to stop the cabin (which would eventually remove the request with a subsequent openForRequest operation).

openForRequest (concurrent version)

The Operation Schema for the openForRequest operation is shown in figure 9.21. It has similar content to the one shown in figure 9.16 (i.e., the atomic version): the same Operation, Description, Use Cases, Scope, Sends, and Pre clause. Again, I will concentrate on the differences that were introduced to take into account the fact that the operation can execute concurrently with other operations.

```
Operation: ElevatorControl::openForRequest ();
Description: The cabin has stopped for a request: it should open the door and inform each
corresponding requester that the request has been serviced;
Use Cases: EnterCabin::{5;}; ExitCabin::{6;};
Scope: Cabin; Floor; Request; IsFoundAt; HasCurrentRequest; HasTargetFloor; HasSource;
Shared: Cabin; Request; HasCurrentRequest; HasTargetFloor; HasSource;
Messages: Door::{OpenDoor;}; ReqSource::{ServicedRequest;};
New: open: OpenDoor; servicedReq: ServicedRequest;
Aliases:
   cab: Cabin Is self.cabin;
   curFlr: Floor Is cab.currentFloor@pre;
   reqsForThisFloor: Set (Request) Is curFlr.requestsForFloor@preACU;
Pre:
   cab.movement = Movement::stopping &
   cab.doorState = DoorState::closed; -- redundant due to invariant
Post:
   cab.movement@postACU = Movement::stopped &
   cab.doorState@postACU = DoorState::open &
   requestDoorToOpen_p (open) &
   self.request@postACU->excludesAll (reqsForThisFloor) & -- removed request(s) for this floor
   serviceReq.oclIsNew ((servicedFlr => curFlr)) & -- indicates which floor is serviced
   reqsForThisFloor.source.sentOneToAll (serviceReq); -- sent to extReqSrc and/or intReqSrc
```

Figure 9.21: Operation Schema for openForRequest (concurrent version)

The Post clause is more or less equivalent to the one shown in figure 9.16, except all updates to shared resources are asserted to be performed atomically. One point to note is that the update to movement and doorState of the cabin and the removal of the serviced requests are an atomic compound-effect. The reason for asserting that the compound-effect is to be performed atomically is so that other operations that are running concurrently are not mislead by incomplete information about the state of the system.

To illustrate this issue, consider the following situation: an openForRequest operation removes all requests for the floor, but before it updates the movement and doorState attributes, an floorRequest operation is executed. In this case, the floorRequest operation will store the request, classifying it as pending, because it thinks that the cabin is still stopping. This situation is undesirable because the request will stay pending until the door becomes closed, where the system would need to unnecessarily reopen the door again (see serveNextRequest operation), because it realizes that there is another request for the current floor.

serveNextRequest (concurrent version)

The Operation Schema for the serveNextRequest operation is shown in figure 9.22. It has similar content to the one shown in figure 9.17. I will concentrate on the differences that were introduced to take into account the fact that the operation can execute concurrently with other operations.

```
Operation: ElevatorControl::serveNextRequest ();
Description: The door is now closed and there are requests to serve, so the system serves the
most eligible request;
Use Cases: ExitCabin::{3; 3a;};
Scope: Cabin; Floor; Request; IsFoundAt; HasTargetFloor; HasCurrentRequest;
Shared: Cabin; Request; HasTargetFloor; HasCurrentRequest;
Messages: Motor::{GoUp; GoDown;};
New: up: GoUp; down: GoDown;
Aliases: cab: Cabin Is self.cabin;
Pre:
   cab.doorState = DoorState::open &
   cab.movement = Movement::stopped;
Post:
   cab.doorState@postACU = DoorState::closed &
   rely self.request->notEmpty () then
      cab.currentRequest@postACU = self.calcNextRequest_f (cab)@rd & -- returns most eligible req
      requestCabinToMoveOrDoorToOpen_p (cab, cab.currentFloor@pre,
         cab.currentRequest@postACU.targetFloor, up, down)'ACU
   endre;
```

Figure 9.22: Operation Schema for serviceNextRequest (concurrent version)

The Post clause asserts that the doorState of the cabin is updated and that if it can be relied upon that there are pending requests, then the most eligible request is identified, and the motor is told to go in the direction of this request or the door is told to open. Note that if the rely-condition is initially true it will stay that way for the duration of the operation, because the only operation that can remove requests is the openForRequest operation, which cannot not occur in parallel to this operation (according to the Protocol Model). All effects in this Post clause describe an atomic compound-effect, even the first line of the Post clause.

This means that if the rely-condition holds, then the doorState of the cabin will be updated together with the effect of choosing the next request and starting to serve it as an atomic action (from the perspective of other operations).

Perhaps surprisingly, there is one functional difference to the version shown in figure 9.17, which is due to the fact that it can not be assumed that this operation is executed atomically. The previous version (figure 9.17) assumes that there are no pending requests for the current floor at the time of the execution of the operation. However, this version can not make this assumption, because a request for the current floor may become pending between the time this operation starts and the time that it reads the request queue (i.e., self.request).

To illustrate the issue, consider the sequence diagram shown in figure 9.23, where the time axis is from top to bottom. The diagram shows a doorIsClosed message arriving and triggering a serveNextRequest operation (at time ❶). At time ❷, an externalRequest message arrives and triggers a floorRequest operation, which terminates execution at time ❸. At time ❹, the serveNextRequest operation starts to execute its atomic compound-effect, which terminates execution at time ❺. Finally, the serveNextRequest operation terminates execution at time ❻.

In this scenario, if the request that triggered the floorRequest operation was for the current floor, then the atomic compound-effect of the serveNextRequest operation would find a pending request for the current floor. As a consequence, the operation must be able to deal with the request by opening the door if it is calculated to be the most eligible request (according to calcNextRequest_f), which is highly likely. Note that if the externalRequest message had arrived before the doorIsClosed message then the floorRequest operation would have opened the door and the doorIsClosed message would be simply ignored. Also, note this is a non-issue for the instantaneous version of the operation, because floorRequest operations could never "sneak" in before the serveNextRequest operation has terminated.

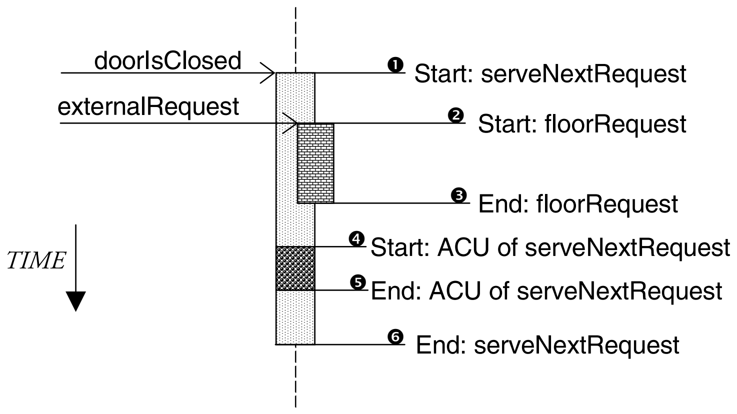

Figure 9.23: Sequence diagram showing interleaving of operation executions

9.5 Remarks

In this chapter, the ANZAC approach was applied to the ElevatorControl system. This involved the development of three use case descriptions: Take Lift, Enter Lift and Exit Lift. These use case descriptions were refined into an ANZAC specification, using the ANZAC mapping technique.

The use case descriptions provide the contextual information necessary to clarify the overall protocol between the ElevatorControl system and its environment. They are written in a form that is accessible to both technical and non-technical stakeholders, providing them with a basis for negotiating the behavioral aspects of the ElevatorControl system. However, the use case descriptions were only able to show the responsibilities of the system from an elevator user's perspective. In this way, the mapping from the use case descriptions to an ANZAC specification allowed the clarification of how the ElevatorControl system should precisely deal with requests, particularly in the case when there are multiple pending requests.

The ANZAC specification offers a precise description of the ElevatorControl system's responsibilities and its interaction protocol with its environment. In particular, it sheds light on some of the tricky aspects of the concurrent behavior of this system. Furthermore, it abstracts above the stopping and scheduling policy of the elevator using functions in the corresponding Operation Schemas. The abstraction of these two policies allows them to vary without requiring a change to the specification. However, the specification does not include fairness constraints on the scheduling policy, e.g., a request is eventually served. This is because the language is not sufficiently rich to do so. To formalize such a constraint, some form of temporal logic would be required. (It would also be useful to be able to define deadlines on serving requests, e.g., requests must be served within a certain deadline.)

The ANZAC specification for the elevator control system has an operational feel to it, and therefore it could be perceived as having implementation bias, since it *is* an implementation in an abstract sense. However, it is arguably easier to use than a purely mathematical specification, and it provides an easier and more certain path to an implementation, which both prove hard to beat.

Readers that are interested in more examples of elevator systems can find ANZAC specifications for a number of different variations on the one shown in this chapter in [Se02b], e.g., multiple cabins rather than a single cabin, requests that can be cancelled, external requests that have directions, subsystem specifications, etc.

Chapter 10:
Conclusion

The central element of this thesis has been the development of a new approach called ANZAC that offers a systematic way to capture the Behavioral Stakeholders Contract and the Behavioral Design Contract for reactive systems. The ANZAC approach is intended to be used to communicate to the development team the required behavior of the future system.

In section 10.1, I restate the main objective of this thesis, give an overview of the general approach taken in this thesis, and summarize the solution proposed by this thesis. In section 10.2, I provide a critical review of the proposed solution. In section 10.3, I describe some perceivable areas of future research. Finally, in section 10.3, I make some concluding remarks.

10.1 Summary

The principal objective of this thesis was to improve the state-of-the-art of specifications that are used to communicate the behavior of the (future) system to the development team. In addressing the objective, this work initially involved defining the essential requirements of specifications that could ensure that the development team has a precise, correct and common understanding of the way the system is required to behave.

After identifying some requirements, it became apparent that some of them diverged on some issues. Specifically, the need for a description that is justified against the interests of the stakeholders suggests that the goals and expectations of stakeholders be explicitly stated. However, the need for a “designer-friendly” description suggests that only information that describes the system's responsibilities be included, where the inclusion of descriptions of stakeholder goals was considered to be too "noisy" and not essential to the task of the development team. This divergence led to the conclusion that it was in fact necessary to address two potentially different audiences that each have different objectives and needs: stakeholders and designers.

As a result of this reasoning, two general kinds of specifications were distinguished and perceived to be necessary to address the requirements adequately; one that addresses the concerns of the designers, providing a precise description of the system responsibilities; and one that addresses the concerns of the stakeholders in general, providing an informal description of the agreed goals that the stakeholders have against the system. The first specification is referred to as the Behavioral Design Contract and the second one is referred to as the Behavioral Stakeholders Contract.

By addressing the requirements with two different specifications, it is necessary to ensure that they can be nevertheless coherently combined to satisfy the main objective (stated earlier). Consequently, an explicit bridge between the two specifications was judged to be necessary. Further motivation for an explicit relationship between the two specification came from the desire to ensure that the Behavioral Design Contract was justified with respect to the interests of stakeholders. In this respect, it seemed appropriate to ensure that the Behavioral Design Contract could be refined from the Behavioral Stakeholders Contract. As a consequence, the formalisms chosen for the two specifications were required to be compatible along refinement lines.

In terms of selecting and establishing a formalism for these two specifications, two additional requirements were proposed: First, the specifications should support an evolutionary software development process (which takes into account the fact that the construction and maintenance of the specification is likely to be performed in iterations and increments). And, secondly, it should be ensured that all proposals could be realized using mainstream languages and tools.

10.1.1 Realization of the Two Contracts

In this thesis, these two specifications were concretely realized as part of the ANZAC approach. The ANZAC approach defines two work artifacts called the ANZAC use case descriptions and the ANZAC specification, which express the Behavioral Stakeholders Contract and the Behavioral Design Contract, respectively. ANZAC use case descriptions offer an informal and usage-oriented description of the concordant goals that the stakeholders have against the system. An ANZAC specification offers a precise, operational description of the system responsibilities in servicing all possible requests that it can receive over its lifetime.

In the ANZAC approach, the ANZAC use case descriptions are developed following the ANZAC use case framework. This framework defines the context, purpose, style and form of an ANZAC use case description, and it provides a goal-based approach to use case elicitation. Once a number of ANZAC use case descriptions are established, they can be refined to an ANZAC specification. This refinement procedure is (informally) defined by the ANZAC mapping technique.

An ANZAC specification is developed by the description of three models, which each express a different but complementary view of the system. These three models are called the Concept Model, the Operation Model, and the Protocol Model. The Concept Model defines the system's state space in terms of concepts from the problem domain, the Operation Model describes the effect of system operations on the system state, and the Protocol Model defines the correct behavior of the system in terms of its (allowable) input protocol.

In terms of support for iterative and incremental development, the ANZAC approach offers some useful qualities: the interplay between ANZAC use cases and the ANZAC specification lends itself nicely to refinement by iteration, and incremental development can be based on the units of use cases and operations, which define partial system capability, i.e., an increment. In terms of an explicit link to mainstream languages and tools, both ANZAC use case descriptions and the ANZAC specification are realized in UML, which is currently the most prominent language for modeling software in practice.

In figure 10.1, I provide a side-by-side overview of the ANZAC approach, which shows the different but complementary focus of ANZAC use case descriptions and the ANZAC specification.

	ANZAC use case descriptions	**ANZAC specification**
Objective	Define the Behavioral Stakeholders Contract for a reactive system.	Define the Behavioral Design Contract for a reactive system.
Target Audience	All stakeholders of the (software) system under development.	The designers of the (software) system under development.
Description of the System	They offer an informal and usage-oriented description of the system. Descriptions are existential rather universal (due to the scenario nature).	It offers a precise, operational description of the system.
Focus	Scenarios describing the usage goals of the stakeholders against the system	System behavior in terms of the effect of operations on the system state and the temporal ordering of those operations.
Unit of Increment	Use case	Operation
Work Artifacts	A set of use case descriptions, which each describe a distinct goal of an actor.	The Concept, Operation and Protocol Models, which each represent a distinct but complementary view of the system.
Language	Informal, structured natural language (narrative)	Mixture of visual and textual notations: UML class diagram, UML state machine, and UML's Object Constraint Language

Figure 10.1: Overview of the ANZAC use case descriptions and the ANZAC specification

10.2 Critical Review

In this section, I examine the strengths and weaknesses of the proposed solution.

10.2.1 Advantages

The ANZAC approach addresses the problems faced by developers in finding languages, tools and techniques that can be used to understand and document the way a system under development should behave. It offers a useful contrast between informal and formal notations, emphasizing precision and rigor in some contexts, and informality and expression in other contexts. Perhaps the most appealing feature of the approach is the way that it deals with capturing the necessary information about the future system by using two specifications, which each has a distinct purpose and a different audience. By explicitly separating the specifications along the lines of the needs and objectives of each audience, the approach is better able to tackle the overall objective.

The ANZAC approach makes extensive use of the principles of abstraction, projection and decomposition to combat the complexities of specifying the behavior of reactive systems. And, it offers a systematic approach to the specification of system behavior by ensuring that each constraint that it places on the design of the system is aligned with and traceable to the (system-related) goals of the stakeholders. The approach explicitly describes the interface between the system and its environment, and it makes quite some attempt at providing a specification to the development team that gives them all the information needed to design the system and no more, avoiding under- or over-specification.

The ANZAC approach places a particular focus on the specification of concurrency and timing constraints, providing a number of guidelines and techniques (e.g., inherent concurrency guideline, divide-by-collaboration technique) and language constraints (e.g., rely expressions, atomic compound-effects, time-based properties on states and transitions, auto-concurrency) for this purpose. The technique and language proposed for capturing timing constraints is especially targeted towards understandability; it offers a response to the question "What is the smallest extension possible to UML state machine so that it can support the specification of typical timing constraints on reactive systems?".

10.2.2 Remaining Problems and Limitations

As much as one would like to believe that there could exist an approach that can do it all (i.e., Brook's silver bullet), one should be extremely skeptical to believe any such claims in a software world that faces new challenges on every turn. In this regard, I admit that the ANZAC approach has a number of limitations and there are still some remaining problems to address.

The work achieved on concurrency and synchronization constraints is still in an experimental stage. It is yet to be proved whether the techniques and guidelines proposed in this work for defining synchronization constraints in Operation Schemas are in fact cost effective. Clearly, providing information on the inherent synchronization needed by the system to the development team is extremely beneficial. However, in some cases, coming up with an abstraction that does not bias one concurrency strategy over another is significantly more difficult than just choosing one strategy. In such cases, the gain in "abstraction" may not be deemed worth the effort in achieving it.

Still on the subject of concurrency, there are some kinds of constraints that cannot be specified with ANZAC specifications. In particular, it is not possible to express fairness constraints in the language. This point was illustrated with the ANZAC specification for the elevator control system (Chapter 9). Also, it is not possible to specify activity-level atomicity constraints—currently it is only possible to define atomicity constraints within operations. Having said that, it is nevertheless important to ensure that any new proposal would not over-complicate the language from a usage perspective.

Although the technique and language chosen for specifying timing constraints is simple and easy to learn and use, it is not able to express all kinds of timing requirements. In particular, it is not able to express timing constraints that require timing information on event arrival times, and that require a means to specify which sub-state of an auto-concurrent (or auto-orthogonal) state that the constraint applies to. Introducing this information into a UML state machine complicates the model and requires unintuitive extensions to it for that purpose, making it less accessible to developers.

10.3 Future Work

There are four possible directions of future work that are concerned with the ANZAC approach:

- Tool support for ANZAC approach:

  There has already been some work performed towards providing tool support for ANZAC specifications. More precisely, there is a syntax and type checker for Operation Schemas called the LGL schema compiler (discussed in Appendix B), and there is a (Java) code generator for the Protocol Model. The LGL schema compiler could be extended with code generation capabilities and integrated with the Protocol Model code generator for the purpose of generating throwaway prototypes. I believe the ability to generate (throwaway) prototypes is extremely important and useful, because it allows the development team to get early and often customer feedback, which has clear advantages in a volatile project context.

With respect to the ANZAC approach in general, it is very important for its usability that there be tool support for enforcing consistency and aiding the refinement activity between ANZAC use case descriptions and ANZAC specifications.

- Specification-based testing:

  Closely related to prototype generation, it would be interesting to explore the possibilities for generating test oracles from ANZAC specifications. Two main obstacles are perceived to lie in the way of this endeavor: finding a way to relate the abstract state space, as defined by the Concept Model, to the concrete state space, as defined by the program static structure (i.e., the implementation data structures); and dealing with the non-determinism in OCL.

- Relating ANZAC to Design (Concurrency/Synchronization Mechanisms):

  It would be useful to provide support for performing analysis on the synchronization constraints in Operation Schemas, and, as a result, provide some hints to the development team as to which concurrency control mechanisms would be suitable.
  In addition, it would be extremely useful to be able to provide support for relating synchronization constructs of the Operation and Protocol Models (and usage patterns of synchronization constructs) to particular transaction models, e.g., nested rely expressions are mapped to nested transactions, etc.

- Component contracts:

  This work has addressed strictly the issues that surround system specification. However, many of the techniques that have been proposed by this work could be reused in the area of component specifications. In fact, it should be possible to enhance the ANZAC specification so that it could be used to define component contracts. In particular, a proposal for enhancing an ANZAC specification with quality of service constraints would be pertinent. Such an extension would promote a more dynamic notion of contract.
  Ideally in component evaluation and selection activities, there should be a specification of the required component and specifications for existing components. This means that selection would involve matching the required component contract to an existing component contract [ABB+01]. Unfortunately, this is a wickedly hard problem in general. So, by providing precise specifications of component behavior, this activity could be facilitated.

- Language for use case frameworks:

  In this thesis, I only informally described the ANZAC use case framework. It would however be interesting to come up with a language and/or template for defining use case frameworks in general. In this way, multiple use case frameworks could be defined in a consistent and comparable manner. A recognized use case framework language would be able to focus future suggestions and proposals for new kinds of use cases, forcing writers to clarify the scope, purpose, style, domain, etc. of their proposed use case description.

- Goal-based elicitation of aspects:

  It is clear that object-oriented design does not cluster by goals. However, there is currently quite some activity in the development of aspect-oriented approaches for design, i.e., aspect-oriented design. It would be interesting to study the relationship between stakeholder system-usage goals as described using ANZAC use case descriptions and aspect-oriented designs.

10.4 Final Remarks

The ANZAC approach to the specification of reactive system behavior has quite some promise. It makes use of mainstream languages and tools but draws upon many well established software engineering techniques and principles. It offers a useful separation of concerns between the different work artifacts and addresses not only different modeling concerns but also different target audiences. Significant parts of the ANZAC approach have been successfully taught to students and practitioners and used in a number of small-to-medium sized projects, which include business systems, embedded systems, application tools and utilities, and games. Finally, the ANZAC approach has been integrated into the Fondue object-oriented development method [Sof02], which offers a process for the full treatment of the software development lifecycle.

Bibliography

[ABB+01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, J. Zettel; "Component-Based Product Line Engineering with UML". Addison-Wesley 2001.

[Abr88] J-R. Abrial; "The B tool (proof proving tool)". In Proceedings of VDM'88: VDM - The Way Ahead. 2nd VDM-Europe Symposium. LNCS (Lecture Notes in Computer Science), no. 328, , pp. 86-87, Springer Verlag, 1988.

[Abr96] J-R. Abrial; "The B-Book: Assigning Programs to Meanings". Cambridge University Press, 1996.

[AF00] B. Anderson, P. Fertig; "Improving Your Use Cases". Tutorial Notes, OOPSLA 2000, Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000.

[AM01] F. Armour, G. Miller; "Advanced Use Case Modeling". Object Technology Series, Addison-Wesley 2001.

[And00] G. Andrews; "Foundations of Multithreaded, Parallel, and Distributed Programming". Addison-Wesley, 2000.

[AP98] V. Alagar, K. Periyasamy; "Specification of Software Systems". Graduate Texts In Computer Science Series, Springer, 1998.

[Atk97] C. Atkinson; "Adapting the Fusion Process to Support the UML". Object Magazine, Sigs Publications, November 1997.

[Bac98] J. Bacon; "Concurrent Programming". Addison-Wesley, 1998.

[Bar96] J. Barnes; "Programming in Ada 95". Addison-Wesley, 1996.

[BBC00] M. Buffo, D. Buchs, S. Chachkov; "CoopnTools a Toolkit for the support of CO-OPN". Proceedings of the Tools Demonstration of the 21th International Conference on Application and Theory of Petri Nets, Aahrus University, June 2000, pp. 2-6.

[BCR00] E. Börger, A. Cavarra, E. Ricobene; "Modeling the Dynamics of UML State Machines". ASM 2000 - International Workshop on Abstract State

Machines: Theory and Applications; Gurevich, Kutter, Odersky, Thiele (Ed.), LNCS (Lecture Notes in Computer Science), no. 1912, pp. 223-241, Springer Verlag, 2000.

[BCR01] E. Börger, A. Cavarra, E. Ricobene; “Solving Conflicts in UML State Machines Concurrent States”. Position Paper for the Workshop on Concurrency Issues in UML. UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, 4th International Conference, Canada, 2001.

[BLP+01] R. Baskerville, L. Levine, J. Pries-Heje, B. Ramesh, S. Slaughter; “How Internet Software Companies Negotiate Quality”. IEEE Computer, pp. 51-57, May 2001.

[BMR93] A. Borgida, J. Mylopolous and R. Reiter; “‘...And Nothing Else Changes: The Frame Problem in Procedure Specifications”. Proceedings of ICSE-15, pp. 303-314, IEEE Computer Society Press, 1993.

[BMR95] A. Borigda, J. Mylopoulos and R. Reiter; “On the Frame Problem in Procedure Specifications”. IEEE Transactions on Software Engineering, vol. 21, no. 10, pp. 785-798, 1995.

[Boe86] B. Boehm; “A spiral model of software development and enhancement”. ACM Software Engineering Notes, vol. 11, no. 4, pp.14-24, 1986.

[Boo94] G. Booch; “Object-Oriented Analysis and Design with Application”. Addison-Wesley, 1994.

[BRJ98] G. Booch, J. Rumbaugh, I. Jacobson; “The Unified Modeling Language User Manual”. Addison-Wesley, 1998.

[Bro87] F. Brooks; “No Silver Bullet: Essence and Accidents of Software Engineering”. IEEE Computer, vol. 20, no. 4, pp. 10-19, April 1987.

[Bro95] F. Brooks; “The Mythical Man-Month”. 20th Anniversary Edition, Addison-Wesley, 1995.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes; “Object-Oriented Development: The Fusion Method”. Prentice Hall, 1994.

[Car95] J. Carroll; “Scenario-based Design: Envisioning Work and Technology in System Development”. Wiley, 1995.

[CD94] S. Cook and J. Daniels; “Designing Object Systems: Object-Oriented Modelling with Syntropy”. Prentice-Hall, 1994.

[CF98] A. Cockburn, M. Fowler; “Question time! about use cases”. ACM SIGPLAN Notices, vol. 33, issue 10; Proceedings of the conference on Object-oriented programming, systems, languages, and applications, 1998.

[Che76] P. Chen; “The Entity-Relationship Model—Toward A Unified View of Data”. ACM Transactions on Database Systems, vol. 1 no. 1, pp. 9-36, 1976.

[CJ93] D. Coleman, P. Jeremaes; “Object-Oriented Development: The Fusion Method”. NTU Course Notes, October, 1993.

[CKM+99a] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, and A. Wills; “Defining the Context of OCL Expressions”. UML ‘99 - The Unified Modeling Language: Beyond the Standard, 2nd International Conference, USA; France & Rumpe (Eds.), LNCS (Lecture Notes in Computer Science), no. 1723, Springer-Verlag, 1999.

[CKM+99b] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills; “The Amsterdam Manifesto on OCL”. Technische Universität München, Department of Informatics, Technical Report, TUM-I9925, 1999.

[CKS+01] V. Crettaz, M. Kandé, S. Sendall, and A. Strohmeier; “Integrating the ConcernBASE Approach with SADL”. <<UML>> 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, 4th International Conference, Canada; Gogolla & Kobryn (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, pp. 166-181, Springer Verlag, 2001.

[CL99] L. Constantine, L. Lockwood; “Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design”. Addison-Wesley, 1999.

[CL01] L. Constantine, L. Lockwood; “Structure and Style in Use Cases for User Interface Design”. In van Harmelen (Ed.), Object-Modeling and User Interface Design, Addison-Wesley, 2001.

[CLD99] P. Coad, E. Lefebvre, J. De Luca; “Java Modeling In Color With UML: Enterprise Components and Process”. Prentice Hall 1999.

[Coc97] A. Cockburn; “Structuring Use Cases with Goals”. Journal of Object-Oriented Programming, Sep/Oct, pp. 35-40, and Nov/Dec, pp. 56-62, 1997.

[Coc98] A. Cockburn; “Basic Use Case Template”. October, 1998. Available on http://members.aol.com/acockburn/papers/uctempla.htm

[Coc00] A. Cockburn; “Writing Effective Use Cases”. Addison-Wesley 2000.

[Col95] D. Coleman; “Fusion with use cases: Extending Fusion for requirements modeling”. HP Labs, slide presentation 1995.

[CY91] P. Coad, E. Yourdon; “Object-Oriented Analysis”. 2nd edition, Prentice-Hall, 1991.

[Dav88] A. Davis; “A Comparison of Techniques for the Specification of External System Behavior”. Communications of the ACM, vol. 31, no. 9, pp. 1098-1115, September 1988.

[Dav90] A. Davis; “Software Requirements Analysis and Specification”. Prentice Hall, 1990.

[DH94] D. Drusinsky, D. Harel; “On the power of bounded concurrency I: Finite automata”. Journal of the ACM, 41(3), pp. 517–539, 1994.

[DH99] W. Damm and D. Harel; “LSCs: Breathing Life into Message Sequence Charts”. Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), March 1999.

[Dij68] E. Dijkstra; “Goto statement considered harmful”. Communications of ACM, vol. 11, no. 3, pp. 147-8, 1968.

[Dij76] E. Dijkstra; “A Discipline of Programming”. Prentice-Hall 1976.

[DLF93] A. Dardenne, A. van Lamsweerde, S. Fickas; “Goal-Directed Requirements Acquisition”. Science of Computer Programming, vol. 20(1-2), pp. 3-50, 1993.

[Dou99] B. Douglass; “Real-Time UML: Developing Efficient Objects for Embedded Systems”. Second Edition, Object Technology Series, Addison-Wesley, 1999.

[DW98] D. D’Souza and A.Wills; “Objects, Components and Frameworks With UML: The Catalysis Approach”. Addison-Wesley 1998.

[Eas91] S. Easterbrook; “Elicitation of Requirements from Multiple Perspectives”. Ph.D Thesis, Department of Computing, Imperial College of Science, Technology & Medicine, London University, 1991.

[EG00] G. Engels, L. Groenewegen; “Object-Oriented Modeling: A Roadmap”. In Finkelstein (Ed.), The Future of Software Engineering, Special Volume published in conjunction with ICSE 2000.

[Eme89] E. Emerson; “Temporal and Modal Logic”. In J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Amsterdam, 1989, pp. 995-1072.

[FFF99] K. Finney, N. Fenton, and A. Fedorec; “Effects of Structure on the Comprehensibility of Formal Specifications”. IEEE Proc.-Softw. vol. 146, no. 4, August 1999.

[FFM90] A. Finkelstein, L. Finkelstein, T. Maibaum; “Engineering-In-The-Large: Software Engineering and Instrumentation”, Proceedings of UK IT '90, pp 1-8, 1990.

[Fir99] D. Firesmith; “Use Case Modeling Guidelines”. Proc. 30th Conference on Technology for Object-Oriented Programming Languages and Systems (TOOLS-30), pp. 184-193, IEEE Computer Society, 1999.

[Flo67] R. Floyd; “Assigning meaning to programs”. In Proceeding of the Symposium in Applied Mathematics: Mathematical Aspects of Computer Science, pp. 19-32, 1967.

[FM01] S. Flake, W. Mueller; “An OCL Extension for Real-Time Constraints”. Advances in Object Modelling with the OCL; Clark and Warmer (Eds.) LNCS (Lecture Notes in Computer Science), no. 2263, Springer Verlag, 2001.

[Fow99a] M. Fowler; “Use and Abuse Cases”. Distributed Computing Magazine, 1999 (electronically available at http://www.martinfowler.com/articles.html).

[Fow99b] M. Fowler; UML Distilled: Applying the Standard Object Modeling Language. Second Edition, Addison-Wesley, 1999.

[FP78] N. Francez, A. Pnueli; “A proof method for cyclic programs”. Acta Informatica, 9, pp. 133-157, 1978.

[Fuc92] N. Fuchs; “Specification are (preferably) executable”. Software Engineering Journal, vol. 7, no. 5, pp. 323-334, September 1992.

[GGJ+00] C. Gunter, E. Gunter, M. Jackson, P. Zave; “A Reference Model for Requirements and Specifications”. IEEE Software, vol. 17 , no. 3, pp. 37-43, May/ June 2000.

[GH98] E. Gagnon, L. Hendren; “SableCC - an object-oriented compiler framework”. Proceedings of TOOLS 1998, August 1998.

[GHW85] J. Guttag, J. Horning, J. Wing; “An Overview of the Larch Family of Specification Languages”. IEEE Software, vol. 2 , no. 5, pp. 24-36, September 1985.

[Gli00] M. Glinz; “Problems and Deficiencies of UML as a Requirements Specification Language”. Proc. 10th International Workshop on Software Specification and Design, San Diego, pp. 11-22, 2000.

[GLM01] G. Génova, J. Llorens, P. Martinez; “Semantics of the Minimum Multiplicity in Ternary Associations in UML”. <<UML>> 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, 4th International Conference, Canada; Gogolla & Kobryn (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, pp. 329-341, Springer Verlag, 2001.

[GM93] M. Gordon, T. Melham; “Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, 1993.

[GPZ93] J. Gannon, J. Purtilo, M. Zelkowitz; “Software Specification : A Comparison of Formal Methods”. Intellect 1993.

[Gra94] I. Graham; “Beyond the Use Case: Combining task analysis and scripts in object-oriented requirements capture and business process re-engineering”. Proc. 13th Conference on Technology for Object-Oriented Programming Languages and Systems (TOOLS-13 Europe), pp. 203-215, Prentice Hall, 1994.

[Gri81] D. Gries; “The Science of Programming”. Springer-Verlag, 1981.

[GW89] D. Gause and G. Weinberg; “Exploring Requirements: Quality Before Design”. Dorset House 1989.

[Har88] D. Harel; “On Visual Formalisms”. Communications of the ACM, vol. 31, no. 5, pp. 514-530, May 1988.

[Har92] D. Harel; “Biting the Silver Bullet: Toward a Brighter Future for System Development”. IEEE Computer, 25(1), pp. 8-20, January 1992.

[HC91] F. Hayes, D. Coleman; “Coherent models for object-oriented analysis”. In Object-Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming, A. Paepcke (Ed.), SIGPLAN Notes 25(11), pp. 171–183, 1991.

[HC01] M. Hammer and J. Champy; “Reengineering the Corporation”. Harperbusiness, 2001.

[HCH+98] A. Hamie, F. Civello, J. Howse, S. Kent, R. Mitchell; “Reflections on the Object Constraint Language”. UML 1998 - The Unified Modeling Language: Beyond the Notation, 1st International Workshop, France; Bezivin and Muller (Eds.), LNCS (Lecture Notes in Computer Science), no. 1618, pp. 137-145, 1998.

[HDF00] H. Hussmann, B. Demuth, F. Finger; “Modular Architecture for a Toolset Supporting OCL”. UML 2000 — The Unified Modeling Language: Advancing the Standard, 3rd International Conference, UK; Kent & Evans (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, pp. 278-293, Springer Verlag, 2000.

[Hen98] B. Hendersons-Sellers; “OML: Proposals to Enhance UML”. UML 1998 - The Unified Modeling Language: Beyond the Notation, 1st International Workshop, France; Bezivin and Muller (Eds.), LNCS (Lecture Notes in Computer Science), no. 1618, Springer Verlag, 1998.

[HG97] D. Harel, E. Gery; “Executable Object Modeling with Statecharts”. IEEE Computer, vol. 30, no.7, pp. 31-42, July 1997.

[HHK98] A. Hamie, J. Howse, S. Kent; “Interpreting the Object Constraint Language”. Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98), December 2-4, 1998, Taipei Taiwan, IEEE Computer Society, 1998.

[HK98] M. Hitz, G. Kappel; “Developing with UML - Some Pitfalls and Workarounds”. UML 1998 - The Unified Modeling Language: Beyond the Notation, 1st International Workshop, France; Bezivin and Muller (Eds.), LNCS (Lecture Notes in Computer Science), no. 1618, pp. 9-20, Springer Verlag, 1998.

[HKV97] D. Harel, O. Kupferman, M. Vardi; “On the Complexity of Verifying Concurrent Transition Systems”. CONCUR'97: 8th International Conference of Concurrency Theory; LNCS (Lecture Notes in Computer Science), no. 1243, pp. 258-272, Springer Verlag, 1997.

[HN96] D. Harel, A. Naamad; “The Statemate Semantics of Statecharts”. In ACM transactions on software engineering and methodology, vol. 5, no. 4 , 1996.

[Hoa69] C. Hoare; “An axiomatic basis for computer programming”. Communications of the ACM, vol. 12, no. 10, pp. 576-583, 1969.

[Hoa72] C. Hoare; “Proof of Correctness of Data Representations”. Acta Informatica, vol. 1, pp. 271-281, 1972.

[HP85] D. Harel, A. Pnueli; “On the development of reactive systems”. In K.R. Apt, editor, Logics and Models of Concurrent Systems, pp. 477-498, Springer-Verlag, 1985.

[HU00] B. Henderson-Sellers, B. Unhelkar; “Open Modeling with UML”. Addison Wesley, 2000.

[Ja95] D. Jackson; “Structuring Z Specifications with Views”. ACM Transactions on Software Engineering and Methodology, vol. 4, no. 4, pp. 365-389, October 1995.

[Ja02] D. Jackson; “Alloy: A Lightweight Object Modelling Notation”. ACM Transactions on Software Engineering (to appear), 2002.

[Jac87] I. Jacobson; “Object-Oriented Development In an Industrial Environment”. Proceedings of OOPSLA'87, special issue of SIGPLAN Notices. Vol. 22, No. 12, pp. 183-191, 1987.

[Jac95] I. Jacobson; “Formalizing Use-Case Modeling”. Journal of Object-Oriented Programming, vol. 8, no. 3, pp. 10-14, 1995.

[Jack83] M. Jackson; “System Development”. Prentice Hall, 1993.

[Jack95] M. Jackson; “Software Requirements and Specification: a lexicon of practice, principles and prejudices”. Addison-Wesley, 1995.

[Jack00] M. Jackson; “Problem Frames and Methods: Structuring and Analyzing Software Development Problems”. Addison-Wesley, 2000.

[Jar99] M. Jarke; “CREWS: Towards Systematic Usage of Scenarios, Use Cases and Scenes”. Proceeding of 4th Internationale Tagung Wirtschaftsinformatik (WI'99), Germany, pp. 469-486, 1999.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh; “The Unified Software Development Process”. Addison Wesley, 1998.

[JC95] I. Jacobson and M. Christerson; “A Growing Consensus On Use Cases”. Journal of Object-Oriented Programming, vol. 8, no. 1, pp. 15-19, 1995.

[JCJ92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard; “Object-Oriented Software Engineering: A Use Case Driven Approach”. Addison-Wesley, 1992.

[JEJ95] I. Jacobson, M. Ericsson, Agneta Jacobson; “The Object Advantage: Business Process Reengineering with Object Technology”. Addison-Wesley, 1995.

[Jon83] C. Jones; “Tentative steps toward a development method for interfering programs”. ACM Transactions on Programming Languages and Systems, 5(4), pp. 596-619, 1983.

[Jon86] C. Jones; “Systematic Software Development Using VDM”. Prentice Hall, 1986.

[Kai95] H. Kaindl; “An Integration of Scenarios with Their Purposes in Task Modeling”. Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods & Techniques (DIS `95), pp. 227-235, ACM Press, 1995.

[Kai99] H. Kaindl; “Difficulties in the Transition from OO Analysis to Design”. IEEE Software, pp. 94-102, September/October 1999.

[KG00] D. Kulak and E. Guiney; “Use Cases: Requirements in Context”. ACM Press, Addison-Wesley, 2000.

[Kie01] J. Kienzle; “Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming”. Ph.D. Thesis EPFL-DI, no 2393, Swiss Federal Institute of Technology in Lausanne, Software Engineering Lab., 2001.

[KK98] D. Keck, P. Kuehn; “The Feature and Service Interaction Problem in Telecommunications Systems: A Survey”. IEEE Transaction on Software Engineering, vol. 24, no. 10, pp. 779-796, 1998.

[KO01] Klasse Objecten; “OCL Tools and Services”. Electronic Resource
http://www.klasse.nl/ocl/ocl-services-text.html

[Kor98] T. Korson; “The Misuse of Use Cases”. Object Magazine, May 1998.

[Kov99] B. Kovitz; “Practical Software Requirements: A Manual of Content and Style”. Manning 1999.

[Lan95] K. Lano; “Formal Object-Oriented Development”. Formal Approaches to Computer Information Technology Series, Springer, 1995.

[Lil99] S. Lilly; “Use case pitfalls: top 10 problems from real projects using use cases”. TOOLS 30, Proceedings of Technology of Object-Oriented Languages and Systems, pp. 174-183, 1999.

[LP99] J. Lilius, I. Paltor; “The semantics of UML state machines”. Turku Centre for Computer Science, Technical Report No 273, May 1999.

[LW00] D. Lefffingwell, D. Widrig; “Managing Software Requirements: A Unified Approach”. Object Technology Series, Addison-Wesley, 2000.

[LX99] J. Lee, N. Xue; “Analyzing User Requirements by Use Cases: A Goal-Driven Approach”. IEEE Software, 16 (4), pp. 92-101, July/August, 1999.

[Mas98] B. Massingill; “A Structured Approach to Parallel Programming”. PhD thesis, Computer Science Department, California Institute of Technology (Technical Report CS-TR-98-04), 1998.

[MB99] R. Malan, D. Bredemeyer; “Functional Requirements and Use Cases”. June 1999. Available at http://www.bredemeyer.com/papers.htm

[McD94] J. McDermid; “Requirements analysis: Orthodoxy, fundamentalism and heresy”. Requirements Engineering: social and technical issues; Jirotka & Goguen (eds.), pp.17-40, Academic Press, 1994.

[Mey85] B. Meyer; “On Formalism in Specifications”. IEEE Software, vol. 2, no. 1, pp. 6-26, 1985.

[Mey90] B. Meyer; “Introduction to the Theory of Programming Languages”. Prentice Hall, 1990.

[Mey92a] B. Meyer; “Applying Design by Contract”. IEEE Computer, pp. 40-51, 1992.

[Mey92b] B. Meyer; “Eiffel: The Language”. Prentice Hall, 1992.

[Mey97] B. Meyer; “Object-Oriented Software Construction”. 2nd Edition, Prentice Hall, 1997.

[MO92] J. Martin, J. Odell; “Object-Oriented Analysis and Design”. Prentice-Hall, 1992.

[Mor94] C. Morgan; “Programming from Specifications”. 2nd Edition, Prentice Hall 1994.

[MOW01] P. Metz, J. O'Brien, W. Weber; “Against Use Case Interleaving”. <<UML>> 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, 4th International Conference, Canada; Gogolla & Kobryn (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, pp. 472-486, Springer Verlag, 2001.

[NC00] G. Naumovich and L. Clarke; “Classifying Properties: An Alternative to the Safety-Liveness Classification”. Proceedings of 8th international symposium on Foundations of Software Engineering for twenty-first century applications, USA. ACM SIGSOFT Software Engineering Notes, vol. 25, no. 6, 2000.

[NE00] B. Nuseibeh, S. Easterbrook; “Requirements engineering: a roadmap”. In Finkelstein (Ed.), The Future of Software Engineering, Special Volume published in conjunction with ICSE 2000.

[NKF94] B. Nuseibeh, J. Kramer, A. Finkelstein; “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications”. In Transactions on Software Engineering volume 20, number 10, pp. 760-773, October, 1994.

[Nus94] B. Nuseibeh; “A Multi-Perspective Framework for Method Integration”. Ph.D Thesis, Department of Computing, Imperial College of Science, Technology & Medicine, London University, 1994.

[OL82] S. Owicki, L. Lamport; “Proving Liveness Properties of Concurrent Programs”. ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 455-495, 1982.

[Omg01] OMG Unified Modeling Language Revision Task Force; “OMG Unified Modeling Language Specification”. Version 1.4, May 2001. http://www.celigent.com/omg/umlrtf/

[OR97] J. Odell, G. Ramackers; “Toward a Formalization of OO Analysis”. Journal of OO Programming, pp. 64–68, July 1997.

[OSR+99] S. Owre, N. Shankar, J. Rushby, D. Stringer-Calvert; “PVS System Guide”. Computer Science Laboratory, SRI International, Menlo Park, CA, USA, September 1999.

[Par72] D. Parnas; “A Technique for Software Module Specification with Examples”. Communications of the ACM, vol. 15, no. 5, pp. 330-336, 1972.

[Pau94] L. Paulson; “Isabelle: A Generic Thereom Prover”.LNCS (Lecture Notes in Computer Science), no. 929, Springer Verlag, 1994.

[PC86] D. Parnas, P. Clements; “A Rational Design Process: How and Why to Fake It”. IEEE Transactions on Software Engineering, Feb. 1986.

[PF02] S. Palmer, J. Felsing; “A Practical Guide to Feature-Driven Development”. Prentice Hall 2002.

[Pre99] Presidents Information Technology Advisory Committee; Report to the President “Information Technology Research: Investing in Our Future”. National Coordination Office for Computing, Information, and Communications, February 1999. Available at http://www.ccic.gov/ac/report/pitac_report.pdf

[P-J99] M. Page-Jones; “Fundamentals of Object-Oriented Design in UML”. Addison-Wesley, 1999.

[Pol97] A. Pols; “Use Case Rules of Thumb: Guidelines and Lessons Learned”. Fusion Newsletter, Feb. 1997.

[Poo00] R. Pooley; “Software Engineering and Performance - a roadmap”. In Proceedings of the conference on the future of software engineering, Finkelstein (Ed.), Part of IEEE International Conference on Software Engineering, pp. 189-200, 2000.

[Pre97] R. Pressman; “Software Engineering: A Practitioner’s Approach”. 4th Edition, McGraw-Hill, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson; “Object-Oriented Modeling and Design”. Prentice Hall 1991.

[Red96] D. Redmond-Pyle; “GUI Design Techniques for OO Projects”. Object-Expert, Nov/Dec 1996.

[RG98] M. Richters, M. Gogolla; “On Formalizing the UML Object Constraint Language, OCL”. Proc. 17th Int. Conf. Conceptual Modeling (ER'98); Ling, Ram, & Lee (Ed.), LNCS (Lecture Notes in Computer Science), no. 1507, pp. 449-464, Springer Verlag, 1998.

[RG01] J. Ryser, M. Glinz; “Dependency Charts as a Means to Model Inter-Scenario Dependencies”. In Engels, Oberweis and Zündorf (Eds.): Modellierung ‘01 Germany; GI-Edition - Lecture Notes in Informatics, Vol. P-1. 71-80, 2001.

[RG02] M. Richters, M. Gogolla; “OCL: Syntax, Semantics, and Tools”. Object Modeling with the OCL, The Rationale behind the Object Constraint Language; Clark and Warmer (Eds.) LNCS (Lecture Notes in Computer Science), no. 2263, Springer Verlag, 2002.

[Ric01] M. Richters; “A Precise Approach to Validating UML Models and OCL Constraints”. PhD Dissertation, University of Bremen, Germany, 2001.

[Rie96] A. Riel; “Object-Oriented Design Heuristics”. Addison-Wesley, 1996.

[RJB98] J. Rumbaugh, I. Jacobson, G. Booch; “The Unified Modeling Language Reference Manual”. Addison-Wesley, 1998.

[Ros97] R. Ross; “The Business Rule Book: Classifying, Defining and Modeling Rules”. V 4.0, Business Rules Solutions Inc. 1997.

[RR00] S. Robertson, J. Robertson; “Mastering the Requirements Process”. Addison-Wesley, 2000.

[RS99] D. Rosenberg, K. Scott; “Use Case Driven Object Modeling With UML: A Practical Approach”. Addison-Wesley, 1999.

[Saa97] M. Saaltink; “The Z/EVES System”. In Proceedings of ZUM'97: The Z Formal Specification Notation (10th International Conference of Z Users, UK, 1997), Bowen, Hinchey, and Till (Eds.); Lecture Notes in Computer Science 1212, Springer-Verlag, pp. 72-85, 1997.

[SC92] R. Sharble, S. Cohen; “The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods.” Software Engineering Notes, vol. 18, no. 2, pp. 60-73, 1992.

[Se02a] S. Sendall; “Auction Case Study”. Electronic Resource, available at: http://lglwww.epfl.ch/research/fondue/case-studies/auction/, 2002.

[Se02b] S. Sendall; “Elevator Control System Case Study”. Electronic Resource, available at:
http://lglwww.epfl.ch/research/fondue/case-studies/elevator/, 2002.

[SF02] SourceForge; “Dresden OCL Toolkit”. Electronic Resource:
http://dresden-ocl.sourceforge.net/

[SG99] A. Simons, I. Graham; “30 Things That Go Wrong in Object Modelling with UML 1.3”. Behavioral Specifications of Businesses and Systems: Kilov, Rumpe and Simmonds (Ed.), Kluwer Academic Publishers, pp. 221-242, 1999.

[Sim99] A. Simons; “Use cases considered harmful”. TOOLS-29 Europe, Proceedings of Technology of Object-Oriented Languages and Systems, Mitchell, Wills, Bosch and Meyer (Ed.); pp. 194-203, IEEE Computer Society, 1999.

[SK98] I. Sommerville, G. Kotonya; “Requirements Engineering: Processes and Techniques”. Worldwide Series in Computer Science, John Wiley & Son, 1998.

[SM89] S. Shlaer, S. Mellor; “An object-oriented approach to domain analysis”. ACM SIGSOFT Software Engineering Notes 14 (5), pp. 66–77, 1989.

[SM92] S. Shlaer, S. Mellor; “Object Lifecycles: Modeling the World in States”. Yourdon Press Computing Series, 1992.

[Smi96] S. Smith; “Formal Justification in Requirements Engineering”. Ph.D Thesis, Department of Computer Science, University of York, 1996.

[Sof02] Software Engineering Lab. (LGL), Swiss Federal Institute of Technology in Lausanne; “The Fondue Method”. Electronic Resource, available at:
http://lglwww.epfl.ch/research/fondue/, 2002.

[Spi89] J.M. Spivey; “The Z Notation: A Reference Manual”. Prentice Hall, 1989.

[SS99a] S. Sendall, N. Guelfi, A. Strohmeier; “Fusion Applied to Distributed Multimedia System Development: the Easy Meeting Case Study”. Technical Report EPFL-DI No 99/304, Software Engineering Lab., Swiss Federal Institute of Technology in Lausanne, 1999.

[SS99b] S. Sendall, A. Strohmeier; “UML-based Fusion Analysis”. UML ‘99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, USA; France & Rumpe (Ed.), LNCS (Lecture Notes in Computer Science), no. 1723, pp. 278-291, Springer-Verlag, 1999; extended version also available as Technical Report (EPFL-DI No 99/319).

[SS00a] S. Sendall and A. Strohmeier; “From Use Cases to System Operation Specifications”. UML 2000 — The Unified Modeling Language: Advancing the Standard, 3rd International Conference, UK; Kent & Evans (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, pp. 1-15, Springer Verlag, 2000.

[SS00b] S. Sendall, A. Strohmeier; “Enhancing OCL for Specifying Pre- and Postconditions”. UML 2.0 - The Future of the UML Object Constraint Language (OCL). Workshop of UML 2000 - The Unified Modeling Language: Advancing the Standard, 3rd International Conference, UK, 2000.

[SS01a] S. Sendall, A. Strohmeier; “Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML”. <<UML>> 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, 4th International Conference, Canada; Gogolla & Kobryn (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, pp. 391-405, Springer Verlag, 2001.

[SS01b] S. Sendall, A. Strohmeier; “Merging Fine-Grained and Coarse-Grained Concurrent Behavior Specifications in UML”. Position Paper for the Workshop on Concurrency Issues in UML. UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, 4th International Conference, Canada, 2001.

[SS02] S. Sendall, A. Strohmeier; “Using OCL and UML to Specify System Behavior”. Object Modeling with the OCL, The Rationale behind the Object Constraint Language; Clark and Warmer (Eds.) LNCS (Lecture Notes in Computer Science), no. 2263, pp. 250-279, Springer Verlag, 2002.

[Ste01] P. Stevens; “On Use Cases and Their Relationships in the Unified Modelling Language”. Fundamental Approaches to Software Engineering (FASE 2001), part of the Joint European Conferences on Theory and Practice of

Software (ETAPS 2001); Hussmann (Ed.), LNCS (Lecture Notes in Computer Science), no. 2029, pp. 140-155, Springer Verlag, 2001.

[StS01] A. Strohmeier, S. Sendall; "Operation Schemas and OCL". Technical Report (EPFL-DI No 01/358), Software Engineering Lab., Swiss Federal Institute of Technology in Lausanne, 2001.

[SW98] G. Schneider, J. Winters; "Applying Use Cases: A Practical Guide". Addison-Wesley,1998.

[TD97] R. Thayer, M. Dorfman (eds.); Software Requirements Engineering. 2nd Edition, IEEE Comp. Soc. Press, 1997.

[VS99] K. van den Berg and A. Simons; "Control flow semantics of use cases in UML". Information and Software Technology, 41(10), pp. 651-659, Elsevier, 1999.

[W-B92] R. Wirfs-Brock; "Characterizing Your Objects". Smalltalk Report, vol. 2, no. 5, February, 1992. Available at: http://www.wirfs-brock.com/pages/resources.html

[W-B94a] R. Wirfs-Brock; "The Art of Designing Meaningful Conversations". Smalltalk Report, February, 1994. Available at: http://www.wirfs-brock.com/pages/resources.html

[W-B94b] R. Wirfs-Brock; "How Designs Differ". Journal of Object-Oriented Programming, Report on Object Analysis and Design, vol. 1, no. 4, November 1994. Available at: http://www.wirfs-brock.com/pages/resources.html

[WG00] A. Wegmann, G. Genilloud; "The Role of 'Roles' in Use Case Diagrams". UML 2000 — The Unified Modeling Language: Advancing the Standard, 3rd International Conference, UK; Kent & Evans (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, pp. 210-224, Springer Verlag, 2000.

[Wie98] R. Wieringa; "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques". ACM Computing Surveys, vol. 30, no. 4, December 1998.

[Win83] J. Wing; "A Two-tiered Approach to Specifying Programs". Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Wir71] N. Wirth; "Program Development by Stepwise Refinement". Communications of the ACM, vol. 14, no. 4, pp. 221-227, 1971.

[WK98] J. Warmer, A. Kleppe; “The Object Constraint Language: Precise Modeling With UML”. Addison-Wesley 1998.

[WK00] Y. Wang, G. King; “Software Engineering Processes: Principles and Applications”. CRC Press, 2000.

[WM99] R. Wirfs-Brock, A. McKean; “The Art of Writing Use Cases”. Tutorial Notes, OOPSLA 1999, Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999.

[WN97] WordNet ® 1.6, © 1997, Princeton University: ftp://clarity.princeton.edu/pub/wordnet/wn1.6unix.tar.gz

[WWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener; “Designing Object-Oriented Software”. Prentice Hall, 1990.

[Zav01] P. Zave; “Requirements for Evolving Systems: A Telecommunications Perspective”. In Proceeding of RE ‘01 (Keynote Address), 5th IEEE International Symposium on Requirements Engineering, pp. 2-9.

Appendix A:
Object Constraint Language

This appendix provides a summary of the Unified Modeling Language's constraint language called the Object Constraint Language (OCL). It details the various elements of OCL, giving an idea how one might write OCL expressions. OCL can be useful for writing invariants, pre- and postconditions, and constraints in general. This summary is based on the OCL specification, which is chapter 6 of the UML 1.4 specification [Omg01].

Introduction

OCL is a formal language for writing expressions whose principles are based on set theory and first-order predicate logic. OCL can be used in various ways to add precision to UML models beyond the capabilities of graphical diagrams. For instance, a UML class diagram may not be able to express all aspects of a specification that are necessary for it to be complete. Thus, modelers can use OCL to specify application-specific constraints in their UML models.

Traditionally, constraints on models are written in natural language. However, natural language is prone to ambiguities and imprecision. Many formal languages have been proposed for reducing these problems. The disadvantage of traditional formal languages is that they require a strong mathematical background, and they can prove difficult to use for those business and system modelers who do not have such a background. OCL has been developed to fill this gap. It is a formal language, its formal semantics is given in [Ric01], that is targeted toward being easy to read and write. It was originally developed as a business modeling language within the IBM Insurance division, and it has roots that go back to the Syntropy method [CD94].

Two common uses of OCL are the statement of system invariants on class models and the definition of pre- and postconditions for operations/methods. OCL is a declarative language. An OCL expression has no side effects, i.e. its evaluation cannot alter the state of the corresponding entity. OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. OCL provides elementary types, like Boolean, Integer, etc., includes collections, like Set, Bag, and Sequence, and has an assortment of predefined operators on these basic types. It also allows user-defined types which can be any type defined in a UML model, in particular classes. OCL uses an object-oriented-like notation to access properties, attributes, and for applying operators.

The Language

In this section, I describe the different elements of OCL. The description is based on chapter 6 of the UML specification [Omg01]. For the purposes of explaining the different aspects of OCL, figure A.1 shows a class diagram that will be used as the reference example throughout this summary.

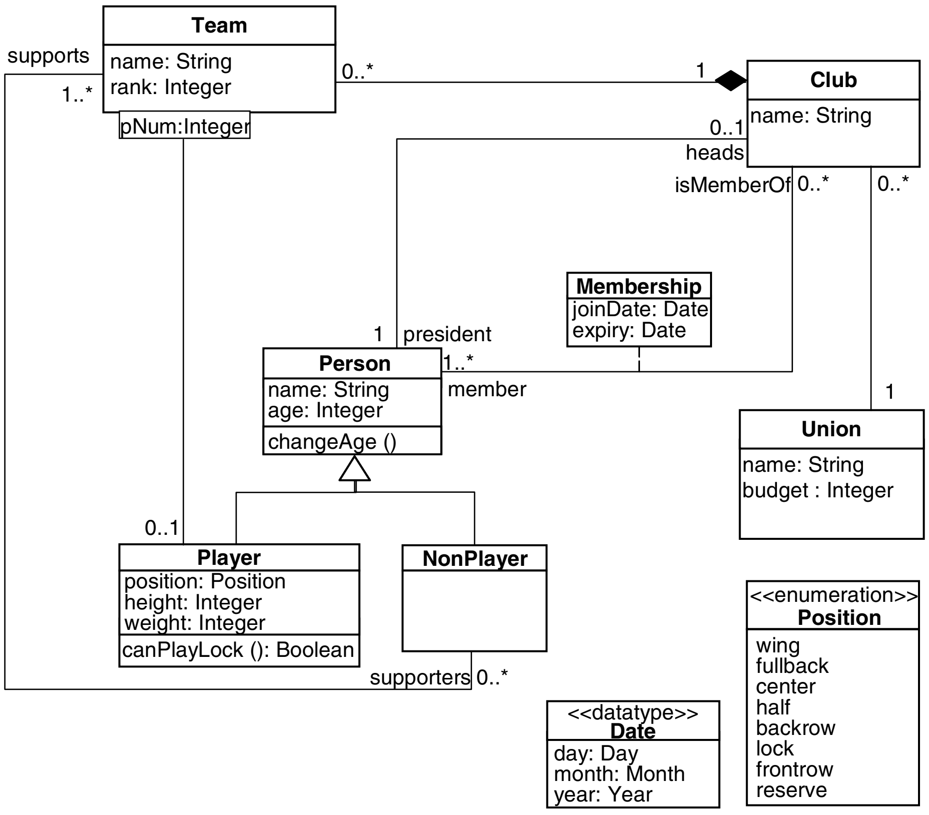

Figure A.1: Example UML Class Diagram

Context

Each OCL expression is written in the context of an instance of a specific type. The predefined identifier self refers to this instance. In OCL, the context is explicitly declared[1]. For example, we could specify an invariant for all persons, constraining the model shown in figure .

```
context Person
   inv: self.age >= 15
```

This is an invariant for all objects of the class Person; it states that a person is always greater than or equal to 15. self.age is a navigation expression that results in the value of the age attribute of the person denoted by self. It is equivalent to the following.

```
context p: Person
   inv: p.age >= 15
```

This form gives an explicit name, p, rather than self. It is also equivalent to the following.

```
context p: Person
   inv noUnderAgePerson: p.age >= 15
```

where a name is given to the invariant, noUnderAgePerson. OCL also allows you to specify pre- and postconditions. They have the following form.

```
context Player::canPlayLock (): Boolean
   pre: self.weight > 100
   post: result = (self.height > 190)
```

In this example, the context is Player, where the precondition and postcondition of it's method canPlayLock is specified. The method does not have any parameters and its return type is Boolean. Note that result is a predefined identifier of OCL that represents the result of a function. The type of result matches the return type, which is suffixed at the end of the parameter list of the method, i.e., Boolean in the case of the canPlayLock method.

The precondition of the canPlayLock method states that the weight of the player is assumed to be over 100 (kilograms) and the postcondition states that the result of the method is true if the height of the player is over 190 (centimeters), otherwise it is false.

Types

Each OCL expression is written in the context of a UML model. All types from the UML model can be used in OCL. Also, OCL has a number of predefined elementary types: Boolean, Integer, Real, String, Enumeration, where each one has a set of predefined features and properties. The first four are similar to the primitive types that you would find in a language like Java. Enumerations are Datatypes in UML and have a name. An enumeration defines a number of enumeration literals, that are the possible values of the enumeration. Within

1. Note that in the case of an Operation Schema, the context is implicit: it is an instance of the system that owns the operation.

OCL, one can refer to the value of an enumeration. For example, given an enumeration type Position (see figure) with values wing, fullback, center, half, backrow, lock, frontrow, or reserve, you can use them in the following way.

```
context Player
  inv: self.position = Position::frontrow
```

This invariant would mean that all players play in the frontrow position. Note the enumeration is accessed by prefixing the literal with its type, followed by "::".

All types need to conform to a number of rules defined in [Omg01]. OCL also has a number of collection types. The collection types play an important role in OCL expressions, as we will see in the discussion of navigation expressions in OCL. Collection is the abstract supertype of all collection types in OCL. Collection has three concrete subtypes: Set, Bag and Sequence. Set is the mathematical set; it does not contain duplicate elements and is unordered. Bag is like a set, i.e., it does not have an order, but it may contain duplicates (i.e., the same element may be in a bag more once). Sequence is like a bag, i.e., it may contain duplicates, but the elements of a sequence are ordered. In OCL, a collection can be specified by literals. Below are several examples of collection literals. Note that comments in OCL are written following two successive dashes (minus signs). Everything immediately following the two dashes up to and including the end of line is part of the comment.

```
Set {1, 2, 5, 88} -- a set of integers
Set {'Josh', 'Richard', 'Krusher'} -- a set of strings
Sequence {1, 3, 45, 2, 3} -- a sequence of integers
Sequence {1..10} -- a sequence of integers
Bag {1, 3, 45, 2, 3} -- a bag of integers
Bag {1, 2, 3, 3, 45} -- an identical bag to the one directly above
```

OCL has many predefined operations for its collections. Consistent with the definition of OCL as an expression language, collection operations never change the collection that it is applied to, instead they result in values that indicate a particular property of the collection or they project the result into a new collection. Some collection operations are specific to a particular collection type, e.g., Set, and some collection operations can be used for all collection types. See [WK98, Omg01] for a description of all collection operations. Below, I briefly explain and give examples of a few of the collection operations available in OCL. Note that the arrow operator is used only on collections, in postfix style, i.e., the operation following the arrow is applied to the previous "term".

Selecting in a collection: select

```
colOfAllTeams -> select (t | t.rank <= 5)

-- t iterates through the teams of the colOfAllTeams
-- selecting all those who are ranked in the top 5.
```

Excluding from a collection: reject

```
colOfPlayers -> reject (p | p.canPlayLock())
-- the collection of all players of colPlayers that can not
-- play the lock position.
```

Collecting the values yielded by an expression: collect

```
colOfPersons -> collect (p | p.age)
-- ages are collected, not persons. The result is a bag of Integers.
```

Projection on an attribute can be used as a shorthand for collect

```
colOfPersons.age
-- equivalent to the expression above
```

Conversion of a bag to a set, by eliminating duplicates: asSet

```
bagOfIntegers -> asSet ()
-- removes duplicates and type converts to a set of integers
```

All objects meet the expression (universal quantification): forAll

```
colOfPlayers -> forAll (p | p.weight > 60)
-- true if all players in colOfPlayers are heavier than 60 (Kilograms).
```

Existence of an object that meets the expression (existential quantification): exists

```
colOfPlayers -> exists (p | p.position = Position::center)
-- true if at least one player in colOfPlayers plays the center position.
```

Asserting that an element is a member of a collection: includes

```
colOfPlayers -> includes (josh);
-- true if josh is a member of colOfPlayers
```

Asserting that an element is not a member of a collection: excludes

```
colOfPlayers -> excludes (david);
-- true if david is not a member of colOfPlayers
```

Existence of a single object: one

```
colOfPlayers -> one (p | p.height > 200)
-- true if and only if one player of colOfPlayers
-- is taller than 200 (centimeters).
```

Selecting an object: any

```
colOfPlayers -> any (p | p.weight < 80)
-- results in any player of colOfPlayers that is under 80 (kilograms).
```

Size of the collection: size

```
colOfPlayers -> size () = 5
-- results in true if there are 5 players in colOfPlayers.
```

State of emptiness of the collection: isEmpty

```
colOfPlayers -> isEmpty ()
-- results in true if there are no players in colOfPlayers
-- (i.e., the collection is empty).
```

OCL Operators

The precedence order for the operations, starting with highest precedence, in OCL is:

- @pre
- dot and arrow operations: '.' and '->'
- unary 'not' and unary minus '-'
- '*' and '/'
- '+' and binary'-'
- 'if' ... 'then' ... 'else' ... 'endif'
- '<', '>', '<=', '>='
- '=', '<>'
- 'and', 'or' and 'xor'
- 'implies'

Parentheses '(' and ')' can be used to change precedence.
For example, the expression in the following invariant,

```
context Player
  inv: self.position = Position::half implies self.weight < 100
          and self.age > 300 / 10 or not self.canPlayLock() <> true
```

should be interpreted in the following way (using parenthesis to show precedence):

```
  inv: ((self.position) = Position::half) implies ((((self.weight) < 100)
          and ((self.age) > (300 / 10))) or ((not (self.canPlayLock())) <> true)))
```

Accessing Properties of Objects

OCL expressions can refer to types, classes, interfaces, and datatypes, and to all properties of objects. A property is one of the following: an Attribute, an AssociationEnd, an Operation or a Method with isQuery being true (the rule guarantees that OCL expressions have no side-effects). For example, josh.name accesses the attribute name of the person josh, and josh.canPlayLock () accesses the canPlayLock method of the person josh. Starting from a specific object, we can navigate an association to refer to other objects. To do so, we refer to the role name of the opposite association-end; it is written in the following form:

```
object.rolename
```

The value of this expression is the set of objects on the other side of the association. When a rolename is missing, the name of the type, starting with a lowercase character is used instead. For example, to find all the teams in a given club c, we could write the following expression:

```
c.team
```

this expression accesses the association role at the opposite end of the composition association, and results in the collection of objects on the other end of the association.

The objects of an association class can be accessed by navigation using the name of the association class, starting with a lowercase character. For example, to find all the membership objects that the person, josh, has with his clubs, we could write the following expression:

```
josh.membership
```

Single navigation results in a set, i.e., one application of the dot operator. A combined navigation results in a bag[1]. Navigation over associations adorned with {ordered} results in a sequence. If the multiplicity of the association end is 0..1 then the expression results in an object. However, such an expression can be treated like it results in a set as well. For example, given a person p, we could write the following expression:

```
p.heads -> notEmpty () implies p.heads.name = 'Canterbury'
```

this expression shows the "double life" of the expression p.heads.

A qualified association can be navigated like a normal association, or one can navigate it by providing the key. For example, given a team allBlacks, and a key 7, which is a position number (pNum), we could write the following expression.

```
allBlacks.player[7]
```

In this example, the result would have the same result as navigating an association end with multiplicity 0..1; i.e., we could treat it as an object or a set.

Complex Expressions

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to an object or value of a specific type, or to a collection of objects of a specific type. After obtaining a result, one can always apply another property to get a new result value. Therefore, each OCL expression can be read and evaluated left-to-right. For example, given a union u, we could write the following expression:

```
u.club.team -> select (t | t.rank <= 5) -> asSet ().player[15] -> forall (p |
    p.weight > 100 and p.heads->isEmpty ())
```

Breaking up the explanation of this expression, the select operation results in the bag all teams that are ranked in the top 5 of all the clubs of the union u. The resulting bag has the asSet operation applied to it, resulting in set. To this set, an implicit collect operation is applied; it uses the key 15 for the qualified association. The resulting set of players that have a position number of 15 has a forall operation applied to it. The forall operation results in true if all the players in the set have a weight greater than 100 (kilograms) and are not the president of a club.

Predefined Properties on All Objects

OCL defines a number of predefined properties for objects. These include:

1. This is true only if more than one of the association ends (used in the navigation expression) has an upper range of multiplicity greater than 1.

Equality: object = (object2: OclAny): Boolean
results in true if object is the same object as object2. For example, given p1 and p2 of class Person, we could write the following expression:

```
p1 = p2
```

which would result in true if p1 references the same person as p2, false otherwise.

Type equality: oclIsTypeOf (t: OclType): Boolean
results in true if the target object is of type t. For example, given josh, we could write the following expression:

```
josh.oclIsTypeOf (Player)
```

which would result in true if josh is of class Player, false otherwise.

Subtype equality: oclIsKindOf (t: OclType): Boolean
results in true if the target object is of type t or one of the supertypes. For example, given josh is of class Player, then the following expression would result in true.

```
josh.oclIsTypeOf (Person)
```

Object creation: oclIsNew (): Boolean
results in true if the object is newly created. It only makes sense in a postcondition. For example, as part of a postcondition, we could write the following expression:

```
blackTeam.oclIsNew () -- blackTeam is an object of class Team
```

which would result in true if blackTeam was created by the corresponding operation.

Preconditions and Postconditions

In a postcondition, an expression or sub-expression can be evaluated at the time immediately before or immediately after the execution of the operation/method. Thus, one can refer to two values for each property of an object:

- the value of the property at the start of the operation,
- the value of the property upon completion of the operation.

The value of a property in a precondition is always the one at the start of the operation. In a postcondition, to refer to the value of a property at the start of the operation, one has to postfix the property name with the suffix @pre. For example, the incAge method of Person, which increments the age of the target object, could be specified using OCL in the following way.

```
context Person::incAge ()
   pre: self.age > 15
   post: self.age = self.age@pre + 1
```

The incAge method does not have any parameters and it does not have a return type. The precondition states that the age (attribute) of the person is greater than 15 and the postcondition states that the age (attribute) of the person (denoted by self) is one greater after the method than it was before the execution of the method, where the post-value of age is self.age and its pre-value is self.age@pre. Note that navigation expressions are also subject to pre- and post-values. For example, given john is of class nonPlayer in a postcondition:

```
  post: john.supports@pre.club->forall(c | c.name@pre.substring(1, 1) = 'W')
```

this would evaluate to true if all the teams that john supported, before the execution of the operation, have clubs, where the navigation is evaluated after execution, that have names, where the navigation is evaluated before execution, that start with the letter 'W'. In this way, combined navigations can get quite confusing if one does not take care.

Evaluation Semantics

The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation. In the context of pre- and postconditions, this means that OCL is only capable of specifying sequential operations. Note that in Chapter 6 I propose a model that allows truly concurrent operations.

Appendix B:
Tool Support

This appendix provides a brief overview of the LGL Schema Compiler, which is a tool that is capable of syntax and type checking Operation Schemas (as defined in Chapter 6). It also provides the grammar for Operation Schemas, which is taken directly from the tool specification.

Overview of LGL Schema Compiler

The LGL Schema Compiler is a syntax and type checker for Operation Schemas. It takes as input an XMI file, generated by Rational Rose (Unisys XMI generator add-in), and a text file, which holds an Operation Schema. The tool indicates whether the supplied Operation Schema is syntax correct, according to the grammar shown below, and if this is the case, it checks to see that it is type correct using the supplied XMI file to retrieve type information. Note that the XMI file represents the Concept Model of the corresponding system (drawn graphically in Rose).

The LGL Schema Compiler is based upon the Dresden OCL Toolkit [SF02]. The Dresden OCL Toolkit is a modular toolkit for OCL support. The Dresden OCL Toolkit is based on the OCL compiler developed by Frank Finger at the Dresden University of Technology [HDF00], whose parser is built using the tool SableCC [GH98].

The task of developing the LGL Schema Compiler involved:

- Extracting type information from the XMI file, which involved interpreting the extensions to UML made by the Concept Model (i.e., stereotypes).
- Updating the OCL 1.3 grammar of the Dresden OCL Toolkit to be OCL 1.4 compliant.

- Defining a grammar for Operation Schemas, which involved dealing with the proposed language extensions to OCL, Operation Schema clauses, Parameterized Predicates, and Functions (which are all described in Chapter 6).
- Type checking an Operation Schema by walking a syntax tree generated by SableCC code (given the grammar file that is similar to the one shown below).

Operation Schema Grammar

The following grammar is based on the one used for the LGL Schema Compiler. It is presented in two parts: Tokens and Productions, following the SableCC input grammar requirements.

Tokens

```
dot = '.';
arrow = '->';
not = 'not';
mult = '*';
div = '/';
plus = '+';
minus = '-';

colon=':';

operation='Operation';
description='Description';
notes='Notes';
use_cases='Use Cases';
scope='Scope';
messages='Messages';
new='New';
aliases = 'Aliases';
t_pre = 'Pre';
t_post = 'Post';

throws='Throws';
t_time = 'pre';
t_pre_au = 'preAU' ;
t_post_au = 'postAU';
t_rd= 'rd';
t_rely = 'rely';
t_fail = 'fail';
t_endre = 'endre';
fun = 'Function';
predicate = 'Predicate';
body = 'Body';
shared  = 'Shared';
import ='import';
is = 'Is';
context = 'context';
t_inv = 'inv';

equal = '=';
n_equal = '<>';
lt = '<';
gt = '>';
lteq = '<=';
gteq = '>=';

and = 'and';
or = 'or';
```

```
xor = 'xor';
implies = 'implies';

l_par = '(';
r_par = ')';
l_squarebracket = '[';
r_squarebracket = ']';
l_brace = '{';
r_brace = '}';

semicolon = ';';
dcolon = '::';
comma = ',';
channel = '#';
at = '@';
bar = '|';
ddot = '..';
apostroph = ''';
assign = '::=';

t_let = 'let';
t_in = 'in';
t_if = 'if';
t_then = 'then';
t_elsif = 'elsif';
t_else = 'else';
endif = 'endif';
t_and = '&';

t_set = 'Set';
t_bag = 'Bag';
t_sequence = 'Sequence';
t_collection = 'Collection';

bool = 'true' | 'false';

impl_arrow = '=>' ;
all = 'all' ;

simple_type_name = uppercase (lowercase | digit | uppercase | '_')* ;

name = lowercase (lowercase | digit | uppercase | '_')*;

int = number;
real_num = number '.' number;
string_lit = ''' s_char_sequence? ''';
back_slash = '\' ;
```

Productions

```
in = include_file* choice* opn_schema? ;

include_file = import file_name semicolon ;

choice=
    funct |
    funct_body |
    pred;

file_name =
    simple_type_name colon back_slash name file_name_tail* back_slash name dot? name?;

file_name_tail = back_slash name ;

opn_schema=
    opn_decl
    desc_decl?
    notes_decl?
    use_cases_decl?
    scope_decl?
    shared_decl?
    messages_decl?
```

```
        new_decl?
        aliases_decl?
        pre_decl?
        post_decl? ;

// Operation clause -- start
    opn_decl=
        operation colon system_class_name dcolon operation_name l_par parameter_list? r_par semicolon;

    parameter_list=
        parameter parameter_list_tail*;
    parameter_list_tail=
        comma parameter;

    parameter=
        entity_declaration |
        actor_instance_declaration;

    actor_instance_declaration=
        name colon actor_class_name;

    system_class_name=
        simple_type_name;

    operation_name=
        name;

    actor_class_name=
        simple_type_name;

    entity_declaration=
        name colon type_name;
// Operation clause -- end

// Description and Notes clauses -- start
    desc_decl=
        description colon text_body;

    notes_decl=
        notes colon text_body;

    text_body=
        line semicolon;
// Description and Notes clauses -- end

// Use Cases clause -- start
    use_cases_decl=
        use_cases colon use_cases_list?;

    use_cases_list =
        use_case_with_semicolon*;

    use_case_with_semicolon =
        use_case_numbers semicolon;

    use_case_numbers =
        simple_type_name |
        simple_type_name dcolon l_brace uc_number_list* r_brace ;

    uc_number_list =
        int semicolon;
// Use Cases clause -- end

// Scope clause -- start
    scope_decl=
        scope colon name_list?;

    name_list=
        name_list_element semicolon name_list_tail*;

    name_list_tail=
        name_list_element semicolon;

    name_list_element=
        simple_type_name |
        association |
        object_name colon simple_type_name;

    object_name=
        name;

    association=
        simple_type_name? l_par role_name_component? simple_type_name
            comma role_name_component? simple_type_name association_tail* r_par  |
        association_name;

    role_name_component=
        role_name colon;

    association_tail=
        comma role_name_component? simple_type_name;

    role_name=
        name;

    association_name=
        simple_type_name;

// Scope clause -- end


//Shared clause --start

    shared_decl =
        shared colon shared_item_with_semicolon* ;

    shared_item_with_semicolon =
        shared_item semicolon ;

    shared_item =
        association |
        simple_type_name ;

//Shared clause --end


// Messages clause -- start

    messages_decl=
        messages colon actor_with_msgs_list;

    actor_with_msgs_list =
        actor_with_msgs_with_semicolon*;

    actor_with_msgs_with_semicolon =
        actor_with_msgs semicolon;

    actor_with_msgs =
        actor_class_name dcolon l_brace msg_name_list* r_brace ;

    msg_name_list =
        msg_name throws_with_exceptions? semicolon;

    throws_with_exceptions =
        throws exception_msgs;

    exception_msgs =
        msg_name exception_msgs_tail*;

    exception_msgs_tail =
        comma msg_name ;

    msg_name=
        simple_type_name ;
// Messages clause -- end


// New clause -- start
    new_decl=
        new colon new_item_list?;
    new_item_list=
        new_item semicolon new_item_list_tail* ;
    new_item_list_tail=
        new_item semicolon;
    new_item =
        entity_declaration;
// New clause -- end


// Aliases clause -- start
    aliases_decl=
        aliases colon item_list?;
    item_list=
        item semicolon item_list_tail* ;
    item_list_tail=
        item semicolon;
    item=
        name colon type_name is expression;
// Aliases clause -- end


// Pre and Post clauses -- start
    pre_decl=
        t_pre colon ocl_body semicolon ;
    post_decl=
        t_post colon ocl_body semicolon ;
// Pre and Post clauses -- end


    signature =
        l_par parameter_list? r_par ;


// Functions and parameterized Predicates definitions -- start
    funct=
        fun colon name signature colon type_name semicolon;
    funct_body=
        fun body colon name signature colon type_name semicolon dec? t_post colon name equal expression
semicolon;
    dec=
        aliases colon item_list;
    pred=
        predicate colon name signature semicolon dec? body colon ocl_body semicolon;
// Functions and parameterized Predicates definitions -- end


// The rest is the OCL part of the productions
    ocl_body=
        expression;
    expression =
        let_expression* logical_expression ;

    if_expression =
        t_if expression t_then expression else_branch;

    else_branch =
        t_else expression endif |
        t_elsif expression t_then expression else_branch |
        endif;

    rely_expr =
        t_rely expression t_then ocl_body fail_branch* last_fail? t_endre ;

    fail_branch  =
        t_fail expression t_then ocl_body ;

    last_fail =
        t_fail ocl_body;

    logical_expression =
        logical_expression logical_operator relational_expression |
        relational_expression ;

    relational_expression =
        relational_expression relational_operator additive_expression |
        additive_expression ;

    additive_expression =
        additive_expression additive_operator multiplicative_expression |
        multiplicative_expression ;

    multiplicative_expression =
        multiplicative_expression multiplicative_operator unary_expression |
        unary_expression ;

    unary_expression =
        unary_operator postfix_expression |
        postfix_expression |
        composite_value;


// Composite Values -- start

    composite_value =
        aggregate ;

// Composite Values -- end


// Aggregates -- start

    aggregate =
        qualifier_aggregate?  l_par aggregate_item? aggregate_tail* r_par ;

    qualifier_aggregate =
        simple_type_name apostroph ;

    aggregate_tail =
        comma aggregate_item ;

    aggregate_item =
        name_with_impl_arrow expression;

    name_with_impl_arrow =
        name impl_arrow ;
// Aggregates -- end


    postfix_expression =
        primary_expression postfix_expression_tail* all_part? |
        literal;

    all_part =
        dot all ;
```

```
postfix_expression_tail =
    postfix_expression_tail_begin feature_call ;

postfix_expression_tail_begin =
    dot |
    arrow ;

primary_expression =
    literal_collection |
    path_name time_expression? qualifiers? feature_call_parameters? |
    type_name |
    l_par expression r_par |
    if_expression |
    rely_expr;

let_expression =
    t_let name let_expression_type_declaration? equal expression t_in ;

let_expression_type_declaration =
    colon type_name ;

literal =
    string_lit |
    real_num |
    int     |
    bool  ;

literal_collection =
    collection_type l_brace expression_list_or_range? r_brace ;

expression_list_or_range =
    expression expression_list_or_range_tail? ;

expression_list_or_range_tail =
    expression_list_tail+ |
    ddot expression ;

expression_list_tail =
    comma expression ;

feature =
    type_name |
    feature_call ;

feature_call_parameters =
    l_par r_par |
    l_par expression fcp_helper* r_par |
    l_par declarator? actual_parameter_list? r_par ;

fcp_helper =
    comma expression  |
    colon type_name |
    semicolon name colon type_name equal expression |
    bar expression ;

feature_call =
    path_name time_expression? qualifiers? feature_call_parameters? ;


declarator =
    name declarator_tail* declarator_type_declaration?  bar |
    name declarator_type_declaration semicolon name declarator_type_declaration equal expression bar;

qualifiers =
    l_squarebracket actual_parameter_list r_squarebracket ;

declarator_tail =
    comma name ;

declarator_type_declaration =
    colon type_name ;

path_type_name =
    path_type_name dcolon simple_type_name |
    simple_type_name ;

type_name =
    path_type_name   |
    collections  ;

collections =
    collection_type l_par postfix_expression r_par;

collection_type =
    t_set |
    t_bag |
    t_sequence |
    t_collection;

path_name =
    path_type_name dcolon name |
    name ;

time_expression =
    at times ;

times =
    t_time |
    t_pre_au |
    t_post_au |
    t_rd ;

actual_parameter_list =
    expression actual_parameter_list_tail* ;

actual_parameter_list_tail =
    comma expression ;

logical_operator =
    and |
    t_and |
    or |
    xor |
    implies;

relational_operator =
    equal |
    n_equal |
    gt |
    lt |
    gteq |
    lteq ;

additive_operator =
    plus |
    minus;

multiplicative_operator =
    mult |
    div;

unary_operator =
    minus |
    not;

actual_parameters=
    name actual_parameters_tail*;

actual_parameters_tail=
    comma name;
```

Appendix C:
Parameterized Predicates and Functions

This appendix contains the declarations and definitions of Parameterized Predicates and Functions that are used in the ANZAC specification for the elevator control system (Chapter 9).

Parameterized Predicates

Predicate: ElevatorControl::requestDoorToOpen_p (openMsg: OpenDoor);
Body:
  openMsg.oclIsNew () &
  self.door.*sent* (openMsg);

Predicate: ElevatorControl::requestCabinToStop_p (stopMsg: StopAtNextFloor);
Body:
  stopMsg.oclIsNew () &
  self.motor.*sent* (stopMsg);

Predicate: ElevatorControl::informSourceThatReqServiced_p (source: ReqSource,
      serviceReq: ServicedRequest, f: Floor)
Body:
  serviceReq.oclIsNew ((servicedFlr => f)) &
  source.*sent* (serviceReq);

Predicate: ElevatorControl::cabinMovedByOneFloor_p (cab: Cabin, newFlr: Floor);
Aliases:
  cabinIsOneFloorHigher: Boolean **Is** cab.currentFloor.num + 1 = newFlr.num;
  cabinIsOneFloorLower: Boolean **Is** cab.currentFloor.num - 1 = newFlr.num;
Body:
  (cab.movement = Movement::up **and** cabinIsOneFloorHigher) or
  (cab.movement = Movement::down **and** cabinIsOneFloorLower);

```
Predicate: ElevatorControl::serveRequest_p (cab: Cabin, sourceFlr: Floor, targetFlr: Floor,
                                            req: Request, upMsg: GoUp, downMsg: GoDown);
Body:
  cab.currentRequest = req &
  requestCabinToMove_p (cab, sourceFlr, targetFlr, upMsg, downMsg);

Predicate: ElevatorControl::storeRequest_p (f: Floor, req: Request, source: ReqSource);
Body:
  req.oclIsNew () &
  req.targetFloor = f &
  req.source = source;

Predicate: ElevatorControl::requestCabinToMove_p (cab: Cabin; srcFlr: Floor, tgtFlr: Floor,
                                                  upMsg: GoUp, downMsg: GoDown);
Body:
  if srcFlr.num < tgtFlr.num then
    upMsg.oclIsNew () &
    self.motor.sent (upMsg) &
    cab.movement = Movement::up
  elsif srcFlr.num > tgtFlr.num then
    downMsg.oclIsNew () &
    self.motor.sent (downMsg) &
    cab.movement = Movement::down
  else -- srcFlr.num = tgtFlr.num
    false -- this should NEVER occur
  endif;
```

Parameterized Predicates that use Synchronization Constraints

```
Predicate: ElevatorControl::serveRequestV2_p (cab: Cabin, sourceFlr: Floor, targetFlr: Floor,
                                              req: Request, upMsg: GoUp, downMsg: GoDown);
Body:
  cab.currentRequest@postAU = req &
  requestCabinToMoveV2_p (cab, sourceFlr, targetFlr, upMsg, downMsg);

Predicate: ElevatorControl::storeRequestV2_p (f: Floor, req: Request, source: ReqSource);
Body:
  req.oclIsNew () &
  req.targetFloor@postAU = f &
  req.source@postAU = source;

Predicate: ElevatorControl::requestCabinToMoveV2_p (cab: Cabin; srcFlr: Floor, tgtFlr: Floor,
                                                    upMsg: GoUp, downMsg: GoDown);
Body:
  if srcFlr.num < tgtFlr.num then
    upMsg.oclIsNew () &
    self.motor.sent (upMsg) &
    cab.movement@postAU = Movement::up
  elsif srcFlr.num > tgtFlr.num then
    downMsg.oclIsNew () &
    self.motor.sent (dnMsg) &
    cab.movement@postAU = Movement::down
  else -- srcFlr.num = tgtFlr.num
```

```
      false -- this should NEVER occur
   endif;

Predicate: ElevatorControl::requestCabinToMoveOrDoorToOpen_p (cab: Cabin, srcFlr: Floor,
                    tgtFlr: Floor, upMsg: GoUp, downMsg: GoDown, openMsg: OpenDoor);
Body:
   if srcFlr.num = tgtFlr.num then
      openMsg.oclIsNew () &
      self.door.sent (openMsg) &
      cab.doorState@postAU = DoorState::open
   else
      requestCabinToMoveV2_p (cab, srcFlr, tgtFlr, upMsg, downMsg)
   endif;
```

Functions

```
Function: ElevatorControl::calcNextRequest_f (cab: Cabin): Request;
-- results in the most eligable request for the cabin to serve
-- (could be just the request that is already the current one)

Function: ElevatorControl::makeStopAccordingToPolicy_f (cab: Cabin,
                                        floorReached: Floor): Boolean;
-- results in true if there is a request for the floor that obeys the current stopping policy of
-- the system (adding something like a "mode" attribute to the cabin object would allow
-- dynamic changes to this strategy)
```

Curriculum Vitae

Shane Sendall is a New Zealand and Australian citizen.

	Education
1994 - 1996	Computer science studies at the University of Queensland, Australia; graduated 1996 with Bachelor of Science.
1997	Computer science studies at the University of Queensland, Australia; graduated 1997 with Honours Degree (Class I) in Computer Science.
	Research
1997	Binary Translation: support for semantic analysis of machine instructions; development of a Java byte-code decoder and the analysis of the Java byte-codes for suitability as an intermediate representation.
1997	Specification-Based Testing: development of translation methods for taking Object-Z specifications to C++ test oracles.
1998 - 2002	Ph.D. thesis in the field of software development methods and software modeling.
	Teaching
1996	Tutor for Program Verification course, given to 2nd year computer science students.
1997	Tutor for Software Specification course, given to 2nd year computer science students.
1997	Tutor for Declarative and Functional Programming course, given to 2nd year computer science students.
1998 - 2002	Assistant of the Software Engineering Project for 3rd year computer science students.
2001	Two day course to private sector on “Requirements Elicitation and Analysis with Use Cases” at Swiss Federal Institute of Technology in Lausanne.
2001 - 2002	Guest Lecturer for Software Engineering Course, given to 3rd year computer science students.
	Other Activities
1994	Rugby Union: Member of the Queensland Country Champions.
1996	Rugby Union: Captain of Souths Club 1st XV (Sunshine Coast).