Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *12 juillet 2019* **par :**

ALEXANDR NAUMCHEV

Exigences orientées objets dans un cycle de vie continu
Seamless Object-Oriented Requirements

JURY

M. YAMINE AIT-AMEUR	Président du Jury
MME SUSANNE GRAF	Examinatrice
MME CATHERINE DUBOIS	Rapporteure
MME ELISABETTA DI NITTO	Rapporteure
M. JEAN-MICHEL BRUEL	Directeur de thèse
M. BERTRAND MEYER	Co-directeur de thèse

École doctorale et spécialité :
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse
Unité de Recherche :
IRIT: Institut de Recherche en Informatique de Toulouse
Directeur(s) de Thèse :
M. Jean-Michel Bruel et *M. Bertrand Meyer*
Rapporteurs :
Mme Catherine Dubois et *Mme Elisabetta Di Nitto*

To Mary

Acknowledgement

Many people were helping me in different ways while I was working on the thesis. As a human being with an imperfect memory, however, I can only highlight some of them here.

Bertrand Meyer, for proposing an amazingly interesting research topic. Bertrand Meyer and Jean-Michel Bruel, for continuously correcting my path and relentlessly exercising all the crazy ideas coming out of my head. Bertrand Meyer, Manuel Mazzara and Victor Rivera from Innopolis University; Jean-Michel Bruel, Florian Galinier and Sophie Ebersold from the University of Toulouse, for actively helping me grow my research ideas into publications. Manuel Mazzara, Andrey Sadovykh and Sergey Masiagin from Innopolis University, for supporting my research administratively and financially. Mansur Khazeev and Hamna Aslam from Innopolis University; Jean-Michel Bruel, Sophie Ebersold and Florian Galinier, for giving me the luxury of having them in the room when I was rehearsing the thesis defense. Alexander Chichigin and Larisa Safina from Innopolis University, for always willing to give their valuable technical advice. Elisabetta di Nitto from Politecnico di Milano and Catherine Dubois from ENSIIE, for providing their insightful feedback on the dissertation that allowed me to better understand the directions for further improvement. Susanne Graf from VERIMAG, for the enormous amount of technical feedback on the dissertation that allowed me to greatly improve the writing and see future research directions. Tanya Stanko from Innopolis University, for introducing me to Manuel and Bertrand and thus initiating our fruitful collaboration. Inna Baskakova, for always smiling and helping with tons of paperwork appearing along the way.

My wife Mary, for her unimaginable love and patience. My kids Mark and Kate, for giving rest to my brain, heart and soul. My parents Vladimir and Galina, my brother Yuri, for constantly expressing their pride of my work. My grandfather Georgy, for being an in-memory example of how passionate I should be about what I do. My grandfather Ruben, for being an invisible example of how brave I should be in what I do. My brother Nikolay, for being an example of how persistent I should be in what I do. With their help I can achieve everything that I can imagine.

I would like to book a special place in this chapter, and in my heart, to the conference and journal reviewers who rejected my submissions, thus forcing me to improve, improve and improve.

The most important property of a program is whether it accomplishes the intention of its user.

Sir Charles Antony Richard Hoare

Résumé

L'évolution constante des besoins des clients et des utilisateurs exige une réponse rapide de la part des équipes logicielles. Cela crée une forte demande pour un fonctionnement sans rupture des processus logiciels. L'intégration, la livraison et le déploiement continus, également connus sous le nom de DevOps, ont fait d'énormes progrès en rendant les processus logiciels réactifs au changement. Ces progrès n'ont toutefois eu que peu d'effets sur les exigences en matière de logiciels. Aujourd'hui, la plupart des besoins sont exprimés en langage naturel. Cette approche a un grand pouvoir expressif, mais au détriment d'autres aspects de la qualité des exigences telles que la traçabilité, la réutilisabilité, la vérifiabilité et la compréhensibilité. Le défi est ici d'améliorer ces aspects sans sacrifier l'expressivité.

Bertrand Meyer, dans sa méthode multi-exigences, relève ce défi et propose d'exprimer les besoins individuels en trois couches: sous-ensemble déclaratif d'un langage de programmation orienté objet, langage naturel et notation graphique. Cette approche a motivé et inspiré les travaux de la présente thèse. Alors que l'approche multi-exigences se concentre sur la traçabilité et la compréhensibilité, l'approche Seamless Object-Oriented Requirements (SOOR) présentée dans cette thèse prend en compte la vérifiabilité, la réutilisabilité et la compréhensibilité.

Cette thèse explore l'hypothèse de Martin Glinz selon laquelle, pour soutenir la continuité, les exigences logicielles devraient être des objets. L'exploration confirme l'hypothèse et aboutit à un ensemble de méthodes basées sur des outils pour spécifier, valider, vérifier et réutiliser les exigences orientées objets. La contribution technique réutilisable la plus importante de cette thèse est une bibliothèque Eiffel prête à l'emploi de patrons de classes, qui capturent les modèles d'exigences logicielles récurrents. Les exigences orientées objets, concrètes et sans rupture, héritent de ces patrons et deviennent des clients du logiciel spécifié. La construction de logiciels orientés objets devient la méthode de spécification, de validation et de réutilisation des exigences; la conception par contrat devient la méthode de vérification de l'exactitude des implémentations par rapport aux exigences.

Cette thèse s'appuie sur plusieurs expériences et montre que la nouvelle approche proposée favorise la vérifiabilité, la réutilisabilité et la compréhensibilité des exigences tout en maintenant l'expressivité à un niveau acceptable. Les expérimentations mettent en oeuvre plusieurs exemples, dont certains sont des standards de l'état de l'art de l'ingénierie des exigences. Chaque expérimentation illustre un problème par un exemple, propose une solution générale et montre comment la solution règle le problème. Alors que l'expérimentation s'appuie sur Eiffel et son support d'outils avancés, tels que la preuve et les tests automatisés, chaque idée présentée dans l'approche SOOR s'adapte conceptuellement à tout langage de programmation orienté objet typé statiquement, possédant un mécanisme de généricité et un support élémentaire pour les contrats.

Abstract

The constantly changing customers' and users' needs require fast response from software teams. This creates strong demand for seamlessness of the software processes. Continuous integration, delivery and deployment, also known as DevOps, made a huge progress in making software processes responsive to change. This progress had little effect on software requirements, however. Specifying requirements still relies on the natural language, which has an enormous expressive power, but inhibits requirements' traceability, verifiability, reusability and understandability. Promoting the problematic qualities without inhibiting the expressiveness too much introduces a challenge.

Bertrand Meyer, in his multirequirements method, accepts the challenge and proposes to express individual requirements on three layers: declarative subset of an object-oriented programming language, natural language and a graphical notation. This approach has motivated and inspired the work on the present thesis. While multirequirements focus on traceability and understandability, the Seamless Object-Oriented Requirements approach presented in the dissertation takes care of verifiability, reusability and understandability.

The dissertation explores the Martin Glinz' hypothesis that software requirements should be objects to support seamlessness. The exploration confirms the hypothesis and results in a collection of tool-supported methods for specifying, validating, verifying and reusing object-oriented requirements. The most significant reusable technical contribution of the dissertation is a ready-to-use Eiffel library of template classes that capture recurring software requirement patterns. Concrete seamless object-oriented requirements inherit from these templates and become clients of the specified software. Object-oriented software construction becomes the method for requirements specification, validation and reuse; Design by Contract becomes the method for verifying correctness of implementations against the requirements.

The dissertation reflects on several experiments and shows that the new approach promotes requirements' verifiability, reusability and understandability while keeping expressiveness at an acceptable level. The experiments rely on several examples, some of which are used as benchmarks in the requirements literature. Each experiment illustrates a problem through an example, proposes a general solution, and shows how the solution fixes the problem. While the experimentation relies on Eiffel and its advanced tool support, such as automated proving and testing, each idea underpinning the approach scales conceptually to any statically typed object-oriented programming language with genericity and elementary support for contracts.

Contents

List of Figures

List of Tables

Introduction

Seamless development

It affects project organization, and the very nature of the software profession; in line with modern trends in other industries, it tends to remove barriers between narrow specialties – analysts who only deal in ethereal concepts, designers who only worry about structure, implementers who only write code – and to favor the emergence of a single category of generalists: developers in a broad sense of the term, people who are able to accompany part of a project from beginning to end.

Bertrand Meyer

Definition 0.0.1 ***Seamlessness*** *is the use of a continuous process throughout the software lifecycle [Mey97].*

Bertrand Meyer, in his "Object-Oriented Software Construction" (OOSC) book [Mey97], presented the idea of developers in a broad sense of the term – as people who are able to accompany part of a project from beginning to end. This idea, originating from the first edition of the OOSC book back in 1988, was prophetic: companies more and more value individual contributors who alone can take a software feature from the analysis through construction to maintenance. Software processes and tools should support such contributors, collectively called *developers*. As opposed to the skills of the people performing specific tasks, such as analysts, architects, programmers and testers, developers' skills crosscut these tasks. Software processes' continuity stands on the developers' shoulders, and the present dissertation has the objective of simplifying their lives at the conceptual level, as DevOps [Ebe+16] does at the level of tools automating mundane tasks, such as building and testing. People naturally want to solve creative tasks, everything else should be automated. DevOps tools not only automate tasks within individual software development lifecycle (SDLC) phases, but also trigger execution of a next phase when observing certain events in the previous phase.

While tools may help, several conceptual gaps remain, one of which is the notational gap. Individual SDLC phases have been historically relying on their own notations, which was sensible when people were given specific tasks and mastering one notation would be enough to handle one task. With individual contributors taking responsibility for entire features, the following problems emerge:

- The developers must learn and practice several notations.
- Dedicated traceability tools must be in place.
- The developers must have enough discipline to record traceability links.

The seamless approach [WN94], [Mey97] attempts to remove the notational gap by applying the implementation programming language throughout the SDLC. Success of this effort would have the following implications:

- Knowing the implementation programming language would be enough to practice the entire SDLC.
- The native code traceability features of integrated development environments (IDEs) would also serve for tracing requirements.
- The developers would not need to record traceability links between different kinds of artifacts – requirements, code, tests, etc.

The idea to use programming languages as requirements notations is gaining support. Many groups of stakeholders prefer descriptions of operational activity paths over declarative requirements specifications [SFO03]. A demand exists for educating developers capable of both abstracting in a problem space and automating the transition to a solution space [WHR14]. The decision to express requirements in programming languages may also be the only way to bring the developers closer to the requirements they implement: industry practitioners are generally not keen to switching their tools [Dal+18].

The real situation does not meet these needs, however. The state-of-the-practice [PQF17] and the literature [IPP18] studies show no evidence that existing requirements approaches consider connecting the problem and the solution spaces. The studied approaches focus on reusing natural language, use cases, domain models and several other artifacts disjoint from the solution space.

The thesis

The object-oriented paradigm builds on the idea of supporting developers at the level of language and environment [Mey97]. This aspiration does not meet the reality, however. Developers specify requirements in natural language or modeling notations, implement them in programming languages, verify correctness of the solutions using tests and sometimes "reuse" the requirements through copying and pasting. Modern IDEs pay a lot of attention to implementation and testing, sometimes to modeling. Requirements are left out to specialized tools working with their own notations and semantics. The Seamless Object-Oriented Requirements approach attempts to make requirements full citizens of the IDEs.

Martin Glinz, in his “Should Requirements Be Objects?” position paper [Gli], discusses arguments in favor and against treating requirements as objects. The “in favor” section concludes with the following remark:

> “Furthermore, if we employ state-of-the-art object-oriented design and implementation techniques, an object-oriented requirements specification would allow a seamless application of object-oriented software engineering methods through the complete development cycle, from inception to deployment. We would get a smooth transition from requirements into architecture and design and could apply round-trip engineering methods and tools. So why longer hesitate? Just let requirements become objects.”

The “against” section then downplays that inspirational argument. It opens with several examples of requirements that are “clearly not objects”:

> “The promises of abstraction and comprehensibility sound good, but – treating requirements as objects is like making a problem fit a solution, instead of doing it vice-versa. What is a requirement? A requirement may be a goal, for example “The new CRM system shall reduce the number of customer complaints by at least 50%.” Is this an object? What does it encapsulate? Has it a state or behavior? Not really. So let’s try another kind of requirement. A requirement may be a function, for example “The system shall compute the maximum speed that the train can run with on the current track segment.” Is a function an object? Definitely not. So let’s again try another kind of requirement. A requirement can be a constraint, for example “In normal operating mode, the lift shall never move when the doors are not closed completely.” But again, a constraint is no object.”

The analysis of these examples, leading to the negative conclusion, looks superficial. Yes, all these requirements are objects – textual objects at a minimum. Natural language text constitutes one dimension of requirements. The developers will eventually write executable tests to verify correctness of candidate solutions against these requirements. These tests will form another dimension of the same requirements. Other dimensions, such as graphical, or audio representations may exist. Requirements frequently follow, as empirical evidence suggests, several patterns (*SRPs – software requirement patterns*) along some of these dimensions [DAC99], [KC02], [KC05]; these SRPs should be reusable. Here comes the main thesis of my dissertation:

> *Requirements, with their recurring structure and multidimensional nature, constitute natural input for the object-oriented analysis.*

The answer to the Martin Glinz’ “Should Requirements Be Objects?” question is a clear “yes”. Here are the sub-theses that refine the main one:

1. Requirements are objects instantiated from requirement classes.

2. Construction of requirement classes follows the object-oriented principles [Mey97].
3. The requirements' dimensions are implemented through the requirement classes' features.

The requirement classes map to recurring requirement patterns, some of which reccur especially often [DAC99], [KC05]. Such patterns should be reusable, and object orientation provides the reusability mechanisms that have already found their place in the developers' daily practices. The requirements' dimensions – textual, graphical, verifiable etc. – constitute their meaning [Mey13]. Different software engineering activities favor different dimensions; it is natural, therefore, for a single requirement to exist in different notations serving different purposes. It seems natural to represent requirements as classes with the features supporting the multidimensional analysis. Bertrand Meyer stated the initial principles behind object-oriented requirements in the "Thesis B" section of his "Multirequirements" article [Mey13]. The present dissertation develops these principles to cover more practical problems and situations. Part I discusses these problems and situations in detail.

Summary of contribution

The thesis presents Seamless Object-Oriented Requirements – a practial requirements methodology optimized for the purposes of seamless development. It reuses the existing features of the modern IDEs for specifying, validating, verifying, reusing and tracing requirements. The IDEs become the single working environment for developers who take full responsibility for complete software features. The methodology relies on the following key notions:

- *Seamless object-oriented requirement (SOOR).*
- *Seamless object-oriented requirement template (SOORT).*

Section 3.3 precisely defines and interconnects these notions, and Section 3.4 presents activities in which these notions serve as the main artifacts. The rest of the dissertation uses the "SOOR" abbreviation to refer either to the approach, or to an individual requirement specified according to the approach; the actual meaning will be clear from the context.

The dissertation presents a unified seamless approach that features a wide range of technical capabilities for specifying, validating, implementing and verifying requirements. The following list summarizes these capabilities:

1. Handling realistic systems with hard to formalize requirements [NMR15].
2. Specifying arbitrary abstract data types (ADTs) [NM16]
3. Statically checking contracts' well-definedness, correctness and completeness [NM16].
4. Static proof-oriented detection of inconsistent contracts [Nau18].

5. Incrementally-iterative proof-oriented software process reusing the underlying IDE for handling requirements [NM17].

6. Specifying and verifying control software temporal properties and timing constraints [Nau+19].

7. Capturing software requirement patterns (SRPs) as object-oriented templates for faster specification, validation and verification of new requirements [Nau19a]. A ready-to-use library of templates capturing known SRPs [Nau19b] supports this capability.

How to read the dissertation

Part I describes the problem in more detail. Part II presents the solution. Chapters 3 - 5 present the key ideas. Chapters 7 - 12 provide the technical details behind these ideas and conduct several experiments to showcase these ideas in practice. Chapter 6 provides the connection between key ideas and the details behind them. Part III reflects on the results, drawing conclusions and paving the road towards future work.

I recommend the following ways of reading the present manuscript:

1. Read it completely, skipping chapters 7 - 12, to overview the most important ideas and develop intuition behind them. This way of reading will require staying focused: the material is dense and contains only the essentials of the thesis.

2. Sequentially read chapters 7 - 12. This will increase the amount of reading but lower its density: the chapters incrementally develop the essential ideas, building each on top of the previous ones in a bottom-up fashion.

3. Read the dissertation completely. Chapter 6 connects chapters 7 - 12 with the essentials overview. This way of reading will give the full picture and require the biggest amount of time.

Part I

The Problem

Chapter 1

State of the Art

The present chapter discusses existing approaches leading to seamlessness (Definition 0.0.1) in some sense. These approaches may not explicitly focus on requirements or seamlessness, rather focusing on some other aspects, such as testing; seamlessness may come as a side effect. Section 1.1 characterizes approaches that are clearly relevant to the discussion, while Section 1.2 characterizes clearly irrelevant approaches.

1.1 Inclusion criteria

We only discuss approaches that lead to seamlessness at the software development life cycle (SDLC) level, in all directions: if a change happens in one SDLC phase, its consequences are observable in the other phases. Such approaches lead to the possibility of using the same set of notations and tools throughout the entire SDLC. Two notations are clearly unavoidable: the implementation programming language and the natural language.

1.2 Exclusion criteria

Some approaches, such as seamless model-based requirements engineering [Teu17], develop seamlessness within the analysis phase alone, with little concern for bridging the gap between requirements and other SDLC phases. The present dissertation has a clear objective: *simplifying lives of individual generalists – software developers*. Multiplying the notations disjoint from the implementation programming language and focusing on individual SDLC phases do not contribute to this objective.

Model-verify-generate approach assume modeling the system formally, verifying correctness of the model and then generating source code from the model. The well-known Event-B [Abr10] and LTSA [MK06] methods fall into this approach.

The present dissertation excludes the model-verify-generate approach from the discussion for the following reasons:

- Entering the solution space too early. The model-verify-generate approaches require a model of the future system already at the requirements specification stage. Requirements are not self-contained in these approaches: they become assertions (invariants, guards, trigger conditions, etc.) in the context of the chosen model. While design decisions must ensure satisfaction of the requirements, with the model-based approaches formulation of requirements themselves depends on pre-taken design decisions.

- Seamlessness in one direction. All changes start with changing the model, from which the source code is then re-generated. There is no way to modify the generated code and see if the modification violates the model. This is a critical problem: in practice it is always necessary to optimize the source code to meet non-functional requirements, such as performance and security, and the model-verify-generate approach does not provide mechanisms for expressing such requirements. While some of these approaches perform the model-to-code translation automatically, the need to modify the code will raise the demand for an additional effort of keeping the model consistent with the source code.

- Difficulty to master. The model-verify-generate approaches rely on mathematical
  formalisms that require specialized education. Forcing an existing, sometimes jelled, development team to learn these formalisms may ruin the project. These may not be a problem for companies developing mission-critical software, but we cater to generalists.

- Capturing the requirements as assertions in the modeling formalism. This may be realistic if both the customer and the contractor understand the modeling notation well enough to agree on the resulting document. Early requirements take the natural language form, and the model-verify-generate approach leaves the problem of connecting these early requirements with models open.

The model-verify-generate approach generally targets mission- and life-critical systems. This focus allows its practitioners to rely on additional strong assumptions about the process' high maturity level, the input requirements' high quality, the developers' awareness of formal methods and the project's generous schedule and budget. These assumptions rarely hold for the mass market software development.

1.3 Design by Contract

The first attempt to achieve full seamlessness and bring requirements to the developers' fingertips belongs to Design by Contract (DbC) [Mey92]. The method equips classes and their features with two-state assertions visible to their clients. DbC benefits seamlessness at the following levels:

Specification: contracts, when written during the analysis phase, prescribe the desired software behavior.

Construction: developers may rely on the IDE's intelligent facilities displaying the components' contracts; this greatly simplifies choosing the most appropriate components.

Verification: DbC enables both static and dynamic verification. Running an application equipped with contracts makes the runtime environment check these contracts; a contract violation forces the developer to debug both the contract and the code implementing it. Program proving, on the other hand, makes it possible to statically verify the absence of runtime violations before the first run of the program [Tsc+15].

Documentation: together with natural language comments, contracts may serve as comprehensive documentation for ready-to-use components.

With all its benefits, DbC in its pure form lacks specifications' incrementality. An individual requirement may crosscut more than two states and several concepts from the problem space. In this case, the contract assertions reflecting the requirement will be spread across several classes and features, which may inhibit the process' continuity. Individual requirements often take the form of standalone prescriptive statements [Lam09], and establishing traceability links between a single statement and several contract assertions will require specialized tools. Requirements that promote the process' continuity, or *seamlessness-oriented requirements*, should be standalone entities to eliminate the issue.

1.4 Multirequirements

The multirequirements method [Mey13] makes specifications incremental. The following principles define the method:

1. Develop individual requirements incrementally on several layers, including the following three: formal, graphical, natural language.

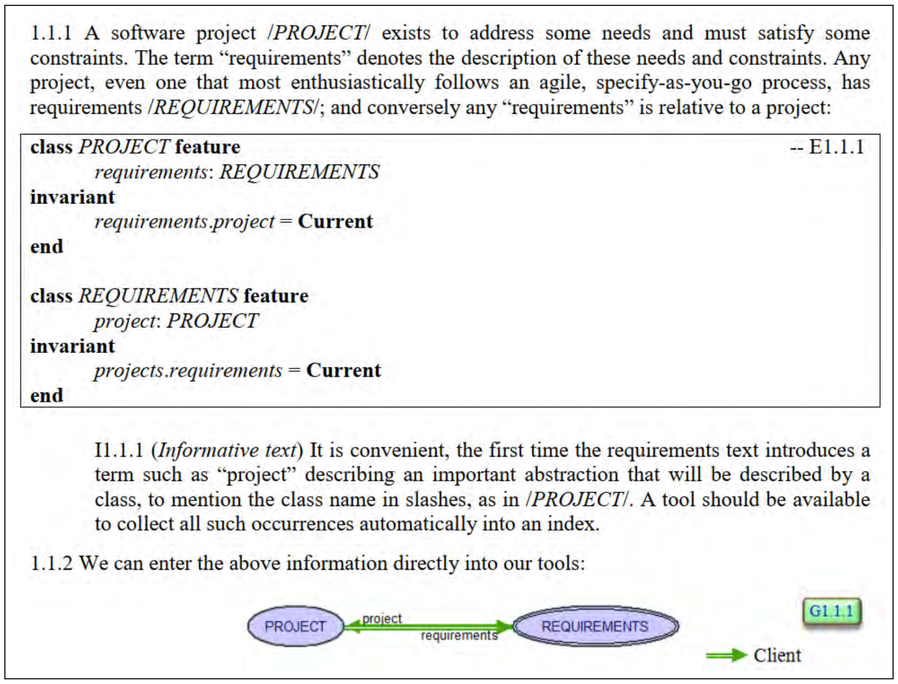
2. Use these layers both in a complementary way (when one of them is more appropriate to the description of a system property) and redundantly (for example to combine the precision of formal descriptions with the convincing power of graphical descriptions).

3. Model systems through object-oriented techniques: classes as the basic unit of decomposition, inheritance to capture abstraction variants, contracts to capture semantics.

4. Use an object-oriented language (e.g. Eiffel) to write the formal layer according to the principles of 3).

5. Use the contract sublanguage of the programming language as the notation for the formal layer.

6. As the goal is to describe models, not implementations, ignore the imperative parts of the programming language (such as assignment).



1.1.1 A software project /*PROJECT*/ exists to address some needs and must satisfy some constraints. The term "requirements" denotes the description of these needs and constraints. Any project, even one that most enthusiastically follows an agile, specify-as-you-go process, has requirements /*REQUIREMENTS*/; and conversely any "requirements" is relative to a project:

```
class PROJECT feature                                    -- E1.1.1
        requirements: REQUIREMENTS
invariant
        requirements.project = Current
end

class REQUIREMENTS feature
        project: PROJECT
invariant
        projects.requirements = Current
end
```

I1.1.1 (*Informative text*) It is convenient, the first time the requirements text introduces a term such as "project" describing an important abstraction that will be described by a class, to mention the class name in slashes, as in /*PROJECT*/. A tool should be available to collect all such occurrences automatically into an index.

1.1.2 We can enter the above information directly into our tools:

Figure 1.1: Multirequirement describing relationships between requirements and projects (taken from the original work [Mey13]). The three representation layers present the same meaning in different notations: natural language, Eiffel and BON. The natural language representation contains traceability links framed with the '/' symbol.

7. Use an appropriate graphical notation (BON [WN94]) for the graphical layer.

8. Weave the layers to produce requirements descriptions, including a comprehensive requirements document if requested, but also any other appropriate views.

9. Enforce and assess traceability between the layers and all products of the requirements process, and between requirements and other product artifacts, both down and up.

10. Rely on appropriate tools to support the process, including incremental development.

These principles expressly pursue seamlessness at the level of requirements to object-oriented software [Mey97] designed around the DbC principles.

Multirequirements interweave natural language prose with pieces of contracted code and BON [WN94] diagrams (Figure 1.1). The prose encloses names of important concepts in slash symbols to enable traceability across the three layers.

1.5 Parameterized unit tests

Parameterized unit tests (PUTs) may lead to seamlessness in the world of programming languages without native support for contracts. Their invention was motivated by the poor reuse of closed unit tests: several unit tests may check software correctness against the same abstract data type (ADT) axiom on different test inputs. In this case, these unit tests would duplicate the axiom's structure. Tillmann and Schulte [TS05] proposed to replace closed unit tests with parameterized methods, where the parameters would serve as universally quantified variables of the respective ADT axioms. For example, instead of writing closed unit test (in C#):

```
[TestMethod]
void TestAdd() {
  ArrayList a = new ArrayList(0);
  object o = new object();
  a.Add(o);
  Assert.IsTrue(a[0] == o);
}
```

they proposed to define a parameterized test axiom:

```
[TestAxiom]
void TestAdd(ArrayList a, object o) {
  Assume.IsTrue(a!=null);
  int i = a.Count;
  a.Add(o);
  Assert.IsTrue(a[i] == o);
}
```

and then rewrite the original unit test as:

```
[TestMethod]
void TestAddWithOverflow() {
  TestAdd(new ArrayList(0), new object());
}
```

Adding another test checking the same axiom becomes straightforward:

```
[TestMethod]
void TestAddWithNoOverflow() {
  TestAdd(new ArrayList(1), new object());
}
```

PUTs promote separation of concerns by splitting ADT axioms and test inputs, where the inputs may be automatically generated from the axioms [TH08]. The approach promotes seamlessness, though the original purpose was to increase the level of reuse: requirements, in the form of ADT axioms, become expressed in the implementation programming language.

PUTs' contributions are (taken from the original work [TS05]):

- They allow unit tests to play a greater role as specifications of program behavior. In fact, PUTs are axiomatic specifications.
- They enable automatic case analysis, which avoids writing implementation-specific unit tests.
- Their generated test cases often result in complete path coverage of the implementation, which amounts to a formal proof of the PUTs' assertions.

PUTs found their place in open source projects [Lam+15] and in a software process that replaces test-driven development (TDD) [Fra+03] with parameterized TDD (PTDD) [DTS10].

1.6 Theory-based testing

Theory-based testing [SBE08] leads to seamlessness in the same way as the PUT-based testing does. *Theories* are partial specifications of program behavior [SBE08]. Their syntax shares a lot with the PUTs' syntax: both represent unit tests parameterized over universally quantified paramenters:

```
@Theory defnOfSquareRoot(double n) {
  // Assumptions
  assumeTrue(n >= 0);

  double result = sqrRoot(n) * sqrRoot(n);

  // Assertions
  assertEquals(n, result, /* precision: */ 0.01);
  assertTrue(result >= 0);
}
```

JUnit, a unit testing framework for Java programs, contains an implementation of theories in version 4.4 and later.

Theory-based testing and PUT-based testing differ in how they handle the respective artifacts – theories and PUTs. Where Tillman and Schulte generate provably minimal test suites based on complete specifications, Saff et al. [SBE08] accept heuristics that generate data points designed to exercise as many code paths as possible in a short time. Theory-based testing relaxes the requirement for the specifications to be complete. From the seamlessness viewpoint, the two approaches are equal. Both encode ADT axioms in the implementation programming language and have interchangeable formats. Choosing one of them amounts to comparing the respective tools for generating test inputs.

1.7 Abstract testing

Abstract testing [Mer+15] expressly attempts to bridge the gap between requirements and test cases, while PUT- and theory-based testing were targeting reuse of unit tests

and high coverage of code with tests. Syntactically, abstract tests rely on the same idea that PUTs and theories build upon: specifying behaviors through contracted routines, possibly parameterized. The approach treats these routines, however, not as abstractions of closed unit tests, but as formalizations of requirements. In this regard, Merz et al. [Mer+15] detach the approach from testing and discuss it in the broader context of verifiable requirements. Abstract testing focuses on control software, for which it is necessary to non-deterministically initialize environment variables. The approach achieves this initialization through auxiliary routine
`nondeterministically_initialize_environment`. Implementing this routine becomes a task of the test engineer.

The following example presents the common structure of abstract tests:

```
abstract_test() {
  nondeterministically_initialize_environment();
  assume(precondition(x1));
  .. .
  assume(precondition(xn));
  y = f(x1,.. .,xn);
  assert(postcondition(x1,.. .,xn,y));
}
```

The first instruction non-deterministically initializes the environment; the **`assume`** statements make assumptions about the environment; the **`assert`** statement requires the postcondition to hold under the stated assumptions. Abstract testing contributes to seamlessness by explicitly proposing PUT-like constructs as a requirements notation.

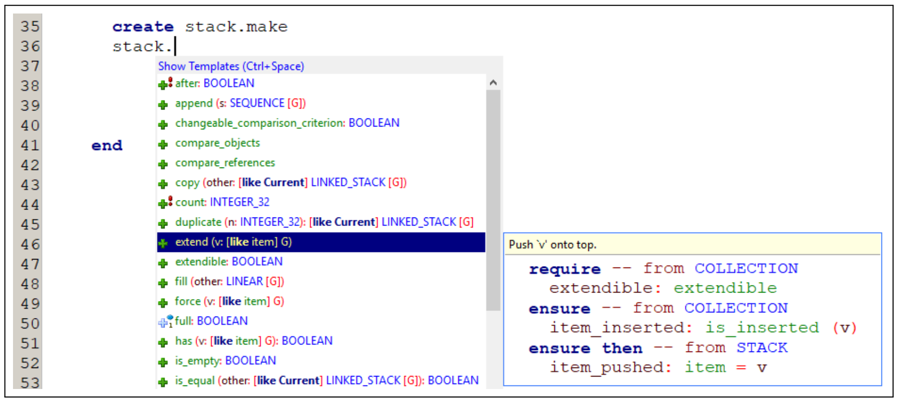
1.8 Reflections

The authors of the PUT-like approaches (PUTs, theory-based testing and abstract testing) sometimes perceive DbC as a competing approach [Lam+15], which prevents the two views from benefitting each other.

Contracts are irreplaceable in how they document software components. Figure 1.2 depicts EiffelStudio during the programming process. More concretely, it depicts a situation in which the programmer has just entered a dot symbol after a variable and is looking for a feature to call. EiffelStudio offers the list of features callable on the variable. Going through the list causes the selected feature's documentation to appear in the rightmost pop-up window. It contains the natural language description of the feature along with its semantics in the form pre- and postconditions. The ability to see the callable features' meanings may significantly speed-up the programming process.

PUTs, on the other hand, offer incrementality: two PUTs may specify different components but reside in the same class, which will simplify searching and modifying them. DbC, on the contrary, assumes that the specified components contain their own specifications in the form of contracts. This approach, also known as "Single-Product Principle" [Mey97], ensures the great documenting capability of contracts. As a side effect, it results in specifications spread across the specified components, which complicates their management.

The present dissertation shows that contracts and the PUT-like specification approaches are, in fact, fundamentally connected and may benefit each other when prac-



```
        create stack.make
        stack.

     end
```

```
Show Templates (Ctrl+Space)
after: BOOLEAN
append (s: SEQUENCE [G])
changeable_comparison_criterion: BOOLEAN
compare_objects
compare_references
copy (other: [like Current] LINKED_STACK [G])
count: INTEGER_32
duplicate (n: INTEGER_32): [like Current] LINKED_STACK [G]
extend (v: [like item] G)
extendible: BOOLEAN
fill (other: LINEAR [G])
force (v: [like item] G)
full: BOOLEAN
has (v: [like item] G): BOOLEAN
is_empty: BOOLEAN
is_equal (other: [like Current] LINKED_STACK [G]): BOOLEAN
```

```
Push 'v' onto top.
  require -- from COLLECTION
    extendible: extendible
  ensure -- from COLLECTION
    item_inserted: is_inserted (v)
  ensure then -- from STACK
    item_pushed: item = v
```

Figure 1.2: EiffelStudio displaying hints, including contracts and natural language comments.

ticed together, thanks to program proving. To illustrate the concepts, the dissertation uses AutoProof [Tsc+15] – the prover of Eiffel programs.

Chapter 2

Important Qualities of Requirements

The present chapter describes important qualities of a practical requirements approach and briefly evaluates the state-of-the-art approaches against the stated qualities; we evaluate the SOOR approach in Chapter 13 and Chapter 14.

We map the stated qualities to the recommendations of the ISO/IEC/IEEE 29148 "Requirements engineering" standard [ISO11], sections "5.2.5 Characteristics of individual requirements" and "5.2.6 Characteristics of a set of requirements". The document recommends, among other characteristics, to keep requirements *singular* – a requirement statement should include only one requirement with no use of conjunctions. The standard does not explain, however, why this characteristic is important. Neither does it define the very notion of conjunction, widely known as a Boolean operator, in the context of requirements. If defined, conjunction would most probably apply to a pair of requirements expressed in the same notation. The dissertation focuses exactly on what this hypothetical notation should look like, and defining operations on top of it seems to be a concern for the future work. Given these arguments, we decided to exclude the singularity characteristic from the discussion.

Most of the standardized characteristics support what we discuss as *understandability* (Section 2.4). *Expressiveness* (Section 2.1) characterizes requirements approaches rather than requirements themselves, which is why the standard does not discuss it. We find this quality important, however, because we are exploring applicability of programming languages as requirements notations; while natural languages have enormous expressive power, programming languages' expressiveness needs to be explored.

2.1 Expressiveness

Definition 2.1.1 ***Expressiveness*** *is the suitability of an approach for capturing requirements of different forms.*

Software takes the following forms:

- *Control software* works in an infinite loop and continuously reacts to events in the environment.
- *Software components* process input data in finite time and produce some output data.

Software components serve as building blocks for control software and other software components. They take the form of command-line utilities, program modules and any other form that meets the definition of a software component. Requirements to software components take the form of *abstract data type* (ADT) specifications [GHM76]. Arrays, stacks, strings are a few examples of software components; they come inside standard libraries of programming languages.

Specification of control software, on the other hand, relies on temporal properties [DAC99] and timing constraints [KC05] – requirements that the theory of ADTs does not cover.

A practical approach thus should be suitable for expressing at least:

- ADT axioms,
- Temporal properties,
- Timing constraints.

The state-of-the-art approaches fail to meet this expressiveness standard. Multirequirements fundamentally rely on contracts, and contracts cannot capture multicommand ADT axioms; they also cannot capture temporal properties nor timing constraints. PUTs, theories and abstract tests can capture multicommand requirements, but not temporal properties and timing constraints.

The SOOR approach combines the expressive power of contracts and PUT-like specifications for capturing all the three kinds of requirements.

2.2 Verifiability

Definition 2.2.1 *A requirement is* ***verifiable*** *if it has the means to prove that the system satisfies the specified requirement. [ISO11]*

The standard [ISO11] does not specify how these "means to prove" should technically look like. In this section we come up with several desired properties that such means should have.

- Verifiability should be *modular*. The state-of-the-art approaches have problems with verifiability. In multirequirements, the requirements in the form of contracts become an integral part of the solution, which makes it conceptually impossible to fully separate the problem from the solution. Contracts represent a powerful verification mechanism suitable both for testing [Mey+09] and program proving [Tsc+15]. Their nature, however, assumes instrumentation of the verified code, which may not be possible for already implemented components. Even if a component is available for modification, the instrumentation may alter it. A modular specification and verification mechanism should be in place that would not require modifying the verified components.

- Verifiability should be *twofold* – both static and dynamic. The PUT-like approaches are free of the modularity problem: they do not require instrumenting the verified solution. They are perceived, however, as purely testing approaches, which is not the case for Design by Contract – it is equipped with tools for both static [Tsc+15] and dynamic [Mey+09] verification. Seamlessness-oriented requirements should have this duality and at the same time support verification modularity.
- Verifiability should be *reusable*, in the sense of reusing requirements' verifiable semantics. Requirements for finite-state verification mostly follow several software requirement patterns (SRPs) [DAC99], [KC05], yet the secondary studies of requirements reuse approaches do not evaluate the approaches' suitability for producing not just reusable but also verifiable requirements. This concern applies to both state-of-the-practice [PQF17] and state-of-the-literature [IPP18] secondary studies.

2.3 Reusability

Definition 2.3.1 ***Reusability*** *is the suitability of recurring requirements' structures to be reused across projects for simplifying specification, comprehension and verification of the new requirements.*

The ISO/IEC/IEEE 29148 "Requirements engineering" standard [ISO11] mentions requirements reusability only in the context of product lines and sends the reader to the corresponding standard, ISO/IEC 26551 "Tools and methods of requirements engineering and management for product lines". We think, however, that requirements reuse should not be limited to product lines. Empirial studies [DAC99], [KC05] uncovered recurring patterns in requirements not intended for development of product lines. In our opinion, requirements reuse is at least as important as software reuse. It might help not only save resources in the analysis phase, but also obtain requirements specifications of better quality both in content and syntax. It might also decrease the risk of writing low quality requirements and lead to the reuse of design, code, and test artifacts.

Reusability has become a success story in the reuse of code [Zai+15] and tests [TS05], but not requirements. Despite the existence of many requirements reuse approaches [IPP18] the actual level of requirements reuse is low [PQF17]. Textual copy and subsequent modification of requirements from previous projects are still the most commonly used requirements reuse techniques [PQF17], which has already been long recognized as deficient in the world of software reuse.

Control software requirements follow several SRPs. Dwyer et al. analyzed 555 specifications for finite-state verification from different domains and successfully matched 511 of them to 23 known SRPs [DAC99]. The SRPs were encoded in modeling notations with no guidance on how to reuse them across projects for verifying software solutions and put to an online catalogue. In 2005, Konrad and Cheng [KC05] emphasized the importance of real-time requirements and created a catalogue of real-time verification-oriented SRPs, inspired by the catalogue of Dwyer et al. The new SRPs have the same qualitative semantics as the original ones but add the real-time quantitative semantics in terms of three commonly used real-time temporal logics. How to make these SRPs seamlessly reusable across projects?

The most critical factors inhibiting the industrial adoption of requirements reuse through SRP catalogues are [PQF17]:

- The lack of a well-defined reuse method,
- The lack of quality and incompleteness of requirements to reuse,
- The lack of convenient tools and access facilities with suitable requirements classification.

Scientific literature studying requirements reuse approaches pays little attention to these factors when measuring the studied approaches [IPP18]. The degree of reuse is the most frequently measured variable, but it is measured under the assumption that the evaluated approach is fully practiced. This assumption does not meet the reality: most of the practitioners who declare to practice requirements reuse approaches, apply them very selectively [PQF17]. Secondary studies, which study other studies, equally ignore the factors that matter to practitioners [IPP18].

Neither multirequirements, nor the PUT-like specification mechanisms consider the reusability concern extensively. PUTs achieve some reuse at the level of tests: they capture ADT axioms often repeated in closed test methods, and testing reduces to replacing the PUTs' parameters with actual values. PUTs do not abstract away the typing information, so they are not reusable across differently typed components. Contracts, on which multirequirements rely, offer reusability across test methods by design [Mey+09]: preconditions check relevance of the test input, and postconditions check correctness of the tested software. From the typing perspective, contracts offer reusability within the same inheritance tree: descendants inherit contracts from their ancestors. The semantics of such inheritance depends on whether it is a precondition, a postcondition, or a class invariant. DbC does not provide, however, explicit mechanism to reuse recurring contracts across components not connected through the inheritance relation.

2.4 Understandability

Definition 2.4.1 ***Understandable*** *requirements have the same meaning to all stakeholders and can immediately serve as input to their activities.*

Seamlessness would allow individual stakeholders to quickly see how a change on someone else's side affects their work. Requirements should serve as the main communication vehicle in responding to change. This places high demands on their understandability. Early requirements typically come in the natural language form, suffering from many understandability problems raised by Bertrand Meyer back in

1985 [Mey85]. These problems happened to map very well to the standardized recommended characteristics of requirements and their compositions [ISO11]:

- *Noise* – the presence in the text of an element that does not carry information relevant to any feature of the problem [Mey85]. Variants: redundancy; remorse. Removing noise results in *necessary* [ISO11] requirements.
- *Silence* – the existence of a feature of the problem that is not covered by any element of the text [Mey85]. Removing silence results in *complete* [ISO11] specifications.
- *Overspecification* – the presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution [Mey85]. Removing overspecification results in *bounded* [ISO11] specifications consisting of *implementation free* [ISO11] requirements.
- *Contradiction* – the presence in the text of two or more elements that define a feature of the system in an incompatible way [Mey85]. Removing contradiction results in *consistent* [ISO11] specifications.
- *Ambiguity* – the presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways [Mey85]. Removing ambiguity results in unambiguous [ISO11] specifications.
- *Forward reference* – the presence in the text of an element that uses features of the problem not defined until later in the text [Mey85]. Forward referencing is a special case of non-traceable requirements. Removing forward referencing results in upwards traceable requirements. Adding downwards traceability results in fully *traceable* [ISO11] requirements.
- *Wishful thinking* – the presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature [Mey85]. Removing wishful thinking results in *affordable* [ISO11] specifications consisting of *feasible* [ISO11] requirements.

The characteristics recommended by the standard promote requirements' understandability. The state-of-the-art approaches lack evaluation against these characteristics.

The PUT-like mechanisms are:

- Implementation free: they have the form of external test methods that call exported implementations' features.
- Unambiguous: they have unique meaning as programming language constructs.
- Downwards traceable: the calls to the specified features become the traceability vehicle.

The PUT-like approaches lack explicitly defined mechanisms that would guarantee the remaining characteristics.

Multirequirements are specified at several representation layers; one of the layers consists of piecemeal contracts. This makes multirequirements:

- Unambiguous: contracts have a precise mathematical semantics.
- Downwards traceable: the multirequirements' piecemeal contracts are part of the implementation.
- Upwards traceable: multirequirements collocate contracts' pieces with representations of the same requirements at the other layers.

Multirequirements promote completeness, consistency, unambiguity and feasibility: the several representations of the same requirement may force the reader to think deeper about its meaning. They inhibit implementation freedom, however: as piecemeal contracts, they will become part of the future implementation.

Meyer proposed the process of passing requirements through a formal notation to produce their more understandable natural language versions – "The Formal Picnic Approach" [Mey18]. The state-of-the-art approaches do not include a similar-purpose mechanism.

Our task is to reuse the existing mechanisms of PUTs, multirequirements and formal picnics to promote the desired understandability characteristics and remove the mechanisms that inhibit them.

Part II

The Unified Solution

Chapter 3

Essentials

The solution to the problem of finding a seamlessness-oriented requirements approach for object-oriented software construction is the object-oriented software construction itself [Mey97]. Requirements should be classes, in the object-oriented sense. Recurring requirement patterns should become abstract template classes with deferred features that, when implemented, will turn into concrete requirements. Technically, project-specific requirements inherit from these template classes and become clients of the specified software components. Methodologically, object-oriented software construction [Mey97] becomes the requirements specification method, and DbC [Mey92] becomes the requirements verification method. The dissertation presents a ready-to-use library of Eiffel classes that capture already identified SRPs for control software and software components. The library provides a starting point for practicing the approach that we called *Seamless Object-Oriented Requirements (SOOR)*.

The SOOR process takes natural language requirements on input and produces on output object-oriented requirements that are reusable and verifiable. Every object-oriented requirement also contains a function that automatically generates paraphrased natural language version of the input natural language requirement. The main purpose of having the paraphrased natural language version is to validate the original input requirement: the developer looking at the two versions will unconsciously start comparing them and possibly correcting the original requirement. This process is currently known as "The Formal Picnic Approach" [Mey18] and was justified more than 30 years ago [Mey85]. Object-oriented requirements also contain preprogrammed contracted routines for verifying correctness of candidate solutions. They encode in the verifiable form either ADT axioms, if the task is to implement a software component, or temporal properties and timing constraints, if the task is to implement a control software.

The present chapter details the key artifacts of the process and the core activities consuming and producing these artifacts.

3.1 The choice of notation and technology

The main task of the thesis was to explore applicability of object-oriented programming languages at the analysis phase. We chose Eiffel as the representative language to illustrate our concepts. It has human-friendly syntax, natively supports contracts and builds around object-oriented concepts. An advanced technology stack accompanies Eiffel. Contracts-based program proving and testing with AutoProof [Tsc+15] and AutoTest [Mey+09], traceability to and from external sources with the Eiffel Information System (EIS) allowed us work at the cutting edge of the programming technology. AutoProof has been playing a key role in our studies. It is a program prover based on Hoare logic [Hoa69] extended with *semantic collaboration* [Pol+14] – reasoning framework that covers phenomena specific to object-oriented programming, such as aliasing, callbacks and information hiding. Polikarpova et al. demonstrated practical applicability of AutoProof by using it to fully verify EiffelBase2 – a specified library of containers [PTF18]. We have been using EiffelBase2 extensively as a valuable source of data for testing our ideas.

3.2 Specification drivers

Design by Contract [Mey92] was originally designed under the assumption that the contracts would be checked at run time. Practitioners were perceiving code solely as an executable artifact. AutoProof makes it possible to use program elements as statically verifiable statements that may never be executed. This possibility has been the main thinking vehicle driving the development of the thesis.

Specification drivers operationalize this possibility and a key hypothesis of the thesis: *Hoare logic is the best notation for capturing software requirements formally.* The dissertation describes several innovative concepts, among which the notion of specification driver is the most fundamental. Understanding this concept is essential for understanding the rest of the work: the other concepts build on top of specification drivers. Syntactically, a specification driver is an object-oriented Hoare triple, or a self-contained contracted routine. The following specification driver formally captures one of the axioms of stack:

```
push_then_pop (s_1, s_2: STACK [G]; x: G)
-- pop (push (s, x)) = s
-- Popping a stack after pushing an element on it results in the original stack,
-- assuming that these operations modify only the stack itself.
  require
    s_1 ~ s_2
  modify
    s_1
  do
    s_1.push (x)
    s_1.pop
  ensure
    s_1 ~ s_2
  end
```

The natural language comment captures the axiom's mathematical representation and informal description. The `push_then_pop` routine depends only on its formal parameter and is self-contained in that sense. The routine may be submitted for static verification to AutoProof, or run as a parameterized unit test for dynamic verification. The **`modify`** clause captures the frame condition, critical for static verification. The **`require`** and **`ensure`** clauses capture the routine's pre- and postcondition, respectively.

Definition 3.2.1 *A **specification driver** is a self-contained contracted routine that captures some behavioral property of its formal parameters through the contract.*

Chapter 8 gives more intuition behind specification drivers and how to apply them in the presence of a program prover. The subsequent chapters develop this idea further and find for it more complex applications – way more complex than specification and verification of stack. The specification drivers' syntax inherits a lot from the PUT-like approaches, which focus on the testing-based verification of ADTs and oppose themselves to contracts. Specification drivers:

- Capture temporal properties and timing constraints in addition to ADT axioms.
- Capture contracts' well-definedness and inconsistency axioms for checking with AutoProof.
- Serve as PUTs in testing-based verification.
- Capture requirement's formal semantics in a form reusable across projects.

The remaining chapters expand, detail and illustrate these benefits of specification drivers. The dissertation concludes with the generalized object-oriented treatment of requirements with specification drivers serving as the verification mechanism. They became the main thinking vehicle taking us to the general notions of seamless object-oriented requirement (SOOR) and SOOR template (SOORT).

3.3 Artifacts

Definition 3.3.1 ***Natural-language requirements (NLR)** are requirements relying on the natural language and serving as the initial input to the software process. Execution of the software process derives other artifacts from the initial input.*

NLRs may take the form of completely informal statements, user stories, use cases, etc. Their specific structure has no importance in the context of the present work.

Definition 3.3.2 ***Seamless Object-Oriented Requirement Templates (SOORT)** are generic and deferred classes capturing SRPs. The formal generic parameters and deferred features represent blank sections of the templates to fill in.*

SOORTs represent the key mechanism for achieving reusability of object-oriented requirements.

Definition 3.3.3 ***Seamless Object-Oriented Requirements (SOOR)** are non-generic concrete classes capturing NLRs and inheriting from a SOORT.*

3.4 Activities

Several major activities characterize the SOOR approach. Chapter 5 provides technical details to help developing new SOORTs. Chapter 4 presents a ready-to-use library of SOORTs capturing already known SRPs. Chapter 6 introduces chapters detailing the remaining activities. These chapters not only detail the respective processes, but also give illustrative examples to develop better intuition behind the approach.

3.4.1 Developing a SOORT

Developing a SOORT requires the same skills as developing any other object-oriented class. It assumes identification of a pattern, hardcoding its immutable part and parameterizing its variable part through abstraction and genericity.

1. Identify the pattern's formal semantics.
2. Declare the SOORT class and name it to reflect the identified semantics.
3. Encode the identified semantics through specification drivers and put them inside the SOORT class.
4. Make the specification drivers work with generic, not actual types; make the generic types part of the enclosing SOORT's declaration.
5. Implement the function (we call it `out` in the rest of the discussion remaining text) producing the template's string representation; use avaialable reflection facilities to extract the generic types' names.

3.4.2 Specifying a SOOR

Converting an NLR to a SOOR assumes identifying patterns to which the NLR belongs, inheriting from the SOORTs capturing these patterns and implementing the SOORTs' variable parts. The resulting class must be fully defined.

1. Identify the NLR's formal semantics.
2. Find the SOORT encoding the identified semantics.
3. Create a concrete class inheriting from the found SOORT.
4. Replace the SOORT's formal generic parameters with actual generic parameters.
5. Implement the SOORT's deferred features.
6. Make sure that the newly implemented SOOR successfully compiles.

3.4.3 Having a formal picnic

Having a formal picnic for an NLR includes instantiating the SOOR corresponding to the NLR, getting the instance's natural language representation produced by the `out` function (Section 3.4.1), and comparing the result with the NLR. This comparison should trigger rethinking and refinement of the input NLR.

1. Construct an object from the SOOR class resulting from the "Specifying a SOOR" process.
2. Generate the object's string representation by calling the standard `out` function. The SOORT, from which the SOOR inherits, redefines the function according to the SOORT's semantics.
3. Compare the generated string with the input NLR.
4. If the generated string reflects the intended requirement's meaning more accurately than the input NLR, fix the NLR; go to step 3.
5. If the generated string does not reflect the intended requirement's meaning, inherit the SOOR from a different SOORT that would capture the NLR's meaning more accurately; go to step 1.

3.4.4 Verifying through testing

Testing correctness of a candidate implementation against a SOOR consists of running the specification drivers inside the SOOR, passing instances of the candidate implementation as formal arguments. The specification drivers serve as PUTs in this case.

1. Instantiate an object from the SOOR.
2. Call the object's specification drivers one by one, providing all the necessary actual arguments.
3. If a call fails with a precondition violation, fix the caller; go to step 2.
4. If a call fails with a loop variant violation, fix the implementation; go to step 2.
5. If a call fails with a postcondition violation, fix the implementation; go to step 2.
6. If a call fails with a loop invariant violation, identify the root cause of the failure.
7. Depending on the identified root cause, fix either the caller or the implementation; go to step 2.
8. If all the calls succeed, consider the tested implementation correct with respect to the SOOR.

The AutoTest technology [Mey+09] automates steps 1 through 3. The practitioner will only need to trace the AutoTest failures to their route causes and fix them.

3.4.5 Verifying through program proving

Proving correctness of a candidate implementation against a SOOR consists of running AutoProof on the SOOR. In this case, AutoProof will check correctness of the SOOR's specification drivers against the candidate solution's contracts. This may require writing additional annotations on the specification drivers that capture the SOOR's formal semantics.

1. Run AutoProof on the SOOR.
2. If AutoProof rejects the input, fix the implementation contract; go to step 1.
3. If AutoProof accepts the input, consider the implementation contract correct.
4. Implement the derived contract and check the implementation's correctness with AutoProof.

Chapter 4

Technical Contribution

The chapter presents two Eiffel libraries of SOORTs publicly available in [Nau19b] and as appendices of this dissertation:

- For specifying *control software* requirements (Appendix A). SOORTs of this kind capture recurring behaviors. They contain only one specification driver for verifying concrete SOORs.
- For specifying requirements to software components (Appendix B). SOORTs of this kind capture recurring concepts from the problem space. They contain several specification drivers capturing the ADT axioms describing the target concepts.

For a better intuition behind this separation, here are examples of typical requirements that might be handled using the two kinds of SOORTs:

- "Turning on air conditioning always results in the specified room temperature within one hour".
- "A store inventory behaves as a stack".

In the first case, the system has only one goal: achieving the necessary temperature in the room. The system achieves this goal by adjusting two parameters: the output air temperature and intensity of blowing out the air. Using a SOORT in this case assumes inheriting from it and connecting the system's main feature to the SOORT. The SOORT encodes the "Global Response" SRP, capturing its semantics through a single specification driver (Appendix A.20). Verification will consist in this case of calling or proving the specification driver.

A store inventory has the following key features: adding a new item and removing the topmost item. Applying the SOOR approach to the second requirement assumes inheriting from and implementing the "Stack ADT" SOORT (Appendix B.22), connecting the inventory's and the stack's features. The SOORT contains several specification drivers capturing the ADT axioms of stack. Verification will consist in this case of calling or proving all these specification drivers.

	Scope					
Pattern	Global	Before	After	Between	Until	Total
Absence	41	5	12	18	9	85
Universality	110	1	5	2	1	119
Existence	12	1	4	8	1	26
Bounded Existence	0	0	0	1	0	1
Response	241	1	3	0	0	245
Precedence	25	0	1	0	0	26
Response Chain	8	0	0	0	0	8
Precedence Chain	1	0	0	0	0	1
UNKNOWN						44
Total	438	8	25	29	11	555

Table 4.1: Distribution of the requirements analyzed by Dwyer et al. [DAC99] among known SRPs. Out of the 40 SRPs, 23 proved to be useful for covering some requirements. "Global Response" and "Global Universality" were the most frequently used SRPs, covering 351 out of the 555 requirements.

The two libraries offer a starting point for practicing the SOOR approach. The present chapter discusses their internals.

4.1 SOORTs for control software

Formal specifications of control software follow several known SRPs [DAC99], [KC05]. We have developed an object-oriented library of Eiffel classes capturing the SRPs' verification semantics and natural language representations. The classes are generic and abstract enough to remain reusable across systems.

In 1999, Dwyer et al. [DAC99] published an article summarizing their study of 555 verification-oriented requirements taken from different software domains (Table 4.1). The authors report that 511 out of the 555 requirements map into 23 known SRPs. The SRPs are available online in 5 notations: LTL, CTL, GIL, Inca, QRE.

The online library of SOORTs [Nau19b] captures in Eiffel the SRPs identified by Dwyer et al. The templates are configurable and thus can be used both in the purely qualitative form and with the real-time semantics added. The SOORTs include the real-time semantics anyway to limit the verification time through loop variants. The templates have the maximum integer [Var] as the default time boundary value. Because both the SOORTs and SOORs are classes, where SOORs implement the SOORTs through the inheritance relation, specifying real-time semantics in SOORs becomes an optional activity. The specifier may stay with the default time boundary provided by the template or redefine it through the standard object-oriented redefinition techniques. The object-oriented nature of SOORTs thus eliminates the need to maintain different catalogues for qualitative and real-time semantics: choosing one of the two becomes a matter of keeping or redefining the default time boundaries in the descendant SOORs.

4.2 SOORTs for software components

We found no studies like the one conducted by Dwyer et al. that would identify SRPs in ADT specifications of software components. After searching the available literature for such specifications, we concluded that few idiomatic ADTs and their variations often illustrate specification and verification approaches. Studying industrial applications of ADTs might be an interesting and challenging task as a possible continuation of the present analysis.

Table 4.2 maps the studied literature to the identified ADTs. Some ADTs are especially popular, and some sources study especially many ADTs. The most discussed ADTs are Stack and Queue plus their variations (5 occurrences each), Symbol table (2 occurrences) and Set plus its variation. The contributions by John V Guttag and his colleagues [GHM76], [Gut76], [GHM78], [GH78] comprise most of the ADTs' studies. Axel van Lamsweerde in his book [Lam09] discusses two examples of ADTs, Book directory and Library, that are not basic data structures but information systems. Do these studies and ADTs matter at all? Having empirical data from the industry would objectively reflect the actual situation, but we have no such data yet; we present a literature-based analysis instead. Besides looking at the number of ADTs discussed in individual papers, we took into account the popularity of these works in terms of citations on Google Scholar.

Table 4.3 maps the studied literature sources to the number of analyzed ADTs and to the number of citations on Google Scholar (as of February, 2019). 2 out of the 8 sources have more than 1000 citations; 5 sources have more than 500 citations, 4 sources out of the 5 analyze 2 or more ADTs. Given the high citation level, we conclude that the analyzed ADTs have practical value and are worth encoding them as reusable templates. The SOORTs encoding the ADTs reside in the "software_components" directory of our GitHub repository [Nau19b] and in Appendix B of this dissertation.

	[GHM78]	[GH78]	[GHM76]	[Lam09]	[KW91]	[Tho87]	[Gut76]	[LZ74]
Array	x						x	
Bag		x						
Binary tree			x					
Binary tree with inorder			x					
Book directory				x				
Bounded queue			x					
Bounded stack	x							
Environment					x			
File			x					
Graph			x					
Library				x				
Mapping	x							
Polynomial			x					
Queue		x	x			x	x	
Queue with append			x					
Set		x						
Set with emptiness check			x					
Stack			x				x	x
Stack with replace	x						x	
String			x					
Symbol table	x						x	

Table 4.2: Mapping the ADTs to the literature sources analyzing them. An ‘x’ symbol means that the source from the topmost row analyzes the ADT from the leftmost column.

Source	[GHM78]	[GH78]	[GHM76]	[Lam09]	[KW91]	[Tho87]	[Gut76]	[LZ74]
Number of ADTs	5	3	11	2	1	1	5	1
Google Scholar citation index	552	719	175	1175	42	5	657	1067

Table 4.3: Mapping the literature sources to the number of the studied ADTs and the number of citations on Google Scholar.

Chapter 5

Internals of Seamless Object-Oriented Requirement Templates

Construction of SOORTs follows the same algorithm (Section 3.4.1), which is why detailing one of them should suffice for understanding the overall idea. The SOORTs' structure follows the philosophy of capturing as much complexity as possible, to simplify specification of concrete SOORs. Specifying a SOOR from a SOORT consists of the following steps:

1. Inheriting from the SOORT.
2. Replacing the SOORT's formal generic parameters with the specified types.
3. Connecting the SOORT's deferred features with the specified types' concrete features.

From the extensibility viewpoint, the approaches to specifying SOORTs for control software and for software components differ as follows:

- A SOORT for control software represents an SRP capturing a finalized behavior.
- A SOORT for software components is an extensible collection of related behaviors.

The following sections illustrate this difference on specific examples.

5.1 Requirement templates for control software

Figure 5.1 depicts the Eiffel SOORT corresponding to the most frequently recurring SRP identified by Dwyer et al. – the "Global Response" SRP (Appendix A.20). Out

```
note
  description: "S responds to P globally"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y44wbnbs"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y2crlkjc"

deferred class
  RESPONSE_GLOBAL [G, expanded S →CONDITION [G], expanded P →CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    require
      p_holds: ({P}).default.holds (system)
    do
      from
        timer := time_boundary
      until
        ({S}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({S}).name + " responds to " + ({P}).name + " globally"
    end

end
```

Figure 5.1: The SOORT encoding the "Global Response" SRP from the catalogue of Dwyer et al (Appendix A.20). Specification driver `verify` encodes the formal semantics of the SRP. String function `requirement_specific_output` produces the natural language representation of the SRP parameterized with the formal generic parameters' names. Integer function `time_boundary`, inherited from `REQUIREMENT`, specifies the default time boundary for finite verification.

of the 555 requirements analyzed by Dwyer et al., 241 were instances of this SRP [DAC99]. It takes the following form in LTL:

$$\square(P \Rightarrow \diamond S) \tag{5.1}$$

where P is called "stimulus" and S is called "response"; "$\square$" and "$\diamond$" encode for the "always" and "eventually" temporal logic [Pnu77] operators. Line 2 in Figure 5.1 captures the string representation of the SRP, where `S` and `P` vary between requirements. Line 3 provides a named hyperlink to a OneNote page detailing the SRP in the initial 5 notations provided by Dwyer et al. [DAC99]. Line 4 provides a named hyperlink to the location of the class on GitHub. The EIS (Eiffel Information System) mechanism of EiffelStudio makes it possible to construct, maintain and follow named hyperlinks. Lines 6-7 declare the class capturing the SRP. The declaration depends on three formal generic parameters – `G`, `S` and `P`:

- `G` stands for the specified type.
- `S` formalizes the "S" in the string representation.
- `P` formalizes the "P" in the string representation.

The S and P parameters are constrained: they must be conditions over the specified type G. Requiring these types to be expanded allows them to have default objects; the benefits of this possibility are coming shortly.

Lines 13-28 implement the `verify` routine that captures the SRP's formal semantics as a specification driver. The routine accepts a formal argument of the specified type and expresses the SRP's semantics in terms of this variable. Lines 16-17 require the stimulus to hold through the precondition, where:

- `p_holds` is a tag for easier debugging.
- `({P}).default` returns the default object of type `P`.
- `holds` is a deferred Boolean function declared in class `CONDITION`, from which `P` inherits.
- The `({P}).default.holds (system)` assertion requires the stimulus to hold for `system`, the formal parameter of `verify`.

Lines 19-20 initialize the timer variable declared in the parent `REQUIREMENT` class. Lines 21-22 capture the response `S` through the loop exit condition. Lines 23-24 modify the system's state, where:

- The `iterate` command is implemented in the `REQUIREMENT` class.
- `iterate` calls deferred command `main` of that class and decreases the timer.
- `main` is deferred for being implemented in concrete SOORs inheriting from the SOORT.

Lines 25-26 guarantee termination of the loop through the timer used as the loop variant. The `verify` routine, when called appropriately on a SOOR implementing the template, becomes a test method; this maps to the “Verifying through testing” activity (Section 3.4.4). When submitted to AutoProof, it becomes a Hoare logic theorem capturing the requirement’s correctness axiom; this maps to the “Verifying through program proving" activity (Section 3.4.5).

String function `requirement_specific_ouput` on lines 32-35 returns the SRP-dependent natural language representation. The `REQUIREMENT` class implements, among other features, string function `out` which, in its turn, takes the value of `requirement_specific_ouput` and embeds it into the SRP-independent natural language representation. The SRP-independent part includes the name of the requirement derived from the SOOR’s class name, the name of the specified type and the real-time constraint.

The `time_boundary` function returns the default time boundary for finite state verification. This value comes from the `REQUIREMENT` ancestor class and is set to `{INTEGER}.max_value`, the maximum integer avaiable on the current system. Concrete SOORs may override this default. The verification process will simulate up to that number of executions of `iterate` to observe the required response. If the response is not observed after the last iteration, the verification process will fail.

5.2 ADT templates for software components

Figure 5.2 depicts the SOORT capturing the Binary Tree ADT specification with the `in_ord` function (Appendix B.4). The class consists of the following important parts:

- Line 2 in Figure 5.2 provides a general description of the template.
- Lines 3-4 provide a hyperlink to the source of the specification in the literature.
- Line 5 provides a named hyperlink to the location of the class on GitHub.
- Lines 7-8 declare type `B` intended to behave as a binary tree containing elements of `I`; `Q` stands for the queue type returned by the `in_ord` function. To show that `Q`, indeed, behaves as a queue of `I`s, the template’s implementers must supply type `QS` conforming to the `QUEUE_WITH_APPEND_ADT` template applied to `Q` and `I`.
- Lines 13-15 reflect the fact that the current SOORT inherits the regular binary tree behavior.
- Lines 17-22 declare the new function, `in_ord` (“in order”).
- Lines 24-50 state the ADT axioms due to the new function.
- Lines 52-61 state the well-definedness axiom for the contract of `in_ord`.

Unlike the SOORTs for control software (Section 5.1), ADT SOORTs have several specification drivers for verification. In the example on Figure 5.2, the new SOORT adds two axioms, `a_11` and `a_12`, to the axioms inherited from the parent SOORT. Because the new ADT declares another function, `in_ord`, these new axioms are required

```
note
  description: "Reusable abstract data type specification of binary tree with ‘‘inord’’ operation."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y3peoll5"

deferred class
  BINARY_TREE_WITH_INORD_ADT [B, I, Q, QS −>QUEUE_WITH_APPEND_ADT [Q, I]]
  -- Binary trees ‘‘B’’ contain elements of ‘‘I’’.
  -- They rely on queues ‘‘Q’’ with elements of ‘‘I’’ conforming to the
  -- ‘‘QUEUE_WITH_APPEND_ADT’’ specification.

inherit

  BINARY_TREE_ADT [B, I]

feature
  -- Deferred definitions.

  in_ord (b_tree: B): Q
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_11
    local
      b_tree: B
    do
      b_tree := empty_tree
      check
        in_ord (b_tree) ~ ({QS}).default.newq
      end
    end

  frozen a_12 (b_tree_left: B; item: I; b_tree_right: B; q_left, q_right: Q)
    require
      in_ord (b_tree_left) ~ q_left
      in_ord (b_tree_right) ~ q_right
    local
      b_tree: B
    do
      b_tree := make (b_tree_left, item, b_tree_right)
      ({QS}).default.addq (q_left, item)
      ({QS}).default.appendq (q_left, q_right)
      check
        in_ord (b_tree) ~ q_left
      end
    end

feature
  -- Well-definedness axioms.

  frozen in_ord_well_defined (b_tree_1, b_tree_2: B)
    require
      b_tree_1 ~ b_tree_2
    do
    ensure
      in_ord (b_tree_1) ~ in_ord (b_tree_2)
    end

end
```

Figure 5.2: The SOORT for the ADT specification of binary tree with function “in order” Appendix B.4). It inherits specification drivers of the BINARY_TREE_ADT SOORT encoding the ADT specification of binary tree without that function.

for completeness of the resulting ADT specification. The `in_ord_well_defined` auxiliary axioms requires the contract of `in_ord` to be well-defined. Well-definedness axioms apply only to verification with AutoProof: they make sure that the respective features' contracts are strong enough to maintain the equivalence classes.

To specify a SOOR stating that objects of a custom type `T` behave as binary trees with elements of `E`, convertible to queues `F` with elements of `E`:

1. Inherit from the `BINARY_TREE_WITH_INORD_ADT` class with `T` for `B`, `E` for `I`, `F` for `Q`.
2. For `QS`, provide a SOOR that specifies `F` as queue with elements of `E`.
3. Implement the SOORT's deferred definitions in terms of the features of types `T`, `E` and `F`.

To verify correctness of `T` against the binary tree axioms:

1. Run the specification drivers with names `a_*` on relevant test input (that is, satisfying the specification driver's precondition) if you practice testing.
2. If you practice program proving, submit the resulting SOOR to AutoProof.

ADT SOORTs differ conceptually from control software SOORTs with their extensibility. If a variation of an ADT emerges, then making the new SOORT a subclass of the original ADT's SOORT will automatically include its specification drivers. SOORTs for control software have finer granularity: they represent finalized behavioral patterns.

Chapter 6

Navigating the Solution

To maximize understanding of the SOOR approach, the dissertation presents the idea incrementally. The multirequirements approach serves as the starting point. Every important idea that underpins the SOOR approach builds on top of the previous one. At the same time, each idea is practically applicable alone, regardless of the ideas building on top of it.

Chapter 7 demonstrates practical applicability of seamlessness in the sense of multirequirements [NMR15] by applying it to a well-known example from the requirements literature [JZ95]. The resulting specification relies on the Mathematical Model Library (MML) [PTF18] – a library of immutable classes used in model-based contracts [SWM04].

Multirequirements rely on contracts to achieve seamlessness, but contracts suffer from the incompleteness problem [SWM04]. Chapter 8 proposes an AutoProof-based reasoning framework relying on the notion of specification driver to achieve contracts' correctness and well-definedness [NM16], which maps to steps 1-6 of the "Verifying through program proving" activity (Section 3.4.5). Chapter 9 additionally describes an AutoProof-driven technique for catching inconsistent contracts through specification drivers [Nau18], which maps to steps 7-9 of the "Verifying through program proving" activity (Section 3.4.5).

Chapter 10 improves multirequirements by replacing contracts with specification drivers that do not suffer from the problems of contracts. The two specification approaches remain fundamentally connected through AutoProof, however. The chapter describes an example of an incremental AutoProof-driven software process relying on seamless requirements [NM17]. Contracts remain vital part of the process, but they move from the problem space to the solution space. Seamless requirements serve as a driving force for specifying good contracts with the help of AutoProof. The chapter presents a complete software process detailing the "Verifying through program proving" activity (Section 3.4.5).

As empirical results demonstrate, frequently recurring verification-oriented SRPs often take the form of temporal properties [DAC99] and timing constraint [KC05] in the control software domain. How good are seamless requirements for specifying such requirements? Chapter 11 presents AutoReq – an AutoProof-driven approach that extends the notion of seamless requirements for specifying and verifying temporal properties and timing constraints of control software [Nau+19]. AutoReq identifies and emphasizes the fundamental connection between requirements expressiveness, verifiability and reusability. In AutoReq, the three aspects reinforce each other. The chapter provides detailed principles behind the "Specify a SOOR" (Section 3.4.2) and "Verify through program proving" (Section 3.4.5) activities in the context of temporal properties and timing constraints.

The recurring SRPs [DAC99], [KC05] raise a question: how to make these SRPs reusable across projects while keeping the expressiveness and verifiability introduced by specification drivers, seamless requirements and AutoReq? Chapter 12 not only makes seamless requirements reuse-oriented, but also adds a round trip requirements engineering mechanism relying on formal picnics into the very notion of requirement [Nau19a]. The chapter presents a ready-to-use library of Eiffel SOORTs encoding the known verification-oriented SRPs [DAC99], [KC05]. The classes contain reusable features for performing formal picnics and verifying candidate implementations. The chapter maps to the "Specify a SOOR" (Section 3.4.2), "Have a formal picnic" (Section 3.4.3) and "Verify through testing" (Section 3.4.4) activities. SOOR-based development through testing conceptually builds on top of paramenterized test-driven development (PTDD) [DTS10], adding the possibility to test temporal properties and timing constraints – types of requirements not covered by the PUT-based approaches.

Appendix A and Appendix B provide the full collection of the SOORTs for control software and software components, respectively.

Chapter 7

Unifying Requirements and Code: an Example

Requirements and code, in conventional software engineering wisdom, belong to entirely different worlds. Is it possible to unify these two worlds? A unified framework could help make software easier to change and reuse. To explore the feasibility of such an approach, the case study reported here takes a classic example from the requirements engineering literature and describes it using a programming language framework to express both domain and machine properties. The chapter describes the solution, discusses its benefits and limitations, and assesses its scalability.

7.1 Introduction

According to the standard view in software engineering, the tasks of requirements, design and implementation require distinct techniques and produce different artifacts.

What if instead of focusing on the differences we recognized the fundamental unity of the software construction process through all its stages? The principle of "seamlessness" (see e.g. [Mey97]) follows from this assumption that the commonalities are more fundamental than the differences, and that it pays to use the same set of concepts, notations and tools throughout the development, from the most general and user-oriented initial steps down to the most technical tasks.

A consequence of the seamlessness principle is that requirements are just another software artifact, susceptible to many of the same techniques as code and design. Assuming a modern programming language with powerful abstraction facilities, the requirements can be written in the same notation as the program.

The notion of multirequirements [Mey13] adds to this principle the idea of using several interleaved descriptions: natural language, graphical, and formal (Eiffel text) serving as the reference.

How realistic is the seamless multirequirements approach, what are its limits, and what benefits does it bring? To help answer this question, the present chapter takes the example used in a classic paper of the requirements literature, Jackson's and Zave's zoo software controller, and describes it entirely in a seamless style, including the key formal constraints of the example.

The goal of the chapter is not advocacy but experimentation. The advocacy is present in the earlier references cited above. We practice a seamless approach to software construction and consider it fruitful, but the present discussion does not attempt to establish its superiority; rather it starts from the seamlessness hypothesis – the hypothesis that a single notation, Eiffel, is applicable to requirements analysis just as much as to programming – and applies this hypothesis fully and consistently to a significant example. While we draw some conclusions, the important part is the result of the experiment as presented here, enabling readers to form their own conclusions as to the benefits and limits of the approach.

Section 7.2 briefly explains why it is interesting to put into question the traditional separation between software development tasks. Section 7.3 proposes an approach to unify software development tasks by combining the approaches described in [Mey13] and [JZ95]. Section 7.4 introduces some theoretical and technical background. Section 7.5 presents the approach applied to an example. Finally, Section 7.6 concludes and mentions future work.

Summary of contributions

Experimentation mentioned at the end of Section 7.1 resulted in the following key outcomes.

- An evidence suggesting that it is possible to use Multirequirements approach [Mey13] for describing cyber-physical systems like zoo turnstile controller. At the same time, different types of exemplar statements go far beyond just the relational statements used in [Mey13].
- An evidence suggesting that a real programming language notation may be even more expressive than most of the popular formal notations. Section 7.5.5 contains all the details.
- An example showing how object orientation helps to effectively manage complexity in specifications. The approach used in [JZ95], where the specification is basically a linear list of statements, does not scale to the case of large systems, when the number of requirements is too big. Object orientation provides a way to relate the conceptual objects so that the resulting specification will be scattered across the classes in an intuitive way.

7.2 The drawbacks of too much separation of concerns

Historically, there was a reason for emphasizing the distinction between development tasks. The goal was to highlight the specific needs of requirements and design, moving away from the "code first, think later" way of building software. But as the precepts of software engineering have gained wide acceptance and programming languages have moved from low-level machine-coding notations to descriptive formalisms with high expressive power, the reverse approach is worth exploring: instead of emphasizing the differences, show the fundamental unity of the software process.

The traditional approach is subject to five criticisms.

i Insufficient information. Requirements analysts do not know what details are important for developers. They are good at expressing customer needs in a form the customer is ready to sign, but they typically do not know what is implementable and what is not. [Mey85] discusses some typical flaws of natural language requirements specifications.

ii Lack of communication. When developers see ambiguous or contradictory elements in the requirements, they will not always go back and ask, but will often interpret the requirement according to their own understanding, which may or may not coincide with user wishes.

iii Impedance mismatches [Mey97]. The use of different formalisms at different stages requires translations and creates risks of mistakes.

iv Impediment to change. With different formalisms, it is difficult [Mey97] to ensure that a change at one level is reflected at other levels.

v Impediment to reuse. The presence of requirements as a document specific to each project may mask the commonality between projects and make the team miss potential reuse of existing developments.

7.3 A seamless approach

7.3.1 Unifying processes

Consideration of the problems listed above leads to trying a completely different approach, which recognizes that beyond the obvious differences between tasks of software development they share fundamental needs, concepts, principles, techniques, and can be addressed through a common notation. Modern programming languages are not just coding tools to talk to a machine, but powerful tools for expressing abstract concepts and modeling complex systems. The Eiffel notation used in the experiment uses object-oriented principles of classes, genericity, polymorphism and inheritance, which have proved adept at describing sophisticated systems (independently of their technical programming aspects) in a modular, flexible, reusable and evolutionary way. Thanks to the presence of Design by Contract mechanisms, it can describe not only the structure of systems but their abstract semantics.

7.3.2 The hypothesis

The hypothesis explored in the experiment, in light of the above analysis, is that it is possible to design a software development process that:

i Uses for requirements the same notation and tools as for design and implementation.

ii Links the resulting documents (requirements, design, code) together, ensuring a major goal of software engineering: traceability.

iii Makes it possible to prove, formally, the correctness of the implementation against the specification.

iv Supports extendibility by ensuring that small changes in the requirements will cause a proportionally small change in the design and the implementation.

7.3.3 How to test the hypothesis

The experiment relies on the following scenario for testing the preceding hypothesis at least in part:

i Propose a candidate process.

ii Select examples and apply the process.

iii Analyze the outcome.

[Mey13] sketches such a process, based on using object orientation for representing the relationships between the conceptual objects in the requirements document. The basic idea was to have an object-oriented code along with the natural language description of a requirement. It is also possible to represent each code fragment graphically as a BON diagram [WN94].

[Mey13], however, uses as example the very notion of requirements process. In other words, it is self-referential. This confers (we hope) a certain elegance to the example, but makes it look artificial. For our experiment we take a more standard example, coming from a classic requirements paper by Jackson and Zave [JZ95].

More precisely, the requirements from the example are represented using the model-based [PTF18] contracts-equipped [Mey09] object-oriented [Mey97] notation (Eiffel).

7.4 Theoretical and technical background

7.4.1 Design By Contract

Work [Mey09] gives a comprehensive description of Design By Contract. Design By Contract integrates Hoare-style assertions [Hoa69] within object-oriented programs [Mey97] constraining the data that run time objects hold. This approach equips each class feature (member) with a predicate expression, that specify its behavior, in the form of pre- and postcondition. The postcondition has to hold whenever the precondition held and the feature finished its computation before the program execution process invokes the next feature. Design By Contract equips the class itself with an invariant predicate expression which holds in all states of the corresponding objects.

7.4.2 Model-based contracts

If classical contracts are for constraining the data that run time objects actually hold, model-based contracts are “meta” contracts for constraining the objects as mathematical entities (sets, sequences, bags, relations etc.), and an execution process does not instantiate the corresponding mathematical representations at run time as parts of the objects. Model-Based Contracts are useful when it is not possible to capture all the nuances by means of classical contracts. Works [SWM04] and [PTF18] give some examples of such situations and a comprehensive description of the concept.

7.4.3 AutoProof

The AutoProof [Tsc+15] tool is capable of formally proving the correctness of contract-equipped object-oriented programs, both classical and model-based. AutoProof proves for every routine that the conjunction of the precondition and the class invariant before invocation ensures the conjunction of the postcondition and the class invariant after invocation. The class is verified if and only if all the class features are verified.

7.5 Unifying the two worlds: an example

Avoiding the problems analyzed in Section 7.2 means unifying the worlds of requirements and code in a unified framework. This section illustrates the approach. It takes the example from the work [JZ95] and shows how to express requirements of various types in the style of work [Mey13] – namely, using Eiffel as a formal specification language for expressing each requirement. Originally the authors used this example to demonstrate the process of deriving specifications from requirements, and the unified approach captures all the nuances of this process.

7.5.1 Example overview

The authors of [JZ95] start with giving the overall context: *“...Our small example concerns the control of a turnstile at the entry to a zoo. The turnstile consists of a rotating barrier and a coin slot, and is fitted with an electrical interface...”* This small paragraph mostly describes the relationships between the conceptual objects. Figure 7.1 contains specification of the context in the style of work [Mey13].

Translating the specification from Figure 7.1 back to natural language using the object-oriented semantics results in almost the same initial description: “A `ZOO` has a `TURNSTILE` turnstile; a `TURNSTILE` has a `COINSLOT` coinslot and a `BARRIER` barrier so that coinslot has Current `TURNSTILE` as turnstile and barrier has `Current` `TURNSTILE` as turnstile...” `COINSLOT` and `BARRIER` hold references to the `TURNSTILE` instances in order to capture the *“electrical interface”* phenomena: the word “interface” means something over which the parties are able to communicate with each other; communicating means sending messages to each other, and to send message to someone in the object-oriented world is to take a reference to the object and perform a qualified call on it. So at the very least the parties should hold references to each other to be able to communicate in two directions.

```
class ZOO
feature
  turnstile: TURNSTILE
end

class TURNSTILE
feature
  coinslot: COINSLOT
  barrier: BARRIER
invariant
  coinslot.turnstile = Current
  barrier.turnstile = Current
end
```

```
class COINSLOT
feature
  turnstile: TURNSTILE
invariant
  turnstile.coinslot = Current
end

class BARRIER
feature
  turnstile: TURNSTILE
invariant
  turnstile.barrier = Current
end
```

Figure 7.1: Expressing the context formally.

- $Push(e)$: In event e a visitor pushes the *barrier* to its intermediate position
- $Enter(e)$: In event e a visitor pushes the barrier fully home and so gains entry to the *zoo*
- $Coin(e)$: In event e a valid coin is inserted into the *coin slot*
- $Lock(e)$: In event e the *turnstile* receives a locking signal
- $Unlock(e)$: In event e the *turnstile* receives an unlocking signal

Figure 7.2: The Zoo Turnstile example designation set.

7.5.2 The designation set

After stating the problem context the authors of [JZ95] describe the *designation set*. Each designation basically corresponds to a separate type of events observed in the problem area. The authors give the designations as a set of predicates as in Figure 7.2. Figure 7.3 is an Eiffel implementation of each designation set described in Figure 7.2. The implementation uses Eiffel features names as labels for the events types. The natural language descriptions from Figure 7.2 provide heuristics on which feature should be added to which class (Figure 7.2. Each event type has an associated history – a sequence of moments in time when the events of this particular type occurred. For example, `enters: MML_SEQUENCE[INTEGER_64]` (in Figure 7.3) is a sequence of moments in time expressed in milliseconds when events of type `enter` took place. `MML_SEQUENCE` is a class from the `MML` (Mathematical Modeling Library) and denotes mathematical sequence. `MML` contains special classes for expressing model-based contracts. Although it is possible to instantiate some objects from these classes (like a sequence containing one element), the instances will not be modifiable. The **`model`** annotation is the Eiffel mechanism to represent model-based contracts (introduced in Section 7.4.2). For instance, expression

```
note
  model: enters
deferred class ZOO
feature
  enter
    deferred
    ensure
      enters.but_last ~ old enters
      enters.last > old enters.last
    end
  enters: MML_SEQUENCE[INTEGER_64]
end

note
  model: locks, unlocks
deferred class TURNSTILE
feature
  lock
    deferred
    ensure
      locks.but_last ~ old locks
      locks.last > old locks.last
    end
  unlock
    deferred
    ensure
      unlocks.but_last ~ old unlocks
      unlocks.last > old unlocks.last
    end
  locks: MML_SEQUENCE[INTEGER_64]
  unlocks: MML_SEQUENCE[INTEGER_64]
end
```

```
note
  model: coins
deferred class COINSLOT
feature
  coin
    deferred
    ensure
      coins.but_last ~ old coins
      coins.last > old coins.last
    end
  coins: MML_SEQUENCE[INTEGER_64]
end

note
  model: pushes
deferred class BARRIER
feature
  push
    deferred
    ensure
      pushes.but_last ~ old pushes
      pushes.last > old pushes.last
    end
  pushes: MML_SEQUENCE[INTEGER_64]
end
```

Figure 7.3: Specifying the designation set formally.

`model: enters` in Figure 7.3 says that `enters` is a model query. The `enters` query models the sequence of timestamps corresponding to moments when people enter the zoo.

The `deferred` keyword states that the specification gives only formal definitions of the events (in terms of pre- and postconditions [Hoa69]) and does not give the corresponding operational reactions of the machine on the events. The `ensure` clause is the postcondition of the feature. It describes how the system changes after reacting on an event of the corresponding type. These specifications are intuitively plausible: an event occurrence should result in extending the corresponding history with the moment in time when the event took place, and the time of the new event should be strictly bigger than the time of the previous event, as shown, for instance, by the postcondition in feature `unlock` of Figure 7.3. The keyword `old` is used to indicate expressions that must be evaluated in the pre-state of the routine, and "~" makes a comparison by value.

7.5.3 Shared phenomena

The authors of [JZ95] introduce the notion of shared phenomena – that is, the phenomena visible to both the world (the environment) and the machine (the notions of the world and the machine were introduced by Jackson in [Jac95]). In the multirequirements approach this notion is covered by using the "has a" relationships between the `ZOO` and the `TURNSTILE` classes, accompanied with the model-based contracts. Namely, since a `ZOO` has a turnstile as its feature, it can see any phenomena hosted by the turnstile: `locks`, `unlocks`, `coins`, `pushes`; since a `TURNSTILE` does not hold any references to a `ZOO`, it can not observe nor control the `enter` events modeled by `ZOO`.

7.5.4 Specifying the system

Work [JZ95] introduces a set of criteria by means of which it is possible to identify whether the machine is specified or not. One of the criteria states that all requirements should be expressed in terms of shared phenomena only. Requirements refinement is the process of converting the requirements stated in terms of both shared and non-shared phenomena to the form in which they are expressed in terms of shared phenomena only. Refinement process consists of identifying some laws, which hold in the environment regardless of the machine behaviour, and constraining the machine behaviour. The resulting constraints imposed on the machine together with the laws of the environment should logically imply the requirements stated in the beginning.

The authors of [JZ95] state that the laws of the environment are always expressed in the indicative mood, while the restrictions imposed on the machine behavior are expressed in the optative mood.

All properties of the problem derived in [JZ95] – be they optative or indicative descriptions – can be conceptually divided into the two main categories.

Properties which hold at any moment in time:

an example of such property is the $OPT1$ requirement (expressed in Figure 7.4) saying that entries should never exceed payments (the authors of [JZ95] use $OPT*$ for labeling properties expressed in an optative mood). Within the multirequirements approach this

```
deferred class ZOO
feature
  turnstile: TURNSTILE
  enters: MML_SEQUENCE[INTEGER_64]
invariant
  enters.count ≤ turnstile.coinslot.coins.count
end
```

Figure 7.4: Entries should never exceed payments.

```
deferred class BARRIER
feature
  push
    require
      not turnstile.unlocks.is_empty
      (not turnstile.locks.is_empty) implies
        (turnstile.unlocks.last > turnstile.locks.last)
    deferred
    end
end
```

Figure 7.5: It is impossible to use locked turnstile.

requirement can be expressed in the following way. The "something always holds" semantics fits perfectly into the semantics of Eiffel invariant: "something holds in all states of the object", as expressed in Figure 7.4.

Properties which hold depending on the type of the next event to occur:

the indicative property $IND2$ saying that it is impossible to push the barrier if the turnstile is locked will serve as an example (the authors of [JZ95] use $IND*$ for labeling properties expressed in the indicative mood). Figure 7.5 depicts the corresponding specification. The initial description is divided into the two different requirements:

1. The turnstile should have received at least one unlock signal.
2. If the turnstile has ever received lock signals, the most recent lock signal should be older than the most recent unlock signal.

If the two requirements hold together, the turnstile will be in the unlocked state.

Real time properties:

the authors of [JZ95] derive several timing constraints on the events processing. For example, the $OPT7$ requirement says that the amount of time between the moment when the number of the barrier pushes becomes equal to the number of coins inserted and the moment when the machine locks the turnstile should be less than 760 milliseconds. This is basically a constraint for the reaction on the *push* event: if the next *push* event

```
deferred class BARRIER
feature
  turnstile: TURNSTILE
  push
    deferred
    ensure
      ((old turnstile.unlocks.last > old turnstile.locks.last) and
       (pushes.count = turnstile.coinslot.coins.count)) implies
        (turnstile.locks.last > pushes.last and
          (turnstile.locks.last − pushes.last) <760)
    end
  pushes: MML_SEQUENCE[INTEGER_64]
end
```

Figure 7.6: The machine locks the turnstile timely.

```
deferred class ZOO
feature
  turnstile: TURNSTILE_ABSTRACT
  enter
    deferred
    end
  enters: MML_SEQUENCE[INTEGER_64]
invariant
  turnstile.coinslot.coins.count > enters.count implies (agent enter).precondition
end
```

Figure 7.7: The turnstile let people who pay enter.

uses the last coin, the machine should ensure that the turnstile is locked in a timely fashion, so that a human being will not have time to enter without paying. The 760 quantity reflects the fact that it takes at least 760 milliseconds for a human being to rotate the barrier completely and enter the Zoo.

Taking this reasoning into consideration, the multirequirements specification approach handles the timing constraint by putting it into the `push` feature postcondition (as depicted in Figure 7.6). The antecedent of the implication assumes the situation when before the *push* event the turnstile was locked (`old turnstile.unlocks.last > old turnstile.locks.last` expression in Figure 7.6), and after the event occurrence the number of barrier pushes became equal to the number of coins inserted (`pushes.count = turnstile.coinslot.coins.count` expression in Figure 7.6). The consequent reflects the requirement that, having in place the situation that the antecedent describes, there should be a *lock* event which is more late than the last *push* event (`turnstile.locks.last > pushes.last` expression in Figure 7.6), and the distance between them should be less than 760 milliseconds (`(turnstile.locks.last − pushes.last)<760` expression in Figure 7.6).

7.5.5 Specifying the "unspecifiable"

One of the requirements mentioned in [JZ95] was *OPT*2 saying that the visitors who pay are not prevented from entering the Zoo. The authors give only informal statement of this requirement: $\forall\, v, m, n \bullet ((\mathit{Enter}\#(v,m) \wedge \mathit{Coin}\#(v,n) \wedge (m < n)) \Rightarrow$ *"The machine will not prevent another Enter event″*.

The antecedent of this implication should be read like "the number of entries is less than the number of coins inserted". The authors of [JZ95] do not formalize the consequent and leave it in the natural language form. The multirequirements specification approach handles this requirement using standard Eiffel mechanism called *agents* (see Figure 7.7).

The `agent` clause treats a feature (the `enter` feature in this particular case) as a separate object so that the feature precondition becomes one of the boolean-type features of the resulting object.

7.6 Summary

Software construction involves different activities. Typically these activities are performed separately. For instance, requirements and code, as developed nowadays, seem to belong to different worlds. The presented experiment shows the feasibility of unifying requirements and code in a single framework.

In this experiment, we take the classic Zoo Turnstile example [JZ95] and implement it using Eiffel programming language. Eiffel is used not just to express the domain properties but also the properties of the machine [Jac95], enabling users to combine requirements and code in a single framework. The complete implementation of the example can be reached in the GitHub project [Naua].

The multirequirements approach is suitable not only for formalizing the statements that [JZ95] formalizes, but also for formalizing those which are not possible to formalize with classical instruments like predicate or temporal logic (like *OPT*2 requirement, see Figure 7.7).

The multirequirements approach is not only expressively powerful – it enables smooth transition to design and implementation. GitHub project [Naua] contains a continuation of the experiment in the form of a complete implementation of the Zoo Turnstile example.

In order to understand the benefits of the multirequirements approach better it seems feasible to evaluate it against the hypothesis stated in Section 7.3.2:

i Unity of software development tasks: indeed, all the code fragments corresponding to different specification items merged together will bring a complete design solution available at [Naua] (the classes ending with `_ABSTRACT`).

ii Traceability between the specification and the implementation: the classes ending with `_CONCRETE` available at [Naua] contain the implementation and relate to the specification classes by means of inheritance.

iii Provability of the classes: the AutoProof system [Tsc+15] is capable of formally proving both classical and model-based contracts in Eiffel. However, it is not yet capable of proving "higher-level" agents-based contracts like the one used in Figure 7.7 for expressing requirement $OPT2$ from the work [JZ95]. Adding this functionality to AutoProof is one of the next work items.

iv Extendibility of the solution: since Eiffel artifacts used in the formalizations of the requirements items correspond to their natural language counterparts, it is visible right away how a change in one representation will affect the second.

Speaking about scalability of the approach, a formal representation of a requirements item specified with Eiffel is as big as the scope of the item and its natural language description are, so the overall complexity of the final document should not depend on the size of the project. Anyway, this is something to test by applying the approach to a bigger project.

The future actions plan include:

i to prove formally that the features' specifications are consistent – that they preserve the invariants of their home classes, and the invariants by themselves are consistent. For example, it should not be possible for $P(x)$ and $\neg P(x)$ to hold at the same time.

ii to design machinery for translating model-based contract-oriented requirements to their natural language counterpart so that the result will be recognizable by a human being.

iii to apply the approach to a bigger project.

iv to extend AutoProof technology [Tsc+15] so that it will be able to handle agents in specifications (like in Figure 7.7).

It seems feasible to utilize AutoProof technology [Tsc+15] for achieving goal (Item i). AutoProof is already capable of proving that a feature implementation preserves its specification (except specifications with agents), and it seems logical to empower it with the capabilities for working solely on the specifications level. Work [Nor09] contains a formal proof that it is possible to achieve goal (Item iv).

As a result of implementing the plan a powerful framework for expressing all possible views on the software under construction should emerge. The threshold of success includes the possibility to generate the specification classes (their names end with _ABSTRACT) available at [Naua] automatically, using requirements documents produced according to the multirequirements process as input.

Chapter 8

Making Contracts Complete

Existing techniques of Design by Contract do not allow software developers to specify complete contracts in many cases. Incomplete contracts leave room for malicious implementations. This chapter complements Design by Contract with a powerful technique that removes the problem without adding syntactical mechanisms. The proposed technique makes it possible not only to derive complete contracts, but also to rigorously check and improve completeness of existing contracts without instrumenting them.

8.1 Introduction

The main contribution of this chapter is a new approach to seamless software development, bridging the heretofore wide gap between two fundamental and widely used techniques: Abstract Data Types (ADTs) and Object-Oriented Programming (OOP). These techniques seem made for each other, but trying to combine them in practice reveals a glaring impedance mismatch. We explain the problem, provide a remedy, and subject it to formal verification.

ADTs [GH78] are a clear, widely known way to specify systems precisely.
OOP [Mey97] is the realization of ADT ideas at the design and programming level, with Design by Contract (semantic properties embedded in the program) providing the connection. At least, that is the accepted view. However, the correspondence is far less simple than this view would suggest. While it would seem natural to use ADTs for specification and OOP for design and implementation, in practice this combination hits an impedance mismatch:

- At the ADT level, some axioms involve two or more commands. For example, an axiom for stacks (the standard example of ADTs, which remains the best for explanatory purposes) will state that if you push an element onto a stack and then pop the stack, you end up with the original stack.

- In a class, the standard unit of OOP, the contracts can only talk about one command, such as push or pop, but not both. Specifically, the postcondition of a command such as push can describe the command's effect on queries such as top (after you have pushed an element, that element is the new top), but there is no way to refer to the effect on pop as expressed by the ADT axiom.

The present chapter introduces a practical solution to this mismatch. The essence of the solution is that classes reflecting ADTs, such as a class STACK, cannot by themselves capture such multi-command (or "second-degree") ADT axioms, but this does not mean that the OOP approach fails us. The idea will be to introduce auxiliary classes whose role is to "talk about" the features of the basic classes such as STACK (the ones corresponding to ADTs). Such a class has features that combine those of basic classes, e.g. a command `push_then_pop` that works on an arbitrary stack, pushing an element on a stack and then popping the stack. Then the postcondition of `push_then_pop` can specify that the resulting stack is the same as the original.

We call such features specification drivers by analogy with "test drivers", which are similarly added to the basic units of a system for the sole purpose of testing them. Like test drivers, specification drivers serve purely verification purposes, rather than providing system functionality. The difference is of course that test drivers appear in dynamic verification (testing), whereas specification drivers are for static verification (for example, as in this chapter, correctness proofs). But the basic idea is the same.

Specification drivers are not just a specification technique; we also submit them to formal, mechanical verification. As part of the AutoProof formal verification tool [Tsc+15], we have mechanically proved the correctness of the presented examples.

Section 8.2 explains the problem through a working example. Section 8.4 describes the essentials of the solution. Section 8.5 compares this approach with other possible ones. Section 8.6 presents our experience with mechanical verification. Section 8.7 draws conclusions and outlines future research prospects.

8.2 Motivating example

Figure 8.1 contains the standard ADT specification of stacks. The standard names of the functions are changed in favor of the mechanical verification experiment in Section 8.6: the existing implementation, to which the experiment is applied, uses exactly these names.

Figure 8.2 contains the result of applying the traditional process of DbC [Mey97] to the specification in Figure 8.1:

- The name of the class is derived from the name of the ADT it implements.
- The signatures of the implementation features are derivatives of the ADT functions' descriptions.
- Preconditions of the ADT functions go to `require` clauses of the implementation features.
- Postconditions of the implementation features capture ADT axioms A1, A3 and A4.
- The `create` clause lists the implementation feature `new` to highlight its special mission of instantiating new stacks.

TYPES

- $STACK[G]$

FUNCTIONS

- $extend : STACK[G] \times G \rightarrow STACK[G]$
- $remove : STACK[G] \nrightarrow STACK[G]$
- $item : STACK[G] \nrightarrow G$
- $is\_empty : STACK[G] \rightarrow BOOLEAN$
- $new : STACK[G]$

AXIOMS
For any $x : G$, $s : STACK[G]$

(A1) $item(extend(s,x)) = x$

(A2) $remove(extend(s,x)) = s$

(A3) $is\_empty(new)$

(A4) **not** $is\_empty(extend(s,x))$

PRECONDITIONS

(P1) $remove(s : STACK[G])$ **require not** $is\_empty(s)$

(P2) $item(s : STACK[G])$ **require not** $is\_empty(s)$

Figure 8.1: ADT specification of stacks.

```
class STACK_IMPLEMENTATION [G] -- Type STACK[G]
create new -- Marking new as a creation feature
feature
  extend (x: G) -- Extending with a new element
    do
    ensure
      a1: item = x
      a4: not is_empty
    end

  remove -- Removing the topmost element
    require
      p1: not is_empty
    do
    end

  item: G -- The topmost element
    require
      p2: not is_empty
    do
    end

  is_empty: BOOLEAN -- Is the stack empty?

  new -- Instantiating a stack
    do
    ensure
      a3: is_empty
    end
end
```

Figure 8.2: Applying the traditional process of DbC to the stacks ADT specification.

Axiom A2 introduces the problem. The axiom constrains two functions simultaneously, *extend* and *remove*: the former one should do nothing but extend the stack with the given element, and the latter should do nothing but remove the topmost element of the stack. As a consequence, it is not possible to capture the axiom in a single implementation feature postcondition. Postconditions operate on two objects: the target object before calling the feature and the target object after invoking the feature. If the feature has formal parameters, they also parameterize the postcondition. Axiom A2 involves three stacks: the original one s, s_1 resulting from applying function *extend* to s, and finally s_2 resulting from applying *remove* to s_1. Formally:

$$\begin{aligned}&\forall\, s, s_1, s_2 : \mathit{STACK}[G];\ x : G \bullet\\ &\qquad\qquad (s_1 = \mathit{extend}(s,x) \wedge s_2 = \mathit{remove}(s_1) \Rightarrow s_2 = s\end{aligned}$$

Or, writing the quantified expression in terms of postconditions:

$$(\mathit{Post}_{\mathit{extend}}(s, s_1, x) \wedge \mathit{Post}_{\mathit{remove}}(s_1, s_2)) \Rightarrow s_2 = s \tag{8.1}$$

On one hand, it is not possible to capture A2 in a single postcondition. On the other hand, postconditions of `extend` and `remove` should exist and be strong enough to satisfy Equation (8.1).

Failures to capture such important properties as A2 in postconditions leave room for invalid implementations. The inability to capture axiom A2 allows for implementing stacks which store only the last added element and thus are useless as data containers. Still, such an implementation satisfies all the other axioms as its postconditions capture them.

Figure 8.3 depicts such an invalid implementation. For the sake of simplicity, it ignores preconditions, but this does not render the reasoning invalid: an empty precondition defaults to `TRUE`, the weakest conceivable precondition. According to the rule of consequence for preconditions [Hoa69], correctness against a weaker precondition implies correctness against a stronger one. Submitting the class `STACK_IMPLEMENTATION` to AutoProof confirms the point: the tool successfully proves “correctness” of the implementation.

For purist developers the problem of underspecified postconditions may become a reason for not using them at all. Intuitively, it seems better to keep all the properties written in a single place, and the described problem prevents doing this: although it is possible to capture some ADT axioms in postconditions, some of them will have to exist in separate documents and thus carry the risk of misuse and all the associated traceability costs.

8.3 Axioms as specification drivers

The example in Figure 8.2 translates axiom A1 to the postcondition of the implementation feature `extend`. Is it in fact the only way to do the translation of the axiom? A closer look at the original axiom and its translation in Figure 8.2 reveals two facts:

- The axiom uses the function *extend* in a sense of applying it, while its translation in Figure 8.2 specifies the implementation feature without invoking it.

```
class STACK_IMPLEMENTATION [G]
create new
feature
  extend (x: G) -- Extending with a new element
    do
      item := x
      is_empty := False
    ensure
      a1: item = x
      a4: not is_empty
    end

  remove -- Removing the topmost element
    do
      is_empty := True
    end

  item: G -- The topmost element

  is_empty: BOOLEAN -- Is the stack empty?

  new -- Instantiating a stack
    do
      is_empty := True
    ensure
      a3: is_empty
    end
end
```

Figure 8.3: Underspecified postconditions may lead to invalid implementations.

```
extend_updates_item (s: STACK_IMPLEMENTATION [G]; x: G)
  do
    s.extend(x)
  ensure
    s.item = x
  end
```

Figure 8.4: Axiom A1 as a specified feature.

```
remove_then_extend (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
  require
    s1.is_equal(s2)
  do
    s1.extend (x)
    s1.remove
  ensure
    s1.is_equal(s2)
  end
```

Figure 8.5: Axiom A2 as a specified feature.

- The axiom uses an explicit stack instance *s*, while the translation implicitly operates on the current object described by class `STACK_IMPLEMENTATION[G]`.

Is it possible to devise a translation of axiom A1 that would be closer to the origin?

Existing techniques of DbC completely ignore a large family of program constructs: features with pre- and postconditions whose only purpose is to serve as proof obligations. Such features do not implement any ADT functions and are not to be invoked. Instead, they are intended solely for static verification.

Figure 8.4 gives an example. The feature `extend_updates_item` is an alternative translation of axiom A1. It possesses the following properties:

- It operates on explicit objects `s` and `x`.
- It uses an explicit invocation of implementation feature `extend`.

The example in Figure 8.4 takes the whole feature `extend_updates_item` as the translation of the axiom, as opposed to the one in Figure 8.2, where the axiom is captured with the assertion `item = x` in the postcondition of implementation feature `extend`.

Using this approach, it is possible to capture axiom A2 in the form of the feature `remove_then_extend` in Figure 8.5. Again, the whole feature is the translation of the axiom. The feature `is_equal` defines an equivalence relation over run time objects representing stacks. It is declared by default in all Eiffel classes and compares its operands by value. The notion of equality deserves a separate analysis; Section 8.4.2 gives the details.

Henceforth, we will use the term *specification drivers* for specified features serving as translations of certain ADT axioms. A specification driver can be proven correct only if the implementation features it invokes have strong enough postconditions.

Consequently, specification drivers, as their name suggests, drive specifying stronger postconditions.

8.4 Specification drivers in practice

The present section derives the complete set of specification drivers for the stacks ADT (Figure 8.1). This set includes not only specification drivers that represent the original axioms of stacks because some specification drivers stem from a fundamental difference between ADT specifications and object-oriented programs: in the former it is not possible to have more than one occurrence of one and the same abstract stack, while in the latter it is possible to instantiate two run time objects denoting one and the same abstract stack. Section 8.4.2 and Section 8.4.3 discuss the issue in detail and derive additional specification drivers caused by it.

8.4.1 ADT axioms

Specification drivers do not bring any functional value to the system: they exist only to be eventually discharged as proof obligations. Consequently, they should not pollute implementation classes like `STACK_IMPLEMENTATION` in Figure 8.2. Concerning where to store them, the simplest option is to create a separate class within the source code project. The
`ADT_AXIOMS_SPECIFICATION_DRIVERS` class in Figure 8.6 contains specification drivers capturing the ADT axioms of stacks. This class is generic: since it talks about instances of a generic concept, `STACK_IMPLEMENTATION [G]` in this case, it needs to assume existence of type `G` to keep the genericity. The `{NONE}` clause suggests that the features listed within the corresponding **`feature`** block do not supply any useful functionality. The **`deferred`** keyword in front of the class declaration suggests that it is not possible to instantiate any objects of this class, which makes sense as the class serves as a document containing specification drivers rather than a blueprint for creating run time objects.

8.4.2 Equivalence

It is possible to see that the specification drivers in Figure 8.6 use two different operators for objects comparison: $=$ and `is_equal`, while the original ADT specification in Figure 8.1 invokes only $=$. This section discusses the difference between comparing instances of ADTs and comparing objects instantiated from object-oriented classes and introduces a set of specification drivers capturing the difference.

ADT specifications operate on sets of instances in the mathematical sense of the word “set”: an abstract data type cannot contain two instances of one and the same abstract object. For example, the range of the function *new* consists of the only stack instance, which is the empty stack, as axiom A4 suggests. When an object-oriented program is running, it is perfectly fine for it to have two run time objects in its memory denoting one and the same instance of the ADT. For example, it is possible to declare two variables of type `STACK_IMPLEMENTATION [INTEGER]` and make them both refer to two different stack objects in the memory, as in Figure 8.7. Consequently, run time objects

```
deferred class ADT_AXIOMS_SPECIFICATION_DRIVERS [G]
feature {NONE}
  axiom_a1 (s: STACK_IMPLEMENTATION [G]; x: G)
    do
      s.extend (x)
    ensure
      s.item = x
    end

  axiom_a2 (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
    require
      s1.is_equal (s2)
    do
      s1.extend (x)
      s1.remove
    ensure
      s1.is_equal (s2)
    end

  axiom_a3 (s: STACK_IMPLEMENTATION [G]; x: G)
    do
      s.extend (x)
    ensure
      not s.is_empty
    end

  axiom_a4: STACK_IMPLEMENTATION [G]
    do
      create Result.new
    ensure
      Result.is_empty
    end
end
```

Figure 8.6: Specification drivers capturing the axioms of stacks.

```
s1, s2: STACK_IMPLEMENTATION [INTEGER]
create s1.new
create s2.new
```

Figure 8.7: Creating two instances of the empty stack.

```
deferred class EQUIVALENCE_SPECIFICATION_DRIVERS [G]
feature {NONE}
  reflexivity (s: STACK_IMPLEMENTATION [G])
    do
    ensure
      s.is_equal (s)
    end

  symmetry (s1, s2: STACK_IMPLEMENTATION [G])
    require
      s1.is_equal (s2)
    do
    ensure
      s2.is_equal (s1)
    end

  transitivity (s1, s2, s3: STACK_IMPLEMENTATION [G])
    require
      s1.is_equal (s2)
      s2.is_equal (s3)
    do
    ensure
      s1.is_equal (s3)
    end
end
```

Figure 8.8: Capturing the definition of equivalence.

form not a set of abstract objects, but a *multiset*, or *bag* [Bli89]. That is why there are two different comparison operators: the $=$ operator checks whether the operands refer to identical run time objects, and `is_equal` checks whether the objects referenced by the operands represent the same instance of the ADT implemented by the class. As a consequence, if specification drivers representing ADT axioms use the feature `is_equal`, the corresponding implementation class should redefine the feature and its postcondition should be strong enough to satisfy the definition of equivalence relations. A relation over stacks is an equivalence relation if and only if it possesses the following properties:

- Reflexivity: every stack is equal to itself.
- Symmetry: if stack s_1 is equal to stack s_2, then s_2 is equal to s_1 as well.
- Transitivity: if stack s_1 is equal to stack s_2, and s_2 is equal to s_3, then s_1 is equal to s_3.

As Figure 8.8 illustrates, the three properties may be captured by a separate class created specifically for this goal. If all the features of class `EQUIVALENCE_SPECIFICATION_DRIVERS` are correct, then the postcondition of `is_equal` indeed defines an equivalence relation over run time objects instantiated from `STACK_IMPLEMENTATION [G]`.

It is worth noting that because equivalence definition is static, specification drivers for equivalence may be generated automatically for every class.

8.4.3 Well-definedness

The ADT specification in Figure 8.1 lists certain functions over stacks. It is necessary to ensure that they maintain equivalence relations over stacks. Invoking a given implementation feature for two run time objects, which represent a single ADT object, should be indistinguishable from applying the ADT function implemented by this feature to that ADT object. Since a function application produces only one element from its range set, the two run time objects should also be considered equal after the invocation. This property is called *well-definedness* under an equivalence relation. The class `WELL_DEFINEDNESS_SPECIFICATION_DRIVERS` in Figure 8.9 contains specification drivers that encode well-definedness for every stack implementation feature. The specification drivers `item_is_well_defined` and `remove_is_well_defined` contain assertions **not** `s1.is_empty` and **not** `s2.is_empty`. These specification drivers invoke implementation features `item` and `remove`, which have preconditions that need to be satisfied. The purpose of the mentioned assertions is exactly this. The `s1` $\neq$ `s2` assertion in the precondition of the specification driver `remove_is_well_defined` is there for a very specific reason. If `s1` and `s2` are identical, the precondition for the `s2.remove` call may not hold: even if the stack object referenced by `s1` and `s2` is not empty in the beginning, it may not be the case anymore after the `s1.remove` call. This additional assertion does not remove any generality: indeed, identity always implies equality, and proving the latter is exactly the purpose of this specification driver, according to its postcondition.

Specification driver `new_is_well_defined` deserves special attention too. In fact, it encodes something stronger than just the well-definedness of the implementation feature `new`. It says that two empty stacks are always equal. This makes perfect sense and at the same time implies the necessary well-definedness property: from the ADT specification in Figure 8.1 and its first approximation in Figure 8.2, it is known that instantiating a stack with function *new* results in the empty abstract stack. Consequently, the `new_is_well_defined` specification driver covers this case, since it applies to every pair of run time objects denoting the empty abstract stack.

Similarly to equivalence, the notion of well-definedness is long-established; as such, it may be possible to generate the corresponding specification drivers automatically.

8.4.4 Complete contracts

Although some works ([PTF18], [SWM04]) talk about contract (in)completeness, they do not define this notion precisely. In light of the fundamental difference between ADT specifications and object-oriented programs, which causes the notion of equivalence over run time objects to appear (Section 8.4.2), the definition cannot be implicitly equal to the definition of sufficiently complete ADT specifications [GH78] and needs to be written down explicitly.

As the other details of the original definition in [Mey97] do not bring any value to the discussion, we use a simplified definition of a contract.

Definition 8.4.1 *A **contract** is a set composed of all pairs of the form* $(Precondition(f), Postcondtion(f))$ *for every implementation feature* f.

```
deferred class WELL_DEFINEDNESS_SPECIFICATION_DRIVERS [G]
feature {NONE}
  new_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
    require
      s1.is_empty
      s2.is_empty
    do
    ensure
      s1.is_equal (s2)
    end

  is_empty_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
    require
      s1.is_equal (s2)
    do
    ensure
      s1.is_empty = s2.is_empty
    end

  item_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
    require
      not s1.is_empty
      not s2.is_empty
      s1.is_equal (s2)
    do
    ensure
      s1.item = s2.item
    end

  extend_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
    require
      s1.is_equal (s2)
    do
      s1.extend (x)
      s2.extend (x)
    ensure
      s1.is_equal (s2)
    end

  remove_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
    require
      not s1.is_empty
      not s2.is_empty
      s1.is_equal (s2)
      s1 ≠ s2
    do
      s1.remove
      s2.remove
    ensure
      s1.is_equal (s2)
    end
end
```

Figure 8.9: Specification drivers for well-definedness.

This definition ignores the possible presence of class invariants as it is always possible to get rid of them by appending to pre- and postconditions of the implementation features.

Definition 8.4.2 *A contract is* ***correct*** *if and only if:*

- *Its postconditions are strong enough to ensure correctness of the specification drivers derived from the input ADT axioms (Section 8.4.1)*
- *In the event that specification drivers for the input ADT axioms use equivalence, its postconditions are strong enough to ensure correctness of the specification drivers for equivalence (Section 8.4.2).*

Definition 8.4.3 *A contract is* ***well-defined*** *if and only if its postconditions are strong enough to ensure correctness of the specification drivers for well-definedness (Section 8.4.3).*

Definition 8.4.4 *A contract is* ***complete*** *if and only if it is correct and well-defined.*

8.5 Related work

Work [PTF18] uses features with pre- and postconditions for checking completeness of model-based contracts (discussed later in this section). The definition of a complete model-based contract is not related to the definition of completeness in Section 8.4.4. According to [PTF18], completeness is what we call well-definedness, expressed in terms of abstract mathematical concepts.

Although the specification driver approach allows capturing ADT axioms in their original form, it does specify how to actually build complete contracts having a set of specification drivers. As Section 8.2 suggests, in many cases it is not possible to specify strong enough postconditions in terms of the ADT specification itself. This is where the need for representation appears: the implementation class has to stick to some already implemented data structure in order to enable stronger postconditions expressed it terms of this data structure. The problem of choosing an ideal representation has been aptly handled in multiple publications, therefore we do not propose our own methodology, but instead reference these publications.

Work [Mey03] shows that it makes sense to use mathematical abstractions for representations: for example, it seems reasonable to think about stacks as mathematical sequences. That work also shows how to prove correctness against contracts strengthened with precise mathematical abstractions. Work [SWM04] introduces the Mathematical Model Library (MML) – Eiffel library containing core abstractions: sets, sequences, bags, tuples etc. A later work [PTF18] introduces EiffelBase2, a usable library of essential data structures, including stacks, represented as mathematical abstractions from MML. EiffelBase2 is fully verified with the AutoProof verifier. The underlying verification methodology [Pol+14] assumes writing quite a number of assertions related to program execution semantics, so giving complete examples here would introduce confusion rather than clarity. Instead, Figure 8.10 presents the idea

```
class STACK_SEQUENCE_IMPLEMENTATION [G]
inherit ANY redefine is_equal end
create new -- Marking new as a creation feature
feature
  sequence: MML_SEQUENCE [G] -- Stack representation

  extend (x: G) -- Extending with a new element
    do
    ensure
      a1: item = x
      a4: not is_empty
      definition: sequence = old sequence.extended (x)
    end

  remove -- Removing the topmost element
    require
      not is_empty
    do
    ensure
      definition: sequence = old sequence.but_last
    end

  item: G -- Retrieving the topmost element
    require
      not is_empty
    do
    ensure
      definition: Result = sequence.last
    end

  is_empty: BOOLEAN -- Is the stack empty?
    do
    ensure
      definition: Result = sequence.is_empty
    end

  new -- Instantiating a stack
    do
    ensure
      a3: is_empty
      definition: sequence.is_empty
    end

  is_equal (other: STACK_SEQUENCE_IMPLEMENTATION [G]): BOOLEAN -- Redefining equality
    do
    ensure then
      definition: Result = (sequence.count = other.sequence.count and then
        (across 1 |..| sequence.count as i all sequence[i.item] = other.sequence[i.item] end))
    end
end
```

Figure 8.10: Abstract model of stacks as sequences.

in a nutshell. The `STACK_SEQUENCE_IMPLEMENTATION` class is the abstract model of stacks from the EiffelBase2 standpoint. EiffelBase2 equips classes implementing stacks with the `sequence` attribute and strengthens postconditions of the implementation features in terms of it. Class `MML_SEQUENCE` cannot be instantiated into any run time objects and exists only for verification purposes: it maps to the data structure representing mathematical sequences in the underlying proving engine. The `sequence` attribute is further connected to meaningful data structures by means of abstraction and refinement techniques [Hoa72]. Work [PTF18] gives more implementation details.

In Figure 8.10, the implementation features are formally defined with assertions over the `sequence` attribute (marked with the "definition" tag) added to the features' postconditions. The comparison feature `is_equal` is redefined so that two stacks are considered equal if and only if the sequences representing them are equal. Two sequences are considered equal if and only if their sizes are equal and they contain same objects. The feature `extended` models a sequence where an object is appended to the target sequence on to which the feature is invoked; feature `but_last` models the target sequence, but without the last element; feature `last` models the element added to the target sequence last; feature `is_empty` models the indication whether the target sequence is empty or not; finally, feature `count` models the size of the target sequence.

Mathematical concepts from MML are abstract, but they still form particular representations in EiffelBase2, though mathematically precise. The concept of model-based contracts helps to specify complete contracts, but does not say how to rigorously check contracts with representations for completeness. Furthermore, it fails to define what complete contracts are. The notion of specification drivers bridges this gap. All the specification drivers derived in the present chapter are expressed in terms of the original ADT specification (Section 8.4.1) plus the abstract equivalence (Section 8.4.2 and Section 8.4.3), whose presence is inevitable due to the nature of computing which allows programs to keep in their memory several instances of one and the same abstract object. They do not require making any assumptions about possible representations and enable defining complete contracts precisely.

8.6 Proving contracts completeness

It is possible to give a manual proof of completeness of the contract depicted in Figure 8.10. Fortunately, this work may be done automatically. This advantage makes it possible to apply the specification drivers approach to legacy implementations. Indeed, if there is a source code project with a number of classes in it, then it is possible to devise an additional class, write all the applicable specification drivers into it and submit the resulting class to the prover. Instead of showing how to derive complete contracts having a set of specification drivers from scratch, we show how to apply the approach to existing contracts.

The EiffelBase2 library seems to be a natural choice for the experiment. The library contains a complete implementation of stacks specified as mathematical sequences. The corresponding implementation class is `V_LINKED_STACK`. In order to perform the experiment, it is necessary to take the stacks specification drivers from Section 8.4 and modify them so that the name of the implementation class would be `V_LINKED_STACK` in-

```
extend_is_well_defined (s1, s2: V_LINKED_STACK [G]; x: G)
  require
    s1.is_wrapped
    s2.is_wrapped
    s1.observers.is_empty
    s2.observers.is_empty
    modify([s1, s2])
    s1.is_equal (s2)
  do
    s1.extend (x)
    s2.extend (x)
  ensure
    s1.is_equal (s2)
  end
```

Figure 8.11: Specification driver for verifying by AutoProof.

stead of `STACKS_IMPLEMENTATION`. The specification driver `axiom_a4` comes with a pitfall: the `V_LINKED_STACK` class does not introduce its own creation feature, but redefines the default creation feature defined for all classes. Hence, the **create Result**.new instruction is not applicable here; one should use **create Result** instead. After these modifications, the specification drivers should successfully compile and be ready for verification.

The initial verification attempt using AutoProof will result in numerous precondition violations. As Section 8.5 suggests, the verification methodology [Pol+14] behind AutoProof assumes writing additional non-stack related assertions. For example, the `extend_is_well_defined` specification driver can be verified by AutoProof only in the form depicted in Figure 8.11. The five assertions in the beginning of the **require** precondition clause seem to be worth explaining them briefly. The `s1.is_wrapped` assertion says that reference `s1` is assumed to be non-void and not participating in any call; the `s1.observers.is_empty` assertion says that the set of objects interested in the state of `s1` should be empty – it is a part of the precondition of feature `extend` of class `V_LINKED_STACK`; finally, the **modify**([s1, s2]) assertion is a frame specification: it says that the enclosing feature, `extend_is_well_defined` in this case, is going to modify objects referenced by `s1` and `s2` (square brackets `[]` denote set constants in Eiffel). The precondition needs the **modify** assertion because the `extend_is_well_defined` feature uses feature invocations with side effects, `extend` in this case, on references `s1` and `s2`. Although the verification failures caused by the absence of these assertions do not bear any relation to stacks, they uncover certain weaknesses in the verification methodology: namely, the defaults do not seem sufficiently reasonable. For example, a violation of the `s1.is_wrapped` assertion would detect a callback situation, and callbacks are not so common as to assume them by default. The `observers.is_empty` requirement makes extending stack objects applicable only in situations when no other objects depend on their states. The **modify** frame specification may be generated automatically based on the presence of invocations with side effects in the implementation body.

After complementing the specification drivers with all necessary assertions related to verification methodology and rerunning AutoProof, it uncovers some stack-related issues. This is visible from the fact that this time the verification errors come from the postconditions. Namely, AutoProof fails to prove correctness of all the verification drivers from classes EQUIVALENCE_SPECIFICATION_DRIVERS and WELL_DEFINEDNESS_SPECIFICATION_DRIVERS as well as verification driver axiom_a2 from the ADT_AXIOMS_SPECIFICATION_DRIVERS class. As all of these specification drivers involve implementation feature is_equal, the first guess is that V_LINKED_STACK does not redefine it. This guess appears to be right: the class defines its own custom feature for comparing run time objects, but does not redefine the standard comparison feature in terms of the new one. Giving this flaw's fix here would not bring much value to the discussion, so it seems better to move on. After redefining feature is_equal, AutoProof succeeds in proving classes ADT_AXIOMS_SPECIFICATION_DRIVERS and EQUIVALENCE_SPECIFICATION_DRIVERS completely, but still fails to prove specification driver new_is_well_defined from the WELL_DEFINEDNESS_SPECIFICATION_DRIVERS class. As this specification driver uses the is_empty implementation feature, it falls under suspicion. Apparently, its postcondition does not have a clause corresponding to the definition clause in its abstract model in Figure 8.10. After fixing this flaw, everything verifies successfully, including the V_LINKED_STACK implementation class.

8.7 Summary

The chapter makes the following main contributions:

- Presents the notion of specification driver for encoding ADT axioms, which are not possible to encode using traditional DbC techniques.
- Illustrates the process of axiomatizing abstract equivalence using the new approach.
- Introduces an exhaustive definition of contract completeness.
- Demonstrates how to apply completeness checks to legacy implementations.

The new approach allows adding, changing or removing ADT axioms at any given moment of the development process without necessarily modifying the implementation classes. Although specification drivers occupy separate classes completely disjoint from implementation classes, they are simultaneously expressed in terms of objects instantiated from the implementation classes. The result is a seamless integration of software axiomatization and implementation driven by automatic verification of functional correctness. Attempts to check specification drivers can uncover weak postconditions of implementation features. Once strengthened, these postconditions potentially yield firmer executable instructions.

In light of the presence of different kinds of specification drivers described in Section 8.4 it seems feasible to propose the following changes to the Eiffel Verification Environment tool set:

- Develop a template for fast creation of classes intended to keep specification drivers.

- Automate generation of specification drivers for equivalence and well-definedness.
- Revise verification methodology underlying AutoProof: specification drivers are a new syntactical specification construct, which may remove some verification challenges.

Work [Mey13] introduces the notion of multirequirements, and Chapter 7 illustrates how to apply this notion in practice. The underlying idea is that a separate item in a software requirements document should be expressed using several interwoven notations, e.g. natural language, graphical form and formal notation. For the formal notation, it was suggested to use a rather expressive programming language. The present chapter talks about expressing ADT axioms in a programming language with pre- and postconditions. Since ADT specifications are one of the languages for expressing software requirements, it makes sense to revisit the original multirequirements approach to see how the idea of specification drivers could improve it.

Chapter 9

Making Contracts Consistent

Existing techniques of Design by Contract do not prevent developers from specifying inconsistent contracts. Any attempt to write a program to meet an inconsistent contract will fail, leading to wasted resources. In the present chapter we describe a technique for catching inconsistent contracts in the development time. Applying the technique may save projects' resources and lower the likelihood of failure.

9.1 Introduction

In the world of program correctness, it takes two to tango: a specification and implementation. A program is correct if the implementation satisfies the specification. If they do not match, the program is incorrect. In general, work on program verification takes the specification for granted and blames any fault on the implementation. But it is possible to write a specification that no implementation can satisfy. Given the routine contract

```
require
  a > 0
ensure
  b > old a
  b <0
```

one cannot implement the routine, since it would have to yield a value of b that is both negative and greater than the positive initial value of a. Little work has addressed the issue of "wrong" specifications, perhaps because the general notion of "wrong" is difficult to define and assess: wrong with respect to what? Most likely to another, higher-level specification, but this is just an escalation of the problem. As the example suggests, however, a specific case of "wrong" does not raise this problem: a specification can be inconsistent, hence impossible to implement. Then we want to know right away; and even if we have written an implementation and the verification process – inevitably – cannot prove it correct, it should direct us to looking for the bug where it lies: in the specification. This chapter presents a technique to find out automatically that a specification is inconsistent.

Empirical studies of contracted programs reveal that the problem is not limited to artificial examples such as this one, but in fact arises widely in practice. Ciupa et al [Ciu+11] (also [Ciu+08], [Mey+07]), in their studies of bugs in contracted programs, found that an astounding 62.42% of contract violations during random testing of their program sample resulted from incorrect specifications (rather than incorrect implementations), although they do not state which ones are inconsistencies.

The technique presented here, enjoying automatic tool support thanks to the AutoProof verification environment [Tsc+15], is powerful enough to catch the following inconsistencies in classes with a contract:

- Inconsistency of the invariant, which results in impossibility to have instances of the class (Section 9.3).

- Inconsistency of a routine's postcondition, which invalidates the client's state after calling a routine (Section 9.4).

- Inconsistency of a routine's precondition, which makes calling the routine of the class impossible (Section 9.5).

The approach also handles some nuances related to non-exported routines, which may be called only in the non-qualified way (Section 9.6).

9.2 Why detect?

If a contract is inconsistent, this will eventually become apparent in any case. For example, if a class has an inconsistent invariant, it will not be possible to develop a correct creation procedure: all creation procedures must establish the class invariant on their completion [Mey92]. If a routine's precondition is inconsistent, no client of the class will be able to call the routine. If the precondition is satisfiable, but the postcondition is not, the outcome will be like the one for an inconsistent invariant: no one will be able to implement the routine correctly.

The biggest problem with this trial and error approach is the waste of resources: it may take multiple man-hours before the developer understands that the specification is not implementable at all. We describe an alternative approach, capable of catching inconsistencies before they turn into problems.

9.2.1 Example

To illustrate our approach, we use a class that describes an ordered triple of integers (Figure 9.1), in which the order is represented by the class invariant. From the description of the approach it will be visible that it scales to classes of unlimited complexity, which is why it does not seem bad to pick an artificial and simplified example. There are no inconsistencies in the INTEGER_TRIPLE class' contract, which consists only of the invariant yet. Throughout the chapter we extend the example, intentionally introduce various inconsistencies to it and show how to detect them.

All the experiments are reproducible in the Eiffel verification environment [Tsc+11].

```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
invariant
  a > b
  b > c
end
```

Figure 9.1: Example: an ordered triple of integers.

9.2.2 The basic idea

If there is an inherent inconsistency in either the signature, the precondition, or the execution part, it should be possible to prove the following specification driver (Chapter 8):

```
routine (ARGS)
  require
    pre(ARGS)
  do
    execution(ARGS)
  ensure
    False
  end
```

This equation basically encodes a proof by contradiction of a potential inconsistency: prove false, assuming the possibility to use `routine`, `pre` and `execution` together. If the assumption is a logical contradiction, the prover will accept the proof.

We successively examine how to apply this general form to express and prove inconsistency of invariants (Section 9.3), postconditions (Section 9.4) and preconditions (Section 9.5). The examples are written in Eiffel and checked with AutoProof.

9.3 Class Invariants

A class invariant is a property that applies to all instances of the class, transcending its routines [Mey92]. From this definition, an immediate conclusion follows: if the class invariant is inconsistent, then no objects can have it as their property. This conclusion leads us to the following definition.

Definition 9.3.1 *Class `TARGET_CLASS` has an inconsistent invariant, if, and only if, the following specification driver is provable:*

```
class_invariant (n: TARGET_CLASS)
  do
  ensure
    False
  end
```

The `class_invariant` routine represents a proof by contradiction, in which the proof assumption is that there can be an object of `TARGET_CLASS`. If its invariant is inconsistent,

```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
invariant -- The assertions below cannot hold together.
  a > b
  b > c
  c > a
end
```

Figure 9.2: Example of an inconsistent invariant.

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
feature
  class_invariant (n: INTEGER_TRIPLE)
    do
    ensure
      False
    end
end
```

Figure 9.3: Proof of the `INTEGER_TRIPLE` invariant inconsistency.

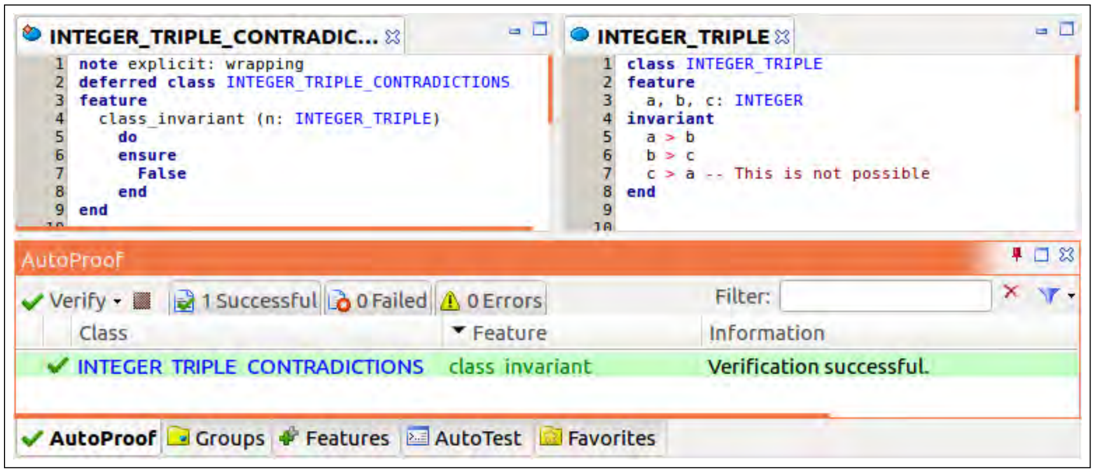
then existence of such an object is not possible in principle, and assuming the opposite should lead to a contradiction. If AutoProof accepts the `class_invariant` specification driver, then `TARGET_CLASS` has an inconsistent invariant.

Assume an artificial inconsistency in the invariant of the `INTEGER_TRIPLE` class (Figure 9.2). Following from the transitivity of the $>$ relation on integers, the last assertion is inconsistent with the first two. According to the Definition 9.3.1, it is necessary to encode the corresponding specification driver and submit it to AutoProof; but a specification driver must exist in some class, which is a minimal compilable program construct in Eiffel. Assume there is such a class, `INTEGER_TRIPLE_CONTRADICTIONS` (Figure 9.3). If the class compiles, the next step is to submit the proof to AutoProof[1]. AutoProof accepts the proof (Figure 9.4), from which we conclude the existence of an inconsistency in the invariant of the `INTEGER_TRIPLE` class. Removal of the problematic assertion from the invariant makes AutoProof reject the `class_invariant` proof (Figure 9.5). For the remaining examples, you can download AutoProof and check them locally.

9.4 Postconditions

According to the principles of Design by Contract [Mey92], a routine will never complete its execution, if it fails to assert its postcondition; consequently, to express the contradiction, the corresponding specification driver needs to assume the termination

[1]The **note explicit**: `wrapping` expression in the first line of the class is a verification annotation for AutoProof [Pol+14]; its meaning is not related to the ideas under the discussion.


INTEGER_TRIPLE_CONTRADIC...

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
feature
  class_invariant (n: INTEGER_TRIPLE)
    do
    ensure
      False
    end
end
```

INTEGER_TRIPLE

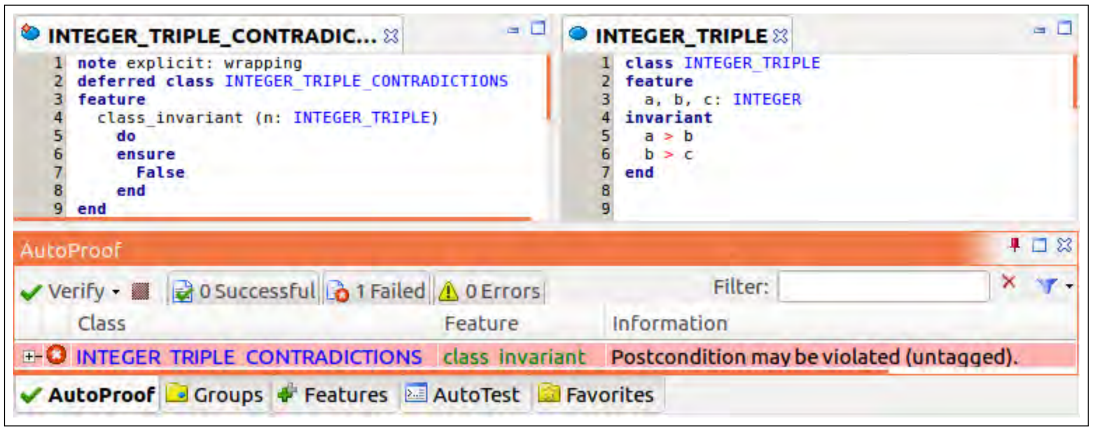
```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
invariant
  a > b
  b > c
  c > a -- This is not possible
end
```

AutoProof

Verify · 1 Successful · 0 Failed · 0 Errors · Filter:

Class	Feature	Information
✓ INTEGER_TRIPLE_CONTRADICTIONS	class_invariant	Verification successful.

Figure 9.4: Proving an inconsistency of the invariant.


INTEGER_TRIPLE_CONTRADIC...

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
feature
  class_invariant (n: INTEGER_TRIPLE)
    do
    ensure
      False
    end
end
```

INTEGER_TRIPLE

```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
invariant
  a > b
  b > c
end
```

AutoProof

Verify · 0 Successful · 1 Failed · 0 Errors · Filter:

Class	Feature	Information
INTEGER_TRIPLE_CONTRADICTIONS	class_invariant	Postcondition may be violated (untagged).

Figure 9.5: Failure to find an inconsistency in the invariant.

```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
  move_c
    do
    ensure -- The assertions below cannot hold together with the invariant
      a = old a
      b = old b
      c = 2 ∗ a − old c
    end
invariant
  a > b
  b > c
end
```

Figure 9.6: A command with an inconsistent postcondition.

and assert `False` in its postcondition. Two definitions follow for commands (Section 9.4.1) and functions (Section 9.4.2); the definitions differ according to the ways in which clients use commands and functions.

9.4.1 Commands

Commands are state-changing routines, which is why clients can use command calls only in routines' bodies, not in contracts. To prove the inconsistency of a command's postcondition, it is necessary to assume that it is possible to call the command and continue execution of the program.

Definition 9.4.1 *An exported command* `c` *with a precondition* `pre` *and a list of formal arguments* `ARGS` *from class* `TARGET_CLASS` *has an inconsistent postcondition, if, and only if, the following specification driver is provable:*

```
c_post (t: TARGET_CLASS; ARGS)
  require
    t.pre (ARGS)
  do
    t.c (ARGS)
  ensure
    False
  end
```

This is a proof by contradiction in which the assumption is the possibility to call the `c` command so that the execution reaches checking the postcondition of `c_post`. If the postcondition of `c` is inconsistent alone or is not consistent with the invariant of `TARGET_CLASS`, the execution will stop right after the call, and the outer postcondition will never be checked.

Assume the task is to implement command `move_c` that should somehow change the value of the `c` attribute in the `INTEGER_TRIPLE` class:

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
feature
  move_c_post (n: INTEGER_TRIPLE)
    require
      modify (n)
    do
      n.move_c
    ensure
      False
    end
end
```

Figure 9.7: Specification driver for detection of the `move_c` command's inconsistent postcondition.

The last line in the postcondition of the `move_c` command makes the value of `c` bigger than that of `a`, which is not consistent with the invariant.

The `move_c_post` specification driver (Figure 9.7) reflects a proof by contradiction of the inconsistency[2].

AutoProof accepts the `move_c_post` specification driver, from which one can see the presence of an inconsistency in the postcondition of `move_c`; removal of its last line will make AutoProof rejecting the proof.

9.4.2 Functions

Functions are state-preserving value-returning routines, which may be used in other routines' pre- and postconditions. To prove by contradiction inconsistency of a function's postcondition, it is necessary to assume that the function can produce some value.

Definition 9.4.2 *An exported function `f` with a return type `T`, precondition `pre`, and a list of formal arguments `ARGS` from class `TARGET_CLASS` has an inconsistent postcondition, if, and only if, the following specification driver is provable:*

```
f_post (t: TARGET_CLASS; ARGS; res: T)
  require
    t.f (ARGS) = res
  do
  ensure
    False
  end
```

If the postcondition of `f` is inconsistent alone, or is not consistent with the class invariant, it will never return any result. The **`require`** block in the Definition 9.4.2 states the opposite: there is some value `res` of type `T`, such that it equals the value of the function; this statement is the assumption of the proof by contradiction.

[2]The **`modify`**(n) expression inside the **`require`** block is a frame specification for AutoProof [Pol+14].

```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
  diff_ab: INTEGER
    do
    ensure -- The assertions below cannot hold together with the invariant
      Result = b − a
      Result > 0
    end
invariant
  a > b
  b > c
end
```

Figure 9.8: A function with an inconsistent postcondition.

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
feature
  diff_ab_post (n: INTEGER_TRIPLE; diff: INTEGER)
    require
      n.diff_ab = diff
    do
    ensure
      False
    end
end
```

Figure 9.9: Specification driver for detection of a function with an inconsistent postcondition.

Assume the task is to implement a function `diff_ab` that returns the difference `b − a` between `a` and `b`. From the invariant of `INTEGER_TRIPLE`, one can see that this difference should always be negative, but the developer may confuse operators $>$ and $<$, in which case the postcondition of `diff_ab` becomes inconsistent (Figure 9.8).

Specification driver `diff_ab_post` (Figure 9.9) reflects the proof by contradiction corresponding to the given example. AutoProof accepts `diff_ab_post`, thus disclosing the presence of an inconsistency.

9.5 Preconditions

Precondition of a routine constitutes requirements that every client has to meet to call the routine. If a precondition is inconsistent, no client will be able to meet it.

Definition 9.5.1 *An exported routine* `callable` *with precondition* `pre` *and list of formal arguments* `ARGS` *from class* `TARGET_CLASS` *has an inconsistent precondition, if, and only if, the following specification driver is provable:*

```
class INTEGER_TRIPLE
feature
  a, b, c: INTEGER
  move_c
    require
      diff_ab > 0 -- Cannot hold together with the invariant
    do
    ensure
      a = old a
      b = old b
    end
```

```
  diff_ab: INTEGER
    do
    ensure
      Result = b − a
    end
invariant
  a > b
  b > c
end
```

Figure 9.10: The `move_c` command with an inconsistent precondition.

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
feature
  move_c_pre (n: INTEGER_TRIPLE)
    require
      n.diff_ab > 0
    do
    ensure
      False
    end
end
```

Figure 9.11: Specification driver for catching the inconsistent precondition.

```
callable_pre (t: TARGET_CLASS; ARGS)
  require
    t.pre (ARGS)
  do
  ensure
    False
  end
```

Assume the `move_c` command requires the result of the `diff_ab` function to be greater than `0`, which is not consistent with the class invariant, according to the postcondition of `diff_ab` (Figure 9.10).

The `move_c_pre` specification driver reflects the Definition 9.5.1 as applied to the precondition of the `move_c` command. It has the same precondition as does the `move_c` command, where every non-qualified call is replaced with its qualified counterpart; the target for the call comes from the `move_c_pre`'s list of formal arguments.

Note that the `move_c_post` (Figure 9.7) specification driver needs to be updated: the `move_c` command now has a precondition that has to be guaranteed by all its callers.

AutoProof discloses the presence of a contradiction by accepting the `move_c_pre` specification driver.

9.6 Non-exported routines

A non-exported routine is a routine that cannot be invoked using a qualified call [Mey09]. Consequently, the definitions, which were presented so far, are not applicable to non-exported routines: those definitions rely on the ability to do qualified calls. The present section gives definitions applicable to non-exported routines.

Definition 9.6.1 *The non-exported command* `c` *with precondition* `pre` *and list of formal arguments* `ARGS` *has an inconsistent postcondition, if, and only if, the following specification driver is provable:*

```
c_post (ARGS)
  require
    pre (ARGS)
  do
    c (ARGS)
  ensure
    False
  end
```

Definition 9.6.2 *The non-exported function* `f` *with return type* `T`*, precondition* `pre`*, and list of formal arguments* `ARGS`*, has an inconsistent postcondition, if, and only if, the following specification driver is correct:*

```
f_post (ARGS; res: T)
  require
    f (ARGS) = res
  do
  ensure
    False
  end
```

Definition 9.6.3 *The non-exported routine* `r` *with precondition* `pre` *and list of formal arguments* `ARGS` *has an inconsistent precondition, if, and only if, the following specification driver is correct:*

```
r_pre (ARGS)
  require
    pre (ARGS)
  do ensure False end
```

In Definition 9.6.1, Definition 9.6.2, and Definition 9.6.3 the routine calls do not have targets, which means the calls can occur only in the class where the routines are defined or in one of its descendants. [Mey09].

Assume the `INTEGER_TRIPLE` class with all its routines non-exported (Figure 9.12), which is denoted by the `{NONE}` specifier next to the **`feature`** keyword. For such an example, the specification drivers class may be a descendant of the `INTEGER_TRIPLE` class so that it will be able to call its routines in the unqualified way (Figure 9.13).

```
class INTEGER_TRIPLE
feature {NONE}
  a, b, c: INTEGER
  move_c
    require
      diff_ab > 0
    do
    end
  diff_ab: INTEGER
    do
    end
end
```

Figure 9.12: The INTEGER_TRIPLE class with all the features non-exported.

```
note explicit: wrapping
deferred class INTEGER_TRIPLE_CONTRADICTIONS
inherit INTEGER_TRIPLE
feature {NONE}
  move_c_post
    require
      diff_ab > 0
    do
      move_c
    ensure
      False
    end
  diff_ab_post (res: INTEGER)
    require
      diff_ab = res
    do
    ensure
      False
    end
  move_c_pre
    require
      diff_ab > 0
    do
    ensure
      False
    end
end
```

Figure 9.13: Specification drivers for detection of contradictions in the non-exported routines.

9.7 Related Work

The problem of inconsistent specifications receives noteworthy attention in Z ([ASM80]). Without an explicit syntactical separation of Z assertions into pre- and postconditions and in the absence of an imperative layer, it is not clear how to apply the techniques from the present chapter. Detection of inconsistencies in Z may occasionally lead to development of complicated theories and tools [MDB02]. We are not aware of any work specifically targeting detection of inconsistencies in Design by Contract.

The problem of inconsistent contracts may also be viewed through the prism of liveness properties in concurrency [MK06]:

- An inconsistent class invariant makes the class “non-alive”: it is not even possible to instantiate an object from the class.
- An inconsistent routine precondition makes the routine never callable.
- An inconsistent routine postcondition leads to its clients’ always crashing after calling the routine.

9.8 Summary

A strength of the approach is the possibility to employ it for real-time detection of inconsistencies. Once generated, the specification driver for the invariant (Section 9.3) never changes; consequently, it is enough to recheck it whenever the invariant changes and display a warning in the event of successful checking. The same applies to detection of inconsistent pre-/postconditions, with the only difference that it will be necessary to update the preconditions of the corresponding specification drivers in the event of modifying the routine’s precondition. In any case, such an update amounts to copying the precondition with possibly adding targets in front of the class’ queries (Section 9.5).

Another strength of the approach is its applicability. Eve is not the only environment in which it is possible to write and statically check contracts: there is a similar environment for .net developers [Bar10], in which the techniques presented here are applicable. There are several programming languages that natively support contracts, for which the presented approach is applicable conceptually, but still needs development of a verifier capable of checking specification drivers.

9.8.1 Limitations of the approach

Results interpretation

In the presented approach, a positive response from the prover means something bad, which is detection of an inconsistency. This may be misleading: the developer may think, instead, that everything is correct. This requires fixing, possibly by development of a separate working mode in AutoProof.

Precision

The approach shows the presence of a contradiction but does not show its location. This is not a problem when developing from scratch: background verification may catch the contradiction as soon as it is introduced. However, if the task is to check an existing codebase, the only way to locate origins of contradictions seem to be in commenting/uncommenting specific lines of the contracts.

Frozen classes

The approach for non-exported routines relies on the ability to inherit from the supplier class. It is not possible to inherit from a class, if it is declared with the `frozen` specifier [Mey09]. Nevertheless, it is always possible to apply the technique to exported routines of the supplier class.

9.8.2 Future work

The present chapter describes the approach conceptually, yet no tools exist that could generate and check the necessary proofs automatically. Two main possibilities exist in this area:

- Build a contradiction detection functionality into AutoProof, without letting developers see the proofs.
- Develop a preprocessing engine on the level of Eiffel code that would generate classes with proofs for checking them with AutoProof in its current state.

Apart from automating the approach, it seems reasonable to investigate whether the proof by contradiction technique may be of any help with other problems of program verification.

Chapter 10

Seamless Requirements

Popular notations for functional requirements specifications often ignore developers' needs, target specific development models, or require translation of requirements into tests for verification; the results can give out-of-sync or downright incompatible artifacts. Seamless Requirements, a new approach to specifying functional requirements, contributes to developers' understanding of requirements and to software quality regardless of the process, while the process itself becomes lighter due to the absence of tests in the presence of formal verification. A development case illustrates these benefits, and a discussion compares seamless requirements to other approaches.

10.1 Introduction

Seamless Requirements is a technique to close the various gaps that have long plagued the practice of software requirements:

- The gap between customers and developers (Section 10.1.1).
- The gap between agile and formal development (Section 10.1.2).
- The gap between construction and verification (Section 10.1.3).

To reach this goal, seamless requirements build on ideas coming from diverse sources, including literate programming [Knu84], multirequirements [Mey13], and formal verification [Tsc+15]. A seamless requirement combines two elements: a contracted self-contained routine, which doubles as a proof obligation, and an associated natural language comment.

The approach assumes object-oriented non-concurrent setting and does not handle non-functional requirements.

10.1.1 Customers vs. developers

By adding programming languages with contracts to the family of requirements specification notations, seamless requirements improve developers' understanding of requirements that typically exist in some declarative form that has nothing to do with programming.

The modern taxonomy of requirements specification languages ([Lam09, Chapter 4 "Requirements Specification and Documentation"]) provides a number of formal and semi-formal notations, and programming languages are not a part of this taxonomy. This implicitly isolates people (customers) who state requirements from people (developers) who implement them. As soon as the customers elicit and document requirements, demonstrate some "good" properties of the requirements within the chosen notation, the developers will have to map the notation into the semantics of the target programming language. Is there any way to check the translation at the same level of rigor used to derive those "good" properties of the requirements? Some approaches advocate modeling software at different angles using different notations to ensure its proper understanding by developers, but such an approach raises the problem of potential inconsistency between the views.

Seamless requirements express software functionality using the language best understood by developers: the programming language. The idea is not new [Mey13], but its implementation is (Section 10.5.6 gives more details). A seamless requirement is a compilable contracted self-contained routine – specification driver (Section 8.3) – equipped with a structured natural language comment. The comment delivers the meaning of the requirement to the customers, and the program construct – to the developers. Specification drivers are expressive enough to fully capture algebraic specifications (Section 8.4), and exercising their expressiveness is a driving force of the present research.

The idea of combining formal and natural language descriptions is present in goal-oriented requirements engineering [Lam01], but the approach does not consider a programming language as a formal notation.

10.1.2 Agile vs. formal development

By nature both self-contained and formal, seamless requirements boost reliability of software produced using agile processes.

Compatibility of agile development and formal methods has long been a concern for software engineers ([TFR14], [Bla+09]), including those developing mission and life-critical software ([DNR04], [SA07]). The studies have something in common: their main concern is integration of agile practices into development of software that has to be reliable and is currently developed using some conservative process. In the same time there is a lack of research that studies applicability of formal methods to agile development of not so critical mass-market software for increasing its reliability. This problem is among the concerns of the seamless requirements approach.

In agile development a functional requirement typically takes a form of a scenario describing user interaction with the to-be software. The scenario is then translated into a set of unit tests for ensuring functional correctness of the software with respect to the scenario. Scenarios and unit tests naturally fit the agile philosophy of frequently delivering software in small increments: they both are self-contained information units suitable for grouping into arbitrary sized sets. It is the very nature of tests that limits the level of formality in agile development: they exercise only a subset of the possible execution paths. Although there are scientific approaches for making a test suit cover the software well enough, agile methods do not consider tests as a very important artifact and do not advocate improving tests coverage too much.

Seamless requirements replace unit tests with specification drivers, testing with formal verification, and move structured natural language scenarios to comments on specification drivers. Specification drivers can capture scenarios in their abstract form (as opposed to unit tests), which is why it makes sense to conjoin them. The resulting requirement form keeps the fine granularity of tests and scenarios, while being mathematically formal.

10.1.3 Construction vs. verification

Seamless requirements enable straightforward verification of existing software with respect to requirements without introduction of intermediate artifacts such as tests.

The modern software mass market rests on testing as the primary mechanism for checking functional correctness. Although tests are fundamentally imprecise (Section 10.1.2), there are scientific approaches to testing that enable production of test suits having reasonable code coverage with respect to some predefined criteria [Jor08]. Such an approach may be suitable for greenfield software construction, but not always for verification of existing software that already works somehow. The problem is real: software quality cannot be higher than that of its least quality component. This means that, in order to reuse a third-party component, the development team has to make sure that its quality conforms to the quality standards defined in the project through generating and running sufficient number of tests on the component. It is not surprising that such an effort is often considered as waste: why test something that is already on the market and works instead of putting more effort into construction?

Seamless requirements fix the issue by replacing testing with formal verification of specification drivers, which are formal and abstract representations of software usage scenarios. The only assumption upon which the approach rests is existence of a contract in the component, which is dictated by modularity of the verification approach[Tsc+15].

10.2 Motivating example

An example illustrates the idea of seamless requirements. The task is to implement a clock class that features seconds, minutes, hours, and days of week. The clock state should be updated through a special command, `tick`, that advances the seconds counter. There is also an existing class `CLOCK` that does not feature the current day of week. The class is implemented and specified in Eiffel [Mey88]. The implementation is closed: only a specification in the form of a contract is available. Figure 10.1 contains the visible part of the class. The **`frozen`** specifier prohibits inheriting from the `CLOCK` class and thus makes it usable only for instantiating and using its instances. It is also known that the hidden implementation of the `tick` command is provably correct with respect to its postcondition in Figure 10.1, and the correctness was established with the AutoProof verifier [Tsc+15] for Eiffel programs with contracts.

```
frozen class CLOCK
feature
  second, minute, hour: INTEGER

  tick
    do
      -- Hidden implementation
    ensure
      old second ≤ 58 implies ((second − old second = 1) and minute = old minute)
      old second > 58 implies
        (second = 0 and (old minute ≤ 58 implies minute − 1 = old minute) and
          (old minute > 58 implies
            (minute = 0 and (old hour ≤ 22 implies hour − old hour = 1)
              and (old hour > 22 implies hour = 0))))
    end
end
```

Figure 10.1: The existing clock class.

10.2.1 Existing code

This section takes a closer look at the visible parts of the `CLOCK` class in Figure 10.1. The space between the **`do`** and **`ensure`** keywords of the `tick` feature would typically contain executable instructions, which are hidden in this case. Logical assertions between the **`ensure`** and the closest **`end`** keyword constitute the postcondition of the feature. The postcondition logically connects the clock pre-state, which precedes any invocation of `tick`, with the post-state, which results from the invocation. The **`old`** keyword before some identifiers denotes values of the respective queries in pre-states. Accordingly, identifiers that go without the **`old`** keyword denote values of the respective queries in post-states.

Since it is not possible to modify the `CLOCK` class, it seems reasonable in this context to implement the required extended class through the composition relation: in the new class declare a reference to an object of type `CLOCK` and reuse its functionality. In order to do so it is necessary to make sure that objects of the existing class, indeed, behave like a real clock. The extended clock development plan thus consists of the following major steps:

1. Identifying requirements to an extended clock.
2. Identifying requirements that are applicable to a non-extended clock.
3. Verifying the existing `CLOCK` class with respect to the requirements for a non-extended clock.
4. Reusing the existing class in the event of its successful verification.
5. Developing a completely new class otherwise.

A clock tick:

(REQ1) Increments current second if it is smaller than 59.

(REQ2) Resets current second to 0 if it equals 59.

(REQ3) Increments current minute if the time is HH:MM:59 for MM smaller than 59.

(REQ4) Resets current minute to 0 if it equals 59 and current second equals 59.

(REQ5) Keeps current minute if current second is smaller than 59.

(REQ6) Increments current hour if the time is HH:59:59 for HH smaller than 23.

(REQ7) Resets current hour to 0 if the time is 23:59:59.

(REQ8) Keeps current hour if current second is smaller than 59.

(REQ9) Increments current day at 23:59:59 if it is not Sunday.

(REQ10) Resets current day to Monday after a clock tick at 23:59:59 on Sunday.

(REQ11) Keeps current day if current second is smaller than 59.

Figure 10.2: Natural-language requirements to clock.

10.2.2 Natural-language requirements

Implementation of the first two steps of the plan starts with enumeration of the requirements in their natural language form in Figure 10.2. Requirements (REQ1)-(REQ8) do not talk about the current day of week and thus are applicable to the existing implementation. Requirements (REQ9)-(REQ11) talk about the days counter and thus are applicable only to the extended implementation. For simplicity, days are represented with numbers from 0 to 6, where 0 corresponds to Monday, and 6 – to Sunday.

In many cases natural language requirements are less clear and precise than the ones in Figure 10.2. This particular issue is irrelevant to the present discussion, which is why the example relies on the assumption that the natural language requirements in the clock example are of high enough quality.

Step 3 of the plan from Section 10.2.1 is to check whether the `CLOCK` class meets requirements (REQ1)-(REQ8). This step, along with steps 4 and 5, is far less trivial than steps 1 and 2 and raises a number of questions.

10.2.3 Research questions

RQ1

How to express precise semantics of the natural language scenarios (REQ1)-(REQ8) using programming language constructs?

Natural-language statements in Figure 10.2 are comfortable for reading by human beings. This may be not enough, however, for those who will potentially implement the requirements. Natural language is a source of misinterpretations and ambiguities, which is why it is not enough to have requirements in this form [Mey85]. What do statements (REQ1)-(REQ8) mean exactly in terms of the programming language abstractions? It would benefit the software developers to be able to precisely express the requirements in the programming language that will later be used for their implementation.

The question does not assume replacement of natural language requirements with their programmatic counterparts: the goal is to have a representation which would encompass both views with the possibility of extracting only one of them.

RQ2

How to make each requirement both self-contained and formal?

Requirements (REQ1)-(REQ11) are already self-contained and thus are suitable for agile development of arbitrary sized increments. How to enrich them with formality without sacrificing their granularity?

RQ3

How to understand whether the partially available implementation in Figure 10.1 meets requirements (REQ1)-(REQ8)?

It is possible to take requirements (REQ1)-(REQ11) and mentally convert them to a correct implementation, but the task assumes reuse of the existing class `CLOCK` in case of its correctness. How can one prove automatically that it meets requirements (REQ1)-(REQ8)? The only available part of the `CLOCK` class is its contract – the postcondition of command `tick`. It is also known that the hidden implementation of `tick` provably meets its postcondition. The question then reduces to the following one: how can one understand if the postcondition of `tick` meets requirements (REQ1)-(REQ8)?

10.3 Seamless requirements

Figure 10.3 contains the (REQ1) requirement in the form of a seamless requirement – a contracted routine with a natural language comment[1]. The comment contains the natural language representation of (REQ1) in Figure 10.2. The routine part, together with the signature and the contract parts, constitutes a proof obligation:
"for any object `clock` of type `CLOCK` and any value `current_second` of type `INTEGER`, such that `clock.second` <59 and `clock.second = current_second`, an execution of `clock.tick` results in `clock.second = current_second + 1`". The **`modify`** `(clock)` clause in the precondition limits side effects of the `tick` routine: the routine is allowed to modify only the target object `clock` plus any object owned by `clock` [Pol+14]. It is possible to submit such a proof obligation to an automatic prover. AutoProof verifier fulfills this role for Eiffel programming language used in this example.

[1]Comments start with a double hyphen `--` in Eiffel

```
req_1 (clock: CLOCK; current_second: INTEGER)
-- A clock tick increments current
-- second if it is smaller than 59.
  require
    modify (clock)
    clock.second < 59
    clock.second = current_second
  do
    clock.tick
  ensure
    clock.second = current_second + 1
  end
```

Figure 10.3: Requirement (REQ1) in the seamless form.

The idea to use auxiliary routines with pre- and postconditions for complete specification of programs was proposed in Chapter 8. The routines are assumed to be expressed only in terms of their formal arguments. That work introduces a new term "specification drivers" to denote such routines and shows that they are expressive enough to fully capture functional semantics of classes. Since specification drivers are, syntactically speaking, routines, it is possible to comment on them with natural language statements – the ability to comment on routines is natural for any modern programming language. A seamless requirement consists of two important parts:

- Specification driver that captures the formal semantics for the requirement.
- Natural-language comment on the specification driver that informally captures the semantics.

A specification driver is a contracted routine expressed only in terms of its formal arguments and is understandable to AutoProof as a proof obligation.

The structure of a seamless requirement, together with the properties of specification drivers, answers the questions from Section 10.2.3 and ensures the core properties of seamless requirements, as the following sections illustrate.

10.3.1 RQ1: understandability to developers

Seamless requirements are contracted routines, which are programming language constructs understandable to programmers. Natural-language comments on these routines capture the informal representation of requirements that is understandable to customers. This duality makes a seamless requirement understandable to the two principal groups of stakeholders and semantically connects natural language requirements to the CLOCK class, thus answering the RQ1 question from Section 10.2.3.

The idea of interweaving natural language prose with programming language constructs was first proposed by Knuth in [Knu84]. One of the underlying theses of the seamless requirements approach is that it makes sense to use the standard commenting mechanism of the underlying programming language for this purpose.

10.3.2 RQ2: introducing formality into agile development

As their specification driver components are mathematically precise, seamless requirements do not accumulate ambiguity. Specification drivers are expressed completely in terms of their formal arguments, which is why they are also self-contained. The combination of the two properties benefits agile development with formality and does not interfere with its incrementality.

10.3.3 RQ3: utility for development activities

A seamless requirement is a natural language statement and, at the same time, is a proof obligation. Consequently, to prove correctness of an implementation with respect to a natural language requirement is to extend this requirement to the seamless form and then try to prove its proof obligation part. The approach also contributes to the following development activities.

Requirements documentation

A requirements document becomes an auxiliary class in the same namespace with the implementation classes. Since seamless requirements are self-contained routines, there is no place for a naming conflict in the event of putting together multiple seamless requirements within a single class. Section 10.4.1 illustrates this concept on the clock example.

Specification validation

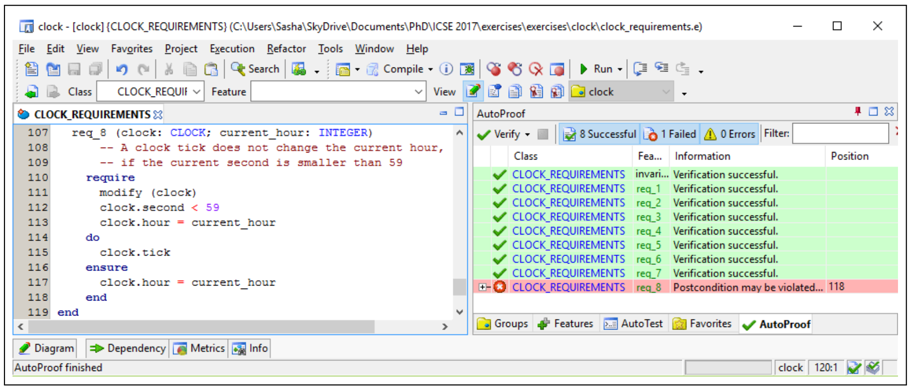
Seamless requirements, being proof-obligations understandable to AutoProof, introduce the notion of proving a requirement. Verification by AutoProof is modular: for example, for proving the `req_1` requirement in Figure 10.3 AutoProof will use only the postcondition of the `tick` command. The modularity means that it is possible to verify a program with a hidden implementation with respect to a seamless requirement, when only a contract of the program is available. Section 10.4.2 illustrates the validation process for the existing `CLOCK` class.

Specification inference

It is possible to use seamless requirements for proof-driven development of programs from scratch. The automatic prover drives the process in this case. To infer a specification from a set of seamless requirements is to equip the implementation classes with contracts strong enough to prove the requirements. Once the requirements pass verification by AutoProof, the development process switches to the implementation phase. To infer an implementation from a specification is to implement all the implementation classes correctly with respect to their contracts [Mey97]. The correctness is proved with the same verifier.

```
note explicit: wrapping -- For AutoProof.
deferred class CLOCK_REQUIREMENTS
feature
-- A clock tick:
  req_1 (clock: CLOCK; current_second: INTEGER)
  -- increments current second if it is
  -- smaller than 59.
    require
      modify (clock)
      clock.second < 59
      clock.second = current_second
    do
      clock.tick
    ensure
      clock.second = current_second + 1
    end
  req_2 (clock: CLOCK)
  -- resets current second to 0 if it
  -- equals 59.
    require
      modify (clock)
      clock.second = 59
    do
      clock.tick
    ensure
      clock.second = 0
    end
  req_3 (clock: CLOCK; current_minute: INTEGER)
  -- increments current minute if the time
  -- is HH:MM:59 for MM smaller than 59
    require
      modify (clock)
      clock.second = 59
      clock.minute < 59
      clock.minute = current_minute
    do
      clock.tick
    ensure
      clock.minute = current_minute + 1
    end
  req_4 (clock: CLOCK)
  -- resets current minute to 0 if it equals
  -- 59 and the current second equals 59.
    require
      modify (clock)
      clock.second = 59
      clock.minute = 59
    do
      clock.tick
    ensure
      clock.minute = 0
    end
  req_5 (clock: CLOCK; current_minute: INTEGER)
  -- keeps current minute if current
  -- second is smaller than 59.
    require
      modify (clock)
      clock.second < 59
      clock.minute = current_minute
    do
      clock.tick
    ensure
      clock.minute = current_minute
    end
  req_6 (clock: CLOCK; current_hour: INTEGER)
  -- increments current hour if the time
  -- is HH:59:59 for HH smaller than 23.
    require
      modify (clock)
      clock.second = 59
      clock.minute = 59
      clock.hour < 23
      clock.hour = current_hour
    do
      clock.tick
    ensure
      clock.hour = current_hour + 1
    end
  req_7 (clock: CLOCK)
  -- resets current hour to 0 if the time
  -- is 23:59:59
    require
      modify (clock)
      clock.second = 59
      clock.minute = 59
      clock.hour = 23
    do
      clock.tick
    ensure
      clock.hour = 0
    end
  req_8 (clock: CLOCK; current_hour: INTEGER)
  -- keeps current hour if current second
  -- is smaller than 59.
    require
      modify (clock)
      clock.second < 59
      clock.hour = current_hour
    do
      clock.tick
    ensure
      clock.hour = current_hour
    end
end
```

Figure 10.4: The seamless requirements document corresponding to (REQ1)-(REQ8).



clock - [clock] {CLOCK_REQUIREMENTS} (C:\Users\Sasha\SkyDrive\Documents\PhD\ICSE 2017\exercises\exercises\clock\clock_requirements.e)

CLOCK_REQUIREMENTS

```
107   req_8 (clock: CLOCK; current_hour: INTEGER)
108       -- A clock tick does not change the current hour,
109       -- if the current second is smaller than 59
110     require
111       modify (clock)
112       clock.second < 59
113       clock.hour = current_hour
114     do
115       clock.tick
116     ensure
117       clock.hour = current_hour
118     end
119 end
```

AutoProof

	Class	Fea...	Information	Position
✓	CLOCK_REQUIREMENTS	invari...	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_1	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_2	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_3	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_4	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_5	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_6	Verification successful.	
✓	CLOCK_REQUIREMENTS	req_7	Verification successful.	
✗	CLOCK_REQUIREMENTS	req_8	Postcondition may be violated...	118

AutoProof finished

Figure 10.5: Eiffel Verification Environment with the AutoProof pane.

10.4 Seamless requirements in practice

Section 10.4.1 and Section 10.4.2 implement step 3 of the development plan from Section 10.2.1 by verification of the existing class in Figure 10.1 with respect to requirements (REQ1)-(REQ8). Section 10.4.3 and Section 10.4.4 use the verification results as input. The resulting artifacts are publicly available on GitHub [Naub].

10.4.1 Requirements documentation

The first step is to document requirements (REQ1)-(REQ8) in the seamless form. Figure 10.4 contains the respective requirements class[2]. The **`deferred`** keyword means that the class is not implemented: it is not possible to instantiate any objects from it.

Since a seamless requirements document is a class, such techniques as inheritance are applicable to it. For example, if a new set of requirements arrives, it is not necessary to add them to the `CLOCK_REQUIREMENTS` class. It is possible to create a subclass where only new requirements are enumerated, and the old ones will be inherited automatically. Section 10.4.4 illustrates this approach on the clock example.

The `CLOCK_REQUIREMENTS` class is provable by AutoProof: to prove it is to prove each of the seamless requirements it contains. Section 10.4.2 describes the meaning of this process.

10.4.2 Specification validation

To prove correctness of the `CLOCK` class with respect to requirements (REQ1)-(REQ8) is to execute AutoProof on the `CLOCK_REQUIREMENTS` class. Verification by AutoProof is modular: verification of a requirements class does not need access to the implementation classes' internals, only to their contracts. AutoProof assumes that these implementations meet their respective contracts.

[2]The "**`note explicit`**: `wrapping`" expression at the top of the class is a verification annotation [Pol+14] not related to the example.

```
class CLOCK
feature
  second, minute, hour: INTEGER

  tick
    do
      -- To implement
    ensure
      -- To specify
    end
end
```

Figure 10.6: Blank clock implementation.

Figure 10.5 contains a screenshot of Eiffel verification environment (Eve) [Tsc+11] with an AutoProof pane on the right side. The AutoProof pane contains the results of verifying the `CLOCK_REQUIREMENTS` class. Apparently, the postcondition of the `tick` feature in Figure 10.1 is insufficiently strong to meet the (REQ8) requirement. Although the hidden implementation of the `CLOCK` class is known to meet its contract, it is possible for the implementation not to meet the requirements. Double-clicking the red line in the AutoProof pane retargets Eve to the `req_8` routine, which represents the seamless form of (REQ8).

Since the specification of `CLOCK` failed validation with respect to requirements, step 5 of the development plan from Section 10.2.1 becomes active. This step consists of developing a completely new `CLOCK` class. Section 10.4.3 describes development of the regular clock functionality (REQ1)-(REQ8), and Section 10.4.4 incrementally extends it with the days counter functionality (REQ9)-(REQ11). The starting point is a blank class `CLOCK` in Figure 10.6, which does not have any contract or executable instructions. It only declares the clock features so that the requirements class in Figure 10.4 could compile.

10.4.3 Increment 0: the basic functionality

This section describes development of the basic clock functionality increment. The development occurs as follows: once all requirements for the increment are collected, software specification is inferred from them; then, an implementation is inferred to meet the specification. The present section illustrates how application of seamless requirements may facilitate the transitions between the adjacent phases with the help of AutoProof. Section 10.4.3 describes inference of a correct `CLOCK` specification based on the seamless requirements from the `CLOCK_REQUIREMENTS` class. Section 10.4.3 infers an implementation of the `CLOCK` class that meets the inferred specification.

Specification inference

Figure 10.7 depicts a postcondition of the `tick` feature, which meets the
`CLOCK_REQUIREMENTS` class, so that the latter passes verification by AutoProof. How can

```
tick
  do
    -- To implement
  ensure
    old second < 59 implies second = old second + 1
    old second = 59 implies second = 0
    old second = 59 and old minute < 59 implies minute = old minute + 1
    old second = 59 and old minute = 59 implies minute = 0
    old second < 59 implies minute = old minute
    old second = 59 and old minute = 59 and old hour < 23 implies hour = old hour + 1
    old second = 59 and old minute = 59 and old hour = 23 implies hour = 0
    old second < 59 implies hour = old hour
  end
```

Figure 10.7: Postcondition of `tick` that meets `req_1`-`req_8`.

one infer postconditions from seamless requirements? This problem does not seem to be solvable in the general case; however, the seamless requirements from the `CLOCK_REQUIREMENTS` class possess some common properties:

- Each of them involves only one feature call.
- Each of them involves only one object of type `CLOCK`.
- The `tick` feature does not have formal parameters.

These observations enable application of the following inference logic. The resulting assertion takes the form of a logical implication. If a seamless requirement involves some object `o: TYPE`, then for every expression of the form `o.q`, where `q` is a query of class `TYPE`, the following rules apply:

- If `o.q` occurs in the precondition of the requirement, it translates to `old q` in the antecedent of the implication.
- If `o.q` occurs in the postcondition of the requirement, it translates to `q` in the consequent of the implication.

A requirement may also use an auxiliary formal argument `a: SUPPLEMENTARY_TYPE`, such as `current_hour: INTEGER` in `req_6`. Assume that the following conditions hold together:

- The precondition of the requirement contains an expression of the form `o.p = a`.
- The postcondition of the requirement contains an expression of the form `o.q = f(a)`.

In this case these conditions translate to `q = f(old p)` in the consequent of the resulting implication.

Each assertion from the postcondition in Figure 10.7 is the result of an application of these inference rules to the respective seamless requirement.

```
tick
  do
    if second <59 then second := second + 1
    else second := 0
      if minute <59 then minute := minute + 1
      else minute := 0
        if hour <23 then hour := hour + 1
        else hour := 0
        end
      end
    end
  ensure
    -- Postcondition assertions
  end
```

Figure 10.8: Implementation of `tick` that meets `req_1`-`req_8`.

Implementation inference

Once there is a contract that meets the requirements class, and the latter passes verification by AutoProof, it makes sense to proceed to inference of an implementation that meets the inferred contract. Figure 10.8 contains an implementation of the `tick` feature, which is correct with respect to the postcondition in Figure 10.7. As in the case of specification inference from requirements, the correctness may be established by an application of AutoProof, but this time it should be executed on the `CLOCK` class, which implements the required functionality. The details of the inference process are omitted because they are studied very well [Mey09] and are irrelevant to the central idea of behind seamless requirements.

10.4.4 Added functionality

The regular clock functionality was implemented in Section 10.4.3 as one increment. The present section extends the basic functionality in smaller increments consisting of one requirement each.

There are three requirements in Section 10.2 that describe the desirable behavior of the clock with a day counter: (REQ9), (REQ10), and (REQ11). Figure 10.9 shows them as a part of a requirements class `EXTENDED_CLOCK_REQUIREMENTS`. This class is inherited from the original `CLOCK_REQUIREMENTS` class, together with all the existing seamless requirements, to which it adds its own. In the present section, each of the newly added requirements corresponds to a separate increment.

Compilation of the new requirements class fails: seamless requirements `req_9`-`req_11` use feature `day`, which is not a part of the `CLOCK` class yet. To fix the compilation error is to add the respective attribute to the existing list of clock attributes:

```
second, minute, hour, day: INTEGER
```

Now that the new requirements class compiles, it is possible to proceed to the first increment.

```
note explicit:wrapping
deferred class EXTENDED_CLOCK_REQUIREMENTS
-- The present class contains requirements
-- for a clock equipped with a days counter.
inherit CLOCK_REQUIREMENTS
feature
-- A clock tick:
  req_9 (clock: CLOCK; current_day: INTEGER)
  -- increments current day at 23:59:59,
  -- if it is not Sunday.
    require
      modify (clock)
      clock.second = 59
      clock.minute = 59
      clock.hour = 23
      clock.day < 6
      clock.day = current_day
    do
      clock.tick
    ensure
      clock.day = current_day + 1
    end
req_10 (clock: CLOCK)
-- resets current day to Monday after
-- a clock tick at 23:59:59 on Sunday.
  require
    modify (clock)
    clock.second = 59
    clock.minute = 59
    clock.hour = 23
    clock.day = 6
  do
    clock.tick
  ensure
    clock.day = 0
  end
req_11 (clock: CLOCK; current_day: INTEGER)
-- keeps current day if current
-- second is smaller than 59.
  require
    modify (clock)
    clock.second < 59
    clock.day = current_day
  do
    clock.tick
  ensure
    clock.day = current_day
  end
end
```

Figure 10.9: The seamless requirements document for extended clock.

Increment 1

Implementation of the first increment starts with submitting the `EXTENDED_CLOCK_REQUIREMENTS` class to formal verification by AutoProof. The new seamless requirements `req_9`, `req_10` and `req_11` fail the verification attempt: the postcondition of the `tick` command does not say anything about the `day` attribute, which has just been added to the implementation class. We choose to implement the `req_9` requirement in the first increment.

According to the inference rules from Section 10.4.3, it should suffice to strengthen the postcondition of `tick` with the following assertion:

```
old second = 59 and old minute = 59 and old hour = 23 and old day <6 implies day = old day + 1
```

Now that `req_9` passes verification, it is necessary to verify the `CLOCK` class. The verification attempt fails because the implementation of `tick` has not been updated yet to meet the new assertion in the postcondition.

The following **if** block meets the new assertion, which may be confirmed with AutoProof:

```
tick
  do
    -- Other lines of code
    else hour := 0
      if day <6 then day := day + 1
      end
    end
  end
```

The new code goes after the existing `hour := 0` line: the current day updates only when the current hour resets to 0, meaning at midnight. The first increment is done: AutoProof successfully verifies both the `req_9` seamless requirement and the implementation class `CLOCK`.

Increment 2

Seamless requirements `req_10` and `req_11` still fail verification of the `EXTENDED_CLOCK_REQUIREMENTS` class. The second increment consists of implementing the `req_10` requirement. This requirement describes the conditions, under which a clock tick resets the current day to Monday.

Applying the rules from Section 10.4.3 to `req_10` results in the following postcondition assertion:

```
old second = 59 and old minute = 59 and old hour = 23 and old day = 6 implies day = 0
```

Correctness of the inferred assertion follows from the fact that `req_10` now passes verification by AutoProof.

Attempts to verify the `CLOCK` class fail, which means that the current implementation of the `tick` feature does not meet the new postcondition assertion. The antecedent of the assertion is different from the preceding one only in the day-related part. This naturally leads to extending the **if** block, introduced in Section 10.4.4, with an **else** block:

```
tick
  do
    -- Other lines of code
    else hour := 0
      if day <6 then day := day + 1
      else day := 0
      end
    end
  end
```

An application of AutoProof to the `CLOCK` class confirms correctness of the modified implementation.

Increment 3

The last increment consists of implementing the seamless requirement `req_11`. The requirement states that nothing happens to the current day in the event of a tick if the current second is smaller than 59.

Here is the new assertion that results from applying the postcondition inference rules to `req_11`: `old second <59 implies day = old day`. This time not only the seamless requirement passes verification by AutoProof: the existing implementation of the `tick` feature does not need any changes, which follows from the fact that the `CLOCK` class passes verification. Since the `req_11` is a safety requirement ("nothing bad happens"), this result should not come as a surprise: no malicious code was introduced during implementation of the preceding requirements.

Implementation of the new seamless requirements is done: both the requirements class `EXTENDED_CLOCK_REQUIREMENTS` and the respective implementation class `CLOCK` pass verification by AutoProof.

10.5 Related work

10.5.1 Dafny

Dafny [Lei10] is a direct example of a setting other than Eiffel/AutoProof in which the seamless requirements method is applicable. The verification approach which AutoProof currently uses is more complicated than that of Dafny (partially because Dafny does not support inheritance and information hiding, but not only), which is why it may make more sense to use the latter for getting familiar with seamless requirements.

10.5.2 Test-driven development

Although testing is fundamentally different from program proving, software development through seamless requirements have much in common with test-driven development (TDD) [Fra+03] in terms of the software process. It may be convenient to perceive the new software process as test-driven development where specification drivers replace tests, natural language comments on the specification drivers capture user stories, and program proving replaces testing. One may talk about *verification-driven development* to emphasize these analogies with TDD.

Goal Maintain[TrackSegmentSpeedLimit]
InformalDef *A train should stay below the maximum speed the track segment can handle*
FormalDef $\forall\, tr : Train, s : TrackSegment \bullet On(tr, s) \Rightarrow tr.Speed \leq s.SpeedLimit$

Figure 10.10: An example of a goal-oriented requirement from [Lam01].

```
maintain_track_segment_speed_limit (tr: TRAIN; s: TRACK_SEGMENT)
-- A train should stay below the maximum speed the track segment can handle
  require
    tr.on (s)
  do
  ensure
    tr.speed ≤ s.speed_limit
  end
```

Figure 10.11: The goal-oriented requirement Maintain[TrackSegmentSpeedLimit] (Figure 10.10) in the form of a seamless requirement.

10.5.3 State-based notations

State-based specifications characterize the admissible system states at some arbitrary snapshot [Lam09]. Languages such as Z, VDM, B, Alloy, OCL rely on the state-based paradigm. The absence of the imperative layer is what makes state-based notations inapplicable for specification of abstract requirements. State-based notations are purely declarative notations in which one cannot say "if some property holds for a set of objects and I modify some of them through some commands, then another property will hold for these objects".

10.5.4 Goal-oriented requirements engineering

Goal-oriented requirements [Lam01] are suitable for addressing the gap between agile and formal development (Section 10.1): goals are self-contained and have place for both formal and informal semantics of requirements. Goals are self-contained because they can be modified locally. With diagrammatic notations, for example, one has to project a self-contained requirement statement onto different portions of a diagram, thus threatening locality of future modifications. Self-contained representations maintain locality during the requirements formalization process. The goal in Figure 10.10, for example, formalizes an informal requirement as a first-order logic formula, which is as self-contained as the informal version.

Goals, on the other hand, do not bridge the semantical gap between formal requirements notations and programs because the approach does not treat a programming language as a formal notation. Goals also fail to bridge the gap between construction and verification: the need to translate them into tests is still there.

Seamless requirements approach, while bringing the same benefits as goals do, offers strong pairwise connection between requirements, specifications and code.

```
maintain_track_segment_speed_limit_without_contract (tr: TRAIN; s: TRACK_SEGMENT)
-- A train should stay below the maximum speed the track segment can handle
  do
    if tr.on (s) then
      check tr.speed ≤ s.speed_limit end
    end
  end
```

Figure 10.12: The goal-oriented requirement Maintain[TrackSegmentSpeedLimit] (Figure 10.11) in the form of a seamless requirement without a contract.

```
class TRAIN
feature
  speed: INTEGER

  on (s: TRACK_SEGMENT): BOOLEAN
    do
    ensure
      Result implies speed ≤ s.speed_limit
    end
end
```

```
class TRACK_SEGMENT
feature
  speed_limit: INTEGER
end
```

Figure 10.13: Specification of classes `TRAIN` and `TRACK_SEGMENT` that meets the `maintain_track_segment_speed_limit` requirement (Figure 10.11).

The `maintain_track_segment_speed_limit` seamless requirement (Figure 10.11) captures the semantics of the corresponding goal (Figure 10.10) in terms of Eiffel programming constructs understandable to Eiffel programmers, though it may be rewritten without contracts at all through `if` and `check` (known as `assert` in other languages) statements (Figure 10.12). The last option may be useful in languages without contracts. Successful verification of the `maintain_track_segment_speed_limit` requirement assumes strong enough specification of classes `TRAIN` and `TRACK_SEGMENT` (Figure 10.13). Successful verification of the specified classes assumes, in its turn, implementing the `TRAIN::on` routine correctly.

10.5.5 Literate programming

Knuth was the first one to apply interwoven notations in programming [Knu84]. Meyer criticized the approach as inapplicable to object-oriented programming and proposed the multirequirements [Mey13] method (Section 10.5.6):

> When I first read about literate programming I was seduced by the elegance of the approach, but found it inapplicable to modern, object-oriented programming which (as discussed in several publications including [Mey97]) is fundamentally bottom-up as implied by the focus on reuse; literate programming seemed inextricably tied to the top-down, function-driven programming style of the nineteen-seventies. In that traditional view, a program implements a single "main" function; as a consequence the "literate"

text is the sequential telling, cradle to grave, of a single story.

10.5.6 Multirequirements

A multirequirement is a combination of a natural language statement and a small piece of the resulting program; the program piece should rephrase what the natural language part says. The multirequirements method [Mey13] adapts Knuth's idea of interwoven notations to object-oriented programming, while focusing on traceability. The method suggests using three notation layers: natural language layer, formal layer, and graphical layer. For the formal layer, it suggests usage of pieces of the presumable final program. When the requirements specification phase is over, specialized tools then take those pieces and merge them into the program skeleton. The tools are also responsible for taking care of both up- and down-traceability. The approach conceptually removes the fundamental flaw of literate programming, which is the need to write a complete story from the beginning to the end.

Michael Jackson in his work [Jac14] criticizes piecemeal construction of cyber-physical systems. Apart from the details of that particular work, the multirequirements method possesses several flaws that are of concern for us:

- The presumed additional tools responsible for keeping the requirements document and the resulting program in sync do not seem trivial to implement. The method assumes that any person responsible for requirements specification admits the concern for traceability and connects natural language descriptions with the corresponding program pieces through special anchors. As a consequence, the tools should also be able to detect mistakenly placed anchors as well as to warn of their potential absence.

- The method is applicable only to "forward" development. There is no way to prove that an existing program meets a multirequirement. The programmatic part of a multirequirement is, conceptually, a small piece of the program itself. In order to submit a multirequirement to formal verification, it is necessary first to integrate that piece into the main program. This process changes the original program, which is why the very notion of verifying a program with respect to a multirequirement does not exist.

- The multirequirements method assumes a strong bias of the requirements specification notation toward features of a specific programming language (Eiffel in that particular work). A seamless requirement is a command with a pre- and a postcondition expressed in terms of its formal arguments. Such commands are a kind of construct available in any modern programming language with contracts, such as Dafny, Spec# or D.

Applicability of the multirequirements method was studied on a realistic example in Chapter 7.

10.6 Summary

As the development case illustrates, seamless requirements empower software engineering with the following properties:

- Unity of software construction and verification: seamless requirements stimulate construction and, at the same time, are suitable for checking correctness of the deliverables.
- Unity of functional requirements and code: the requirements document becomes one of the classes in the source code repository, readable by both customers and developers.
- Independence from a particular development model choice: there is no need to adjust the requirements notation in the event of switching the development model on the go.
- Traceability for free: existing features of the underlying IDE are suitable for traceability management in the following form:
  - to trace a seamless requirement to specification (downward traceability [Lam09]) is to retarget the IDE to the definitions of the implementation classes and features that occur in the requirement; this functionality is present in some form in any modern IDE.
  - tracing a class or a feature to requirements that constrain it (upward traceability [Lam09]) reduces to an application of the "Show Callers" feature, which is also present in all modern IDE's (up to a name); every call from the requirements class is done by some seamless requirement.

10.6.1 Limitations of the example

Several potential complications were ignored in favour of simplicity of the narrative:

- There is only one command in the clock example: `tick`. Despite this, the approach scales to multi-command examples. Specification drivers, which serve as the formal layer of seamless requirements, are capable of handling cases with an arbitrary number of commands (Chapter 8).
- The `tick` command does not accept any formal arguments. In fact, the approach scales to the case with formal arguments: if a seamless requirement describes desirable behavior of a command with a formal argument, the corresponding routine may assume the presence of the argument through extending the list of its own formal arguments (Equation (8.1)).

The postcondition inference logic from Section 10.4.3 only work in the context of these two simplifications. In general, inference of a sufficiently strong postcondition does not seem to be a solvable problem.

10.6.2 Limitations of the approach

As the primary concern of the approach is functional correctness, all questions related to the suitability of seamless requirements for non-functional requirements lie expressly outside of the chapter's scope.

Another assumption that underlies this approach is the use of a programming language with contracts plus the existence of a prover for this language. This assumption is adequate: Eiffel plus AutoProof is not the only representative of this technology combination. The "Code Contracts for .NET" project [Bar10] offers similar benefits in the .NET world.

Seamless requirements approach is applicable to non-concurrent programs. Although the approach may have potential in concurrent setting too, the question is not studied yet.

10.6.3 Future work

Translation between the notations

The seamless requirements approach poses an immediate question: how to check the consistency between the natural language and the programming language components? Currently there is no way to do that. Work [Mey85] describes a requirements refinement process that relies on round trip engineering: given a natural language requirement translate it into a formal form and then back and see how close the result is to the original statement. This process needs support in the form of two tools that would perform the necessary translations. Development of these tools is the immediate goal of the present research.

Consistency of seamless requirements

Another research question is: how to understand if seamless requirements are consistent with each other? With an inconsistent set of requirements it will never be possible to develop a provably correct solution. With trial-and-error considerable amount of resources may be spent before the inconsistency becomes apparent. How could one detect inconsistencies in requirements before initiating implementation of a solution?

Chapter 11

Specifying and Verifying Control Software

The considerable effort of writing requirements is only worthwhile if the result meets two conditions: the requirements reflect stakeholders' needs, and the implementation satisfies them. In usual approaches, the use of different notations for requirements (often natural language) and implementations (a programming language) makes both conditions elusive. AutoReq, presented in this chapter, takes a different approach to both the writing of requirements and their verification. Applying the approach to a well-documented example, a landing gear system, allowed for a mechanical proof of correctness and uncovered an error in a published discussion of the problem.

11.1 Overview and main results

A key determinant of software quality is the quality of requirements. Inconsistent or incomplete understanding of the requirements can lead to catastrophic results. We present a tool-supported method, AutoReq, for producing verified requirements, with applications to control software. It illustrates it on a standard case study, an airplane Landing Gear System (LGS). The goal is to obtain requirements of high quality:

- Easy to write.
- Clear and explainable to domain experts.
- Amenable to change.
- Supporting traceability through close connections to later development steps.
- Amenable to mechanical verification and validation.

As the last point indicates, AutoReq includes techniques for not only expressing requirements but also verifying their correctness. The LGS case study illustrated the effectiveness of such verification by uncovering a significant error in a previous description of this often-studied example (Section 11.6.5).

AutoReq takes natural language requirements and environment assumptions as an input and converts them into a format having the above properties. The new format relies on a programming language with contracts. This viewpoint brings one of the biggest advantages of AutoReq – it makes the requirements verifiable both against the underlying assumptions and future candidate implementations, while maintaining their readability through natural language comments on the code. We take the natural language statements from the LGS case study and translate them to seamless statements, readable and verifiable. The ASM treatment of the case study [AGR17] provides the candidate implementation – an executable ASM specification [GH94] of the system. This by no means implies applicability of AutoReq to ASMs only. The approach applies to any candidate implementation that follows the small step semantics of ASMs. More precisely, the implementation should run in an infinite loop polling the system environment's state and sending appropriate control signals. To the best of our knowledge, most control software implementations follow this approach.

The method of expressing requirements does not introduce any new formalism but instead relies on a standard programming language, Eiffel, using mechanisms of Design by Contract (DbC) [Mey92] to state semantic constraints. While DbC relies on Hoare logic [Hoa69], which at first sight does not cover temporal and timing properties essential to the specification of control software, we show that it is, in fact, possible to express such properties in the DbC framework.

The verification part relies on an existing tool, associated with the programming language: AutoProof [Tsc+15], a program proving framework, which can verify the temporal and timing properties expressed in the DbC framework. Applying it to LGS automatically and unexpectedly uncovered the error. Hoped-for advantages include:

- Expressiveness: requirements benefit both from the expressive power of declarative assertions and from that of imperative instructions.
- Ease of learning: anyone familiar with programming languages has nothing new to learn.
- Continuity with the rest of the development cycle: design and implementation may rely on the same formalism, avoiding the impedance mismatches that arise from the use of different formalisms, and facilitating change.
- Precision: formal specifications (contracts) cover the precise semantics of the system and its environment.
- Existing tools, as available in modern IDEs, that support the requirements process: a compiler for a typed language performs many checks that are as useful for requirements as for code.

The AutoReq approach, while not claiming to have fully reached these ambitious goals, makes the following contributions:

- The outline of a general method for requirements engineering with application to control software.

- The use of a programming language as an effective mechanism for requirements specification.
- A precisely defined concept of *verifying requirements* for control software (complementing the usual concept of verifying programs). This idea originates from seamless requirements (Chapter 10).
- A translation scheme from temporal and timing properties to Hoare logic properties (first-order predicates on states) as traditionally used in Design by Contract.
- A way to combine *environment* and *machine* aspects (the two components of requirements in the well-known Jackson-Zave approach).
- A direct mapping of these *requirements* concepts into well-known *verification* concepts, `assume` and `assert`.
- The demonstration that it is possible to use an existing *program* prover to verify requirements.

Section 11.2 discusses consequences of poor requirements. Section 11.3 presents LGS. Section 11.4 describes the methodology: how to specify and verify requirements. Section 11.5 shows how to translate common requirements patterns (originally expressed through temporal logic, timing constraints or Abstract State Machines) into a form suitable for AutoReq. Section 11.6 sketches the method's application to the case study, including an analysis of the uncovered error. Section 11.7 discusses related work, and Section 11.8 discusses limitations and future work.

11.2 The importance of verifying requirements

Control software in aerospace, transportation, and other mission-critical areas raise tough reliability demands. Ensuring reliability begins with the quality of requirements: the best implementation is useless if the requirements are inconsistent or do not reflect needs. Requirements for software deserve as much scrutiny as other artifacts such as code, designs, and tests.

The literature contains many examples of software disasters arising from requirements problems of two kinds:

- In the requirements themselves: inconsistencies, incompleteness, inadequate reflection of stakeholders' needs.
- In their relationship to other tasks: design, implementation etc. may wrongly understand, implement or update them.

Examples of the first kind include [Lak10]:

- The year 2000, National Cancer Institute, Panama City: patients undergoing radiation therapy get wrong doses because of a software miscalculation.

- In 1996, Ariane 5 maiden flight fails from flight computer's code crash, out of an uncaught arithmetic exception, in code that was reused from Ariane 4 but relied on assumptions that no longer hold in the new technology.
- In 1990, a bug in software for AT&T's #4ESS long-distance switches crashes computers upon receipt of a specific message sent out by neighbors when recovering from a crash.

Analysis of these examples suggests that the problem lies in part from the use of different methods and of different notations for requirements and other tasks such as implementation. This observation is a basis for the *seamless* approach ([Mey97], [WN94], [Mey13], following ideas in [Rum+91]), which AutoReq applies by using a single notation throughout.

Examples of the second kind include [Lam09]:

- London underground system: several cases [Neu95] of passenger deaths from doors opening or closing unexpectedly, without an alarm notification being sent to the train driver.
- An aerospace project [HF01] where 49% of requirements errors were due to incorrect facts about the problem world.
- An inadequate assumption about the environment of the flight guidance system, which may have contributed to the crash of a Boeing 757 in Cali [Mod+97]. Location information for the pilot to extend the flap arrived late, causing the guidance software to send the plane into a mountain.

These examples and others in the literature illustrate the importance of *verifying* requirements. We will see that it is possible to apply to requirements both the concept of verification, as commonly applied to code, and modern proof-oriented verification tools devised initially for code.

11.3 The Landing Gear System

To illustrate AutoReq, we will use, rather than examples of our own making, the LGS [BW14], probably the most widely discussed case study in the control software literature, e.g. [SA17], [AGR17], [DT14], [Lad+17], [ML17], [BDZF14], [Ban17].

The Landing Gear System physically consists of the landing set, a gear box that stores the gear in the retracted position, and a door attached to the box (Figure 11.1). A digital controller independently actuates the door and the gear. The controller initiates either gear extension or gear retraction depending on the current position of a handle in the cockpit. The task is to program the controller so that it sends the correct signals to the door's and the gear's actuators.

The discussion will restrict itself to the system's *normal mode* (there is also a *failure mode*). The defining properties are the following:

R_{11}*bis*: When the landing gear handle has been pushed down and stays down, then eventually the gear will be seen extended and the doors will be seen closed. We

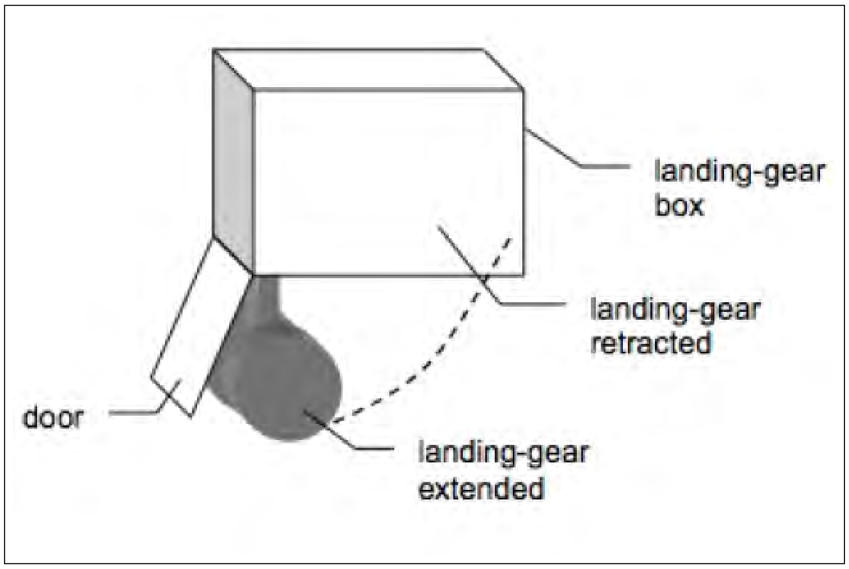


Figure 11.1: Landing set (from Boniol et al. [BW14]).

interpret this requirement in LTL as $\Box(\Box \mathit{handle\_down} \Rightarrow \Diamond(\mathit{gear\_extended} \wedge \mathit{door\_closed}))$ where $\Box$ stands for the *always* temporal operator, and $\Diamond$ stands for the *eventually* temporal operator.

R_{12}*bis*: When the landing gear handle has been pulled up and stays up, then eventually the gears will be seen retracted and the doors will be seen closed. We interpret this requirement in LTL as $\Box(\Box \mathit{handle\_up} \Rightarrow \Diamond(\mathit{gear\_up} \wedge \mathit{door\_closed}))$.

R_{21}: When the landing gear handle remains in the *down* position, then retraction sequence is not observed. We interpret this requirement in LTL as $\Box(\mathit{handle\_down} \Rightarrow \bigcirc\neg\ \mathit{gear\_retracting})$ where $\bigcirc$ stands for the *next* temporal operator.

R_{22}: When the landing gear handle remains in the *up* position, then outgoing sequence is not observed. We interpret this requirement as $\Box(\mathit{handle\_up} \Rightarrow \bigcirc\neg\ \mathit{gear\_extending})$.

We will work not from the original description of the LGS but from one of the most interesting treatments of case study [AGR17], which uses the abstract state machine (ASM) approach and applies a process of successive refinements:

1. Start with a *ground model* covering a subset of the requirements.
2. Model-check it.
3. Repeatedly extend (refine) it with more properties of the system, proving the correctness of each refinement.

The AutoReq specification discussed in the next sections starts from the ASM ground model. Some of its features are a consequence of this choice:

- It only accounts for properties specified in the first of the successive models in [AGR17].
- As already noted, it only covers *normal mode*.
- Like the ASM model, it assumes that the only *environment-controlled machine-visible* phenomenon is the pilot's handle [JZ95]. In the failure mode, there might be others.
- It takes over from the ASM model such instructions as $\textit{gears} := \textit{RETRACTED}$ which posit that the control software has a way to send the gear to the retracted position in one step. This assumption is acceptable at the modeling level but not necessarily true in the actual LGS system.
- The ASM-to-Eiffel translation scheme (Section 11.5.4) ensures preservation of the one-step semantics of ASM.

11.4 Requirements methodology

AutoReq builds on the ideas of seamless development [Mey97], [WN94], multirequirements [Mey13] and seamless requirements (Chapter 10). The new focus is on requirements verification and reuse of previous requirements through a routine call mechanism. We examine in turn how to specify and reuse requirements and environment assumptions (Section 11.4.1), and what it means to verify them (Section 11.4.2).

11.4.1 Specifying requirements

Specifications in AutoReq, often in practice translated from a document in structured natural language, take the form of contracted Eiffel routines with natural language comments. These routines are further consumed by:

- The verification tool. Since the routines coming out of the translation process are equipped with contracts, they may be formally verified by a Hoare logic based prover.
- Possible implementers of the system. The combination of a programming language and natural language helps developers, who will use the same programming language for implementation, understand the requirements. The contracts state the semantics.

Previous work ([Mey13], Chapter 10) explains the reasons for choosing this mixed notation: unity of software construction and verification, unity of functional requirements and code, use of complementary notations geared towards different stakeholders.

Additional properties are specific to control software:

- Specification of temporal assumptions and requirements.

- Specification of timing assumptions and requirements.
- Reuse of assumptions and requirements in stating new ones.

The basic notation is Eiffel. All the examples have been processed by the EiffelStudio IDE [Eif], compiled, and processed by the AutoProof verification environment. The interest of compilation is not in the generated code, since at this stage the Eiffel texts represent requirements only, but in the many correctness controls, such as type checking, of a modern compiler.

The requirements can and do take advantage of object-oriented mechanisms such as classes, inheritance and genericity.

There is sometimes an instinctive resistance to using a programming language for requirements, out of the fear of losing the fundamental difference between the goals of the two steps: programming languages normally serve for implementation, while requirements should be descriptive. The AutoReq approach, however, uses the programming language not for implementation but for specification, restricting itself to requirements patterns discussed next. The imperative nature of these patterns does not detract from this goal; empirical evidence indeed suggests [Fah+09b], [Pic+11], [Fah+09a] that operational reasoning works well not just for programmers but for other requirements stakeholders. An added benefit is the availability of program verification tools, which AutoReq channels towards the goal of verifying requirements.

For this verification goal, there seems to be a mismatch between the standard properties that program verification tools address and the needs of control software. Program verification generally relies on Hoare logic properties as embodied in Eiffel's Design by Contract: properties of the program state (or, for postconditions, two states). The specification of control software generally relies on temporal and timing requirements, involving properties of an arbitrary number of (future) states of the system. A contribution of this work is to resolve the mismatch, using the programming language to emulate temporal and timing properties, through schemes described in Section 11.5.

11.4.2 Verifying requirements

Verification of AutoReq requirements relies on AutoProof [Tsc+15], the prover of contracted Eiffel programs. AutoProof is a Hoare logic [Hoa69] based verifier that follows *semantic collaboration* [Pol+14] – a specification and verification methodology adapting Hoare logic to specific needs of object-oriented programming. The verification unit of AutoProof is feature with contracts. AutoReq assumptions and requirements take the form of such features, with natural language comments for better readability, to enable their direct verification with AutoProof.

Contracts for verification with AutoProof may be modular – visible to the feature's callers, and non-modular – visible only in the feature's implementation. Modular contracts take the following forms:

- *Precondition* imposes obligations on the feature's callers and benefits the callees' implementation.
- *Postcondition* guarantees benefits to the callers and imposes obligations on the callees' implementation.

Non-modular contracts take the following forms, going back at least as far as ESC-Java [CK04]:

- `assume` `x` `end` *allows* the verification to take advantage, at the given program point, of property `x`, adding `x` to the set of properties that the prover *may* use (assumption).

- `assert` `x` `end` *requires* the verification to establish `x` before going beyond the program point, adding `x` to the set of properties that the prover must prove (proof obligation).

Both *precondition* and `assume` contracts add information to verifying the *postcondition* and `assert` contracts, but preconditions impose verification obligations on their own: they have to hold whenever the respective features are called. AutoReq requirements take the form of features with non-modular contracts because of their fundamental connection with the core requirements engineering terminology, as discussed further. From the purely technological perspective, AutoReq depends on the ability of AutoProof to inline callees' non-modular contracts into the callers' code.

AutoReq specifications include formal properties which can be submitted to proof tools for verification. Jackson & Zave's seminal work ([JZ95], also van Lamsweerde [Lam09]), introduced a fundamental division of these properties:

- *Environment* (or *domain*) assumptions characterize the context in which the system must operate. The development team has no influence on them.

- *Machine* (or *system*) properties characterize what the system must do. It is the job of the development team to work on them.

Although each of these two distinctions is well-known and widely used in the corresponding sub-community of software engineering, respectively requirements and formal verification, the existing literature does not, to our knowledge, connect them. The AutoReq approach, covering both requirements and verification concepts, unifies them into a single distinction:

- `assume` `E` `end` specifies an *environment assumption* `E`.

- `assert` `E` `end` specifies a *machine property* `E`.

Verifying requirements in AutoReq means proving that all `assert` hold, being permitted to take `assume` for granted.

Notational convention: the above notations are for presentation. The actual texts verified through the process reported in the next sections use the following standard Eiffel equivalents:

- For `assert` `x` `end`, the notation in the actual Eiffel texts is `check` `x` `end` (`check` is a standard part of Eiffel's Design by Contract mechanism).

- For `assume` `x` `end`, the Eiffel notation is `check` `assume:` `x` `end`. The `assume` tag is a standard part of the notation for programs to be verified by AutoProof. `old` `e`, in a routine body, denotes the value of an expression `e` on routine entry.

The only difference with verifying programs comes from the elements that appear between these assertions: in program verification, they may include any instructions; in requirements verification, we only permit patterns discussed below (Section 11.5.1). In addition, specifications include timing properties, using the translation into classic assertions described in Section 11.5.2 and Section 11.5.3.

Formal methods and notations are essential for one of the goals of AutoReq (precision/completeness, see Section 11.1), but non-technical stakeholders sometimes find them cryptic at first sight, hampering other goals such as readability and ease of use. The *multirequirements* approach [Mey13], which AutoReq extends, addresses the problem by using complementary views, kept consistent, in various notations: formal (such as Eiffel or a specification language), graphical (such as UML) and textual (such as English). In line with this general idea, AutoReq specifications rely on systematic commenting conventions (somewhat in the style of Knuth's *literate programming* [Knu84]). A typical example from the specification in the next section is

```
-- Assume the system
run_in_normal_mode
```

The second line is formal; the comment in the first line puts it in context. Such seemingly informal comments follow precise rules. For non-expert users, and for the sake of discussion, it is enough to treat them as natural language explanations.

11.5 Structuring a control software specification

The mechanisms of the preceding section enable us to write the requirements for control software and verify them. Such specifications will follow standard patterns:

- Overall structure of control software implementations (Section 11.5.1).
- Translation rules for temporal properties (Section 11.5.2).
- Translation rules for timing properties (Section 11.5.3).
- Translation rules for ASM properties (Section 11.5.4).

These schemes and translation patterns are fundamental to the methodology because they govern the use of the programming language. While the methodology relies on a programming language for expressing requirements, it does not use its full power, since some of its mechanisms are only relevant for programs. Programming language texts expressing requirements stick to the language subset relevant to this goal.

The translation schemes of Section 11.5.2, Section 11.5.3 and Section 11.5.4 guarantee that their output will conform to these patterns. A goal for future work (Section 11.8) is to formalize the input languages, timed temporal logic and ASM, and turn the translation patterns into formal rules and automatic translation tools.

Pending such formalization, we did not for now address the soundness of the translation.

	Temporal Properties	**Timing Properties**
Environment Assumptions	P1	P3
System Obligations	P2	P4

Table 11.1: The map of AutoReq translation patterns.

11.5.1 Representing control software

Control software is typically (unlike most sequential programs) repeating and non-terminating. AutoReq uses programs of the form

```
from until False loop main end
```

to represent control software. The task of the requirements is then to specify `main`.

The translation uses four patterns that look like Eiffel features with non-modular (`assume` and `assert`) contracts. These patterns are not part of AutoProof, but they serve as blueprints for features that AutoProof can verify. *P1* and *P2* (Section 11.5.2) are time-independent (although *temporal* in the sense of temporal logic). *P3* and *P4* (Section 11.5.3) take timing into account. These cases suffice for the examples addressed with AutoReq so far. Translation schemes are possible for more general LTL/CTL/TPTL schemes if the need arises in the future.

The patterns use the Jackson-Zave distinction (Section 11.4.2) between describing an environment assumption and prescribing an expected system (machine) property. Specifically: *P1* and *P3* correspond to environment assumptions (respectively time-independent and timed); *P2* and *P4* correspond to system obligations (with the same distinction). The Eiffel translations accordingly use `assume` for *P1* and *P3* and `assert` for *P2* and *P4*. When asked to verify an AutoReq requirement, AutoProof tries to infer the `assert` statements by simulating an execution of the requirement's body to a state satisfying the `assume` statements. Table 11.1 maps the patterns according to the taxonomy of system properties used in the present chapter.

11.5.2 Translating temporal properties

In the control software world, the starting point for requirements is often a description expressed in a temporal logic, usually LTL [Pnu77], CTL [BPM83], or a timed variant such as propositional temporal logic (TPTL [AH94]). Even if not using a specific formalism, they often state temporal properties such as *all future system states must satisfy a given condition* or *some future state must satisfy a given condition*. The LGS properties given in Section 11.3 are an example.

- *P1* (environment assumption)
  Consider the system running in mode cs under assumption c. The LTL formulation is $\square(c \wedge cs)$.

- *P2* (system obligation)
  The system running in mode cs should immediately meet property p. The LTL formulation is $\square(cs \Rightarrow \circ p)$. This property constrains the system to maintain response p whenever stimulus cs holds.

The translation scheme for *P1* is:

```
-- Assume the system
run_under_condition_c
  do
    assume
      c
    end
    main_under_conditions_cs
  end
```

where `main_under_conditions_cs` is of the form *P1* or *P3*. The `run_under_conditions` routine should be used instead of the original `main` in all requirements that talk about the system operating in mode `c`. This pattern may be useful for encoding $\Box c$ in properties of the form $\Box(\Box c \Rightarrow \Diamond d)$.

The translation scheme for *P2* is:

```
-- Require the system to
immediately_meet_property_p
  do
    main_under_conditions_cs
    assert
      p
    end
  end
```

where `main_under_conditions_cs` is of the form *P1* or *P3*.

11.5.3 Translating timing properties

Although not all approaches to requirements take time into account, timing requirements, such as *the response time must not exceed 1 second*, are essential to the proper specification and implementation of control software. AutoReq recognizes the following timing-related patterns:

- *P3* (environment assumption)
  Assume the system running in mode *cs* spends *t* time units to meet property *p*. The TPTL formulation is $\Box x.((cs \land \neg p) \Rightarrow \circ y.(p \Rightarrow y = x + t))$. $x.$ and $y.$ record the current time of corresponding states [AH94].

- *P4* (system obligation)
  The system running in mode *cs* should spend no more than *t* time units to meet property *p*. In TPTL: $\Box x.(\Box cs \Rightarrow \Diamond y.(p \land y \leq x + t))$.

The translation scheme for *P3* is:

```
-- Assume it takes t time units to take the system
from_not_p_to_p:
  do
    main_under_conditions_cs
    if (not old p and p) then
      duration := duration + t
    end
  end
```

The technique for timing system obligations of the *P4* form differs from the others by using loops as the core mechanism:

```
-- Require that
meeting_p_under_persistent_conditions_cs
-- never takes more than t time units:
  do
    from
      main_under_conditions_cs
    until
      p or (duration − old duration) > t
    loop
      main_under_conditions_cs
    end
    assert
      p and (duration − old duration)≤ t
    end
  end
```

where `main_under_conditions_cs` is of the form *P1* or *P3*. The `(duration − old duration)> t` exit timeout condition ensures termination of the loop, and assertion `(duration − old duration)≤ t` checks that the timeout condition has not been reached.

The technique for handling the timing-related patterns relies on an integer, non-decreasing auxiliary variable `duration`. It has the same role as x and y in the TPTL formulations. The `duration` variable is part of the AutoReq approach – not a predefined variable nor part of AutoProof. It does not play a role in the actual execution of the system but caters to static reasoning about the system's timing properties. The `from_not_p_to_p` routine updates the value of `duration` instead of using **`assume`**, which would lead to a contradiction: the prover would detect that the variable was not, in fact, updated, and would infer **`False`** from assuming the opposite.

11.5.4 Translating ASM specifications

Abstract State Machines [GH94], are a commonly used specification formalism for control software, and the treatment of the LGS case study in [AGR17] served as a starting point for our own treatment of the example. We do not formally prove soundness of the ASM-to-Eiffel translation. The decision to work with the ASM treatment was motivated by the general ASM specifications' executability: fundamentally, they are verifiable abstractions of infinitely running control software. Such software may be implemented in a general-purpose programming language, and the present chapter demonstrates that such a language may serve as a verifiable abstraction of itself, in the presence of a program prover.

Below comes the ASM-to-Eiffel translation scheme. The translation scheme omits the nondeterministic version of the ASM formalism. The original work [GH94] presents *"Nondeterministic Sequential Algebras"* as an extension to the basic model. As Section 11.1 explains, the ASM formalism serves as an implementation language example in the present discussion of AutoReq, with no intent of covering every aspect of ASMs. Nondeterministic updates seem to be inappropriate for implementing mission- and life-critical software, such as the LGS controller, and control software in general. Every possible environment's state should be predictably handled in such systems. The ASM treatment of the LGS, for example, does not use nondeterminism.

A basic ASM specification is a collection of rules taking one of three forms [Gur00]: assignment, do-in-parallel and conditional. An ASM *assignment* reads:

$$f(t_1, .., t_j) := t_0 \tag{11.1}$$

The semantics is: update the current content of location $\lambda = (f, (a_1, .., a_j))$, where $a_{i:\{1..j\}}$ are values referenced by $t_{i:\{1..j\}}$, with the value referenced by t_0.

The Eiffel representation for an ASM location is an attribute (field) of the class; the representation for a location update is an attribute assignment.

The ASM *do-in-parallel* operator applies several assignments in one step. Eiffel offers no native support for do-in-parallel, but it can emulate one sequentially without changing the behavior. The following example gives intuition behind the translation idea:

$$a, b := max(a - b, b), min(a - b, b) \tag{11.2}$$

The instruction in Equation (11.2), when run infinitely, reaches the fixpoint in which a contains the greatest common divisor of a and b. The Eiffel translation of this instruction is:

```
local
  a_intermediate, b_intermediate: INTEGER
do
  a_intermediate := max (a−b, b)
  b_intermediate := min (a−b, b)
  a := a_intermediate
  b := b_intermediate
end
```

The generalization should be clear at this point: instead of updating the target locations, introduce and update intermediate local variables, and then assign them to the target locations.

The translation of an ASM *conditional* (`if t then R else Q`) is an Eiffel conditional instruction.

The ASM-to-Eiffel translation scheme scales out to the multiple classes case. The translation overhead in this case consists of implementing assigner procedures for the supplier classes' attributes. The assigner procedures will make it possible for the clients to update the suppliers' attributes while keeping them consistent. The LGS example is simple enough to avoid the multiple classes case, which is why this translation rule does not apply to the analyzed example.

11.6 The Landing Gear System in AutoReq

Equipped with the AutoReq mechanisms as described, we can now see the core elements of the AutoReq specification of the LGS example. The entire example is available in a public GitHub repository [Nau17].

11.6.1 Normal mode of execution

Execution runs in *normal mode* if all the parameter values are in the expected ranges and meet the system invariant. Application of the `run_under_condition_c` pattern results in the following Eiffel model of normal mode:

```
-- Assume the system
run_in_normal_mode
  do
    -- the handle status range:
    assume
      handle_status = up_position or
      handle_status = down_position
    end
    -- the door status range:
    assume
      door_status = closed_position or
      door_status = opening_state or
      door_status = open_position or
      door_status = closing_state
    end
    -- the gear status range:
    assume
      gear_status = extended_position or
      gear_status = extending_state or
      gear_status = retracted_position or
      gear_status = retracting_state
    end
    -- the gear may extend or retract only with the door open:
    assume
      (gear_status = extending_state or gear_status = retracting_state)
        implies door_status = open_position
    end
    -- closed door assumes retracted or extended gear
    assume
      door_status = closed_position implies
        (gear_status = extended_position or gear_status = retracted_position)
    end
    main
  end
```

The first three **`assume`** express that attribute values fall into specific ranges. The last two express the LGS invariant. Ranges, the invariant and the definition of normal mode come from the original. `run_in_normal_mode` is a multiple application of the `run_under_condition_c` pattern (Section 11.5.2). It wraps around `main` to make additional assumptions before calling it.

11.6.2 Timing properties

The ASM treatment of the LGS case study ignores timing properties stated in the original description. For a practical system, timing is essential; an otherwise impeccable LGS that takes two hours to perform *extend landing gear* would not be attractive. We rely on AutoReq's timing mechanisms of the AutoReq methodology (Section 11.5.3) and the `from_not_p_to_p` pattern (Section 11.5.3). Timing values, e.g. 8 units for door closing, are for illustration only. Each of the translations that follow are produced by applying the same pattern, which is why only the first translation is accompanied by a detailed explanation.

- *It takes 8 time units for the door to close.* Replacing `p` with `door_status = closed_position`, and `t` with 8 in the `from_not_p_to_p` pattern yields:

  ```
  -- Assume it takes 8 time units to take the door
  from_open_to_closed -- position:
    do
      run_in_normal_mode
      if (old door_status ≠ closed_position and
         door_status = closed_position) then
        duration := duration + 8
    end end
  ```

- *It takes 12 time units for the door to open*:

  ```
  --Assume it takes 12 time units to take the door
  from_closed_to_open -- position:
    do
      from_open_to_closed
      if (old door_status ≠ open_position and
         door_status = open_position) then
        duration := duration + 12
      end
    end
  ```

- *It takes 10 time units for the gear to retract*: Replacing `p` with `gear_status = retracted_position`, and `t` with 10 in the `from_not_p_to_p` pattern leads to:

  ```
  --Assume it takes 10 time units to take the gear
  from_extended_to_retracted -- position:
    do
      from_closed_to_open
      if (old gear_status ≠ retracted_position and
         gear_status = retracted_position) then
        duration := duration + 10
    end end
  ```

- *It takes 5 time units for the gear to extend*:

  ```
  -- Assume it takes 5 time units to take the gear
  from_retracted_to_extended -- position:
    do
      from_extended_to_retracted
      if (old gear_status ≠ extended_position and
         gear_status = extended_position) then
        duration := duration + 5
    end end
  ```

`from_retracted_to_extended` will include all the previously stated **`assume`** instructions together with `main`.

11.6.3 Baseline requirements

Section 11.3 introduced a set of core LGS requirements, $R_{11}bis$ to R_{22}, which we now express in AutoReq. $R_{11}bis$ and R_{21} talk about the system running with the handle pushed down. Application of the `run_under_condition_c` pattern (Section 11.5.2) with `handle_status = down_position` for `c` results in the following routine to model the required mode of operation:

```
-- Assume the system
run_with_handle_down
  do
    assume handle_status = down_position end
    from_retracted_to_extended
  end
```

`run_with_handle_down` is an application of the `run_under_condition_c` pattern (Section 11.5.2). It calls `from_retracted_to_extended` to include all assumptions so far.

Now that the execution mode with the handle pushed down is formally defined, it is possible to express the requirements in terms of it. Property R_{21} requires the controller to prevent retraction immediately whenever the handle is pushed down. Application of the `immediately_meet_property_p` pattern (Section 11.5.2) with `gear_status ≠ retracting_state` for `p` yields, for R_{21}:

```
-- Require the system to
never_retract_with_handle_down
  do
    run_with_handle_down
    assert gear_status ≠ retracting_state end
  end
  -- known as R_{21}
```

$R_{11}bis$ requires the system eventually to extend the gear and close the door if the handle stays down. The absence of timing makes it unsuitable for the specification of control software: we need to specify an upper bound on the time the system may spend on gear extension. That bound is the sum of the maximal times for door closing, door opening and gear extension. Under earlier assumptions, this value is 25. Applying `meeting_p_under_persistent_conditions_cs` (Section 11.6.2) with `gear_status = extended_position and door_status = closed_position` for `p`, `run_with_handle_down` for `main_under_conditions_cs` and 25 for `t` turns $R_{11}bis$ into:

```
-- Require that
extension_duration
-- never takes more than 25 time units:
  do
    from
      run_with_handle_down
    until
      (gear_status = extended_position and door_status = closed_position) or
      (duration − old duration) > 25
    loop
      run_with_handle_down
    end
    assert gear_status = extended_position end
    assert door_status = closed_position end
    assert (duration − old duration) ≤ 25 end
  end
  -- known as R_{11}bis
```

Requirements $R_{12}bis$ and R_{22} talk about the system running with the handle pulled up. Application of `run_under_condition_c` (Section 11.5.2) with `handle_status = up_position` for `c` yields:

```
-- Assume the system
run_with_handle_up
  do
    assume
      handle_status = up_position
    end
    from_retracted_to_extended
  end
```

The rest of the requirements can rely on the specification of the execution mode with handle up, as we have now obtained.

R_{22} requires the system to prevent immediate extension whenever the handle is pulled up. Application of `immediately_meet_property_p` (Section 11.5.2) with `gear_status ≠ extending_state` for `p` yields, for R_{22}:

```
-- Require the system to
never_extend_with_handle_up
  do
    run_with_handle_up
    assert
      gear_status ≠ extending_state
    end
  end
  -- known as R_{22}
```

$R_{12}bis$ requires the system eventually to retract the gear and close the door if the handle stays up. Like $R_{11}bis$, it does not include timing. The upper bound for $R_{12}bis$ is the sum of the maximal times for door closing, door opening and gear extension, 30 from earlier assumptions. Applying `meeting_p_under_persistent_conditions_cs` (Section 11.6.2) with `gear_status = retracted_position and door_status = closed_position` for `p`, with `run_with_handle_up` for `main_under_conditions_cs` and 30 for `t` yields:

```
-- Require that
retraction_duration
-- never takes more than 30 time units:
  do
    from
      run_with_handle_up
    until
      (gear_status = retracted_position and door_status = closed_position) or
      (duration − old duration) > 30
    loop
      run_with_handle_up
    end
    assert
      gear_status = retracted_position and
      door_status = closed_position and
      (duration − old duration) ≤ 30
    end
  end
  -- known as R_{12}bis
```

11.6.4 Complementary requirements

$R_{11}bis$ and $R_{12}bis$ talk about reaching a desired state under some conditions, but not about preserving it. For example, even if the gear becomes extended and the door closed with the handle down, this situation must not change without the handle pulled up. The following application of `immediately_meet_property_p` (Section 11.5.2) with `gear_status = extended_position and door_status = closed_position` for `p` captures this property:

```
-- Require the system to
keep_gear_extended_door_closed_with_handle_down
  do
    run_with_handle_down_gear_extended_door_closed
    assert
      gear_status = extended_position and
      door_status = closed_position
    end
  end
```

under the assumption that the doors are already closed, the gear is extended, and the handle is down. Application of `run_under_condition_c` (Section 11.5.2) with `gear_status = extended_position and door_status = closed_position` for `c` yields, for this assumption:

```
-- Assume the system
run_with_handle_down_gear_extended_door_closed
  do
    assume
      gear_status = extended_position and
      door_status = closed_position
    end
    run_with_handle_down
  end
```

The state with the gear retracted, the door closed and the handle pulled up should be stable without pushing the handle down. The following application of `immediately_meet_property_p` (Section 11.5.2) with `gear_status = retracted_position and door_status = closed_position` for `p` yields:

```
-- Require the system to
keep_gear_retracted_door_closed_with_handle_up
  do
    run_with_handle_up_gear_retracted_door_closed
    assert
      gear_status = retracted_position and
      door_status = closed_position
    end
  end
```

under the assumption that the doors are already closed, the gear is retracted, and the handle is up. Application of `run_under_condition_c` pattern (Section 11.5.2) with `gear_status = retracted_position and door_status = closed_position` for `c` yields, for this assumption:

```
-- Assume the system
run_with_handle_up_gear_retracted_door_closed
  do
    assume
      gear_status = retracted_position and
      door_status = closed_position
    end
    run_with_handle_up
  end
```

11.6.5 An error in the ground model

Contracts do not just yield expressive power: they also make automatic verification possible in the AutoReq approach thanks to AutoProof. One of the principal potential benefits would be to uncover errors in the requirements.

Our work on the LGS example shows that this benefit is not just a theoretical possibility. Applying the AutoReq method and tools to the published ASM specification of the LGS system [AGR17] uncovered an error. The verification process applied the following sequence of steps.

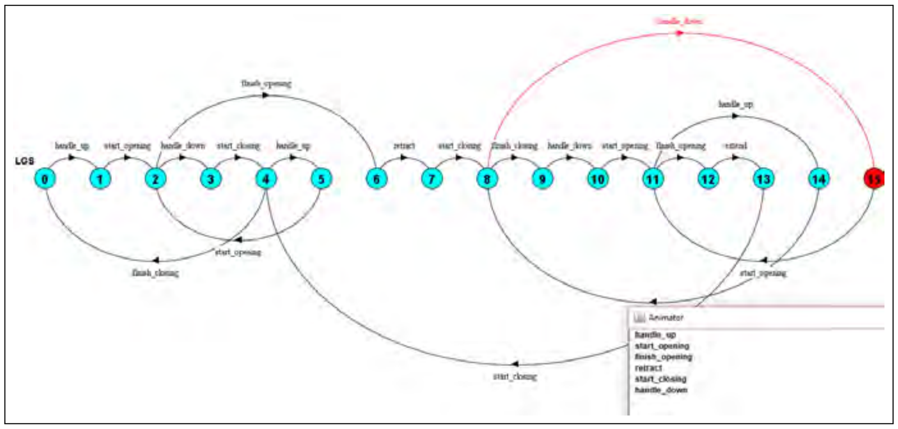
1. *Start from the ASM specification.* The language in which the ASM specification is expressed contains syntactic sugar in addition to the standard ASM operators. The first step consisted of analyzing these additional constructs to understand how they should translate to Eiffel.

2. *Translate it into Eiffel.* This step consisted of manual translation of the specification and the requirements to Eiffel. One can find the original ASM specification in an online archive [ML14], inside the *LandingGearSystemGround.asm* file. File *ground_model.e* in the GitHub repository [Nau17] contains the result of the translation.

3. *Verify it with AutoProof.* Note that AutoProof, by default, performs modular contract-based verification. AutoReq specification techniques rely on `assume` and `assert` rather than traditional contracts. These specification techniques require tuning AutoProof command-line options. The GitHub repository [Nau17] with the Eiffel translation includes a *readme* file that says in detail how to launch AutoProof.

4. *Identify the error.* When AutoProof reports a verification failure, it does not point at its root cause. The last step was devoted to identifying that cause.

The error uncovered by this procedure is subtle and revealing: *The specification does not meet the* R_{11}*bis requirement, which states that pushing the handle down should lead to the gear extended and the door closed.* Normally, when the crew pushes



handle_up
start_opening
finish_opening
retract
start_closing
handle_down

Figure 11.2: A correctly working LGS state machine. Pushing the handle down cancels the gear retraction process and initiates gear extension. The bottom-right box contains the trace leading to state 15.

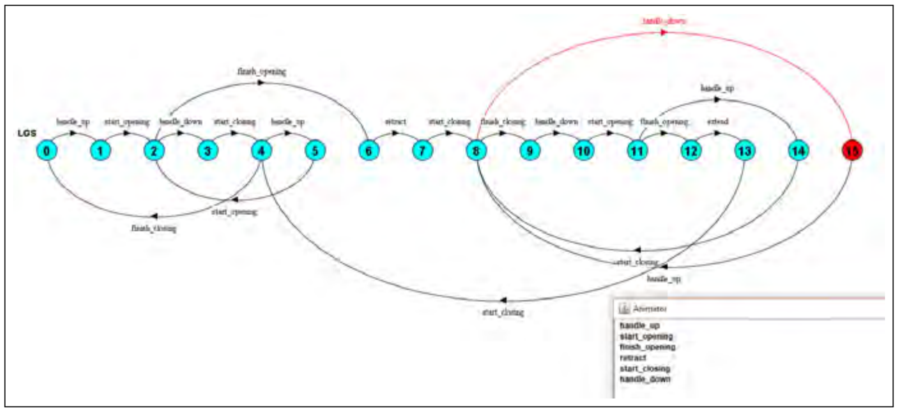
the LGS handle down, the controller should initiate the gear extension process. Regardless of the initial system's state, this process should end up correctly – so that in the end the gear is extended and the LGS latch is closed.

There exists, however, a state from which the erroneous ASM specification will not bring the system to the correct configuration. This state corresponds to a situation in which the gear has just been retracted, the door is closing, and the crew decides to cancel retraction by pushing the handle down. A correctly working system would cancel the retraction sequence and initiate gear extension. State 15 on Figure 11.2 illustrates this situation: the *start_opening* outgoing action cancels the door closing process initiated by action *start_closing* back in state 7. The state machine proceeds with the gear extension procedure. The erroneous ASM specification models a system that waits for the crew to pull the handle up again to let the system complete the gear retraction process. State 15 on Figure 11.3 features only one outgoing transition: pulling the handle up again. Instead of canceling the door closing process (Figure 11.2), the system starts waiting for the crew to pull the handle up. Imagine a situation in which the crew tries to retract the gear during take-off, and some physical obstacle prevents the latch from closing completely. In this case a possible solution might be to extend the gear back, and then try to retract it again. A real controller implemented around the erroneous specification would make extension with the latch partially closed impossible.

The published Eiffel translation of the specification does not have the error. To catch it with the AutoReq method one needs first to introduce the error back by commenting out two lines in the `open_door` routine of the Eiffel translation:

```
when closing_state then
door_status := opening_state
```

and then submit routine `extension_duration` to the AutoProof tool; the verification will fail. The "README" file in the accompanying GitHub repository [Nau17] provides detailed instructions on submitting AutoReq requirements to AutoProof. Internally,



handle_up
start_opening
finish_opening
retract
start_closing
handle_down

Figure 11.3: The erroneous LGS state machine. Pushing the handle down fails to cancel the gear retraction process. It puts the system to waiting for the crew to pull the handle up again. The bottom-right box contains the trace leading to state 15.

AutoProof transforms the Eiffel routine to Boogie code and submits it to the Boogie executable [Bar+05]. The Boogie executable converts its input to first-order logic formulae and submits them to the Z3 SMT solver [MB08].

AutoProof detects the error in the following major steps:

1. *Inline* the unqualified calls inside of the `extension_duration` routine to the level of attribute updates and **`assume`** statements.
2. *Unroll* the loop inside of `extension_duration`. How much to unroll is a configurable setting; the default configuration suffices for the LGS example.
3. *Check* the **`assert`** statements based on the outcome from the *Inline* step.

The intent of applying AutoReq to this example was not to look for errors but to try out the approach, illustrate it on a widely used problem, and compare it with other treatments of that problem. No error had been reported and we did not expect to find one. To ascertain its presence, we contacted one of the authors of the original article describing the ASM implementation. He confirmed the presence of the error in the paper. (He also noted that the private repository used by his colleagues and him had a correct specification.)

11.7 Related work

11.7.1 Similar studies

The ASM treatment of the LGS example comes from a collection including other treatments [BW14], such as Event-B [SA17], [Lad+17], [ML17], Fiacre [BDZF14] and Hybrid Event-B [Ban17]. The original collection [BW14] discusses pros and cons of these approaches, and we do not repeat that discussion. AutoReq complements these approaches with the following:

- Language reuse: AutoReq captures temporal and timing properties in a general purpose programming language. This will inevitably save resources for software teams that want to apply formal methods.
- Technology reuse: AutoReq relies on AutoProof, a Hoare logic based program prover. The original use case of AutoProof was specifying and verifying programs according to the principles of Design by Contract. With AutoReq, software teams can use the tool throughout the whole software lifecycle, starting from the requirements phase.
- Specification reuse: AutoReq makes it possible to avoid copying-and-pasting already stated assertions through the standard routine call mechanism, familiar to any post-Assembly programmer.
- Implementation reuse: AutoReq does not require translating programs to models and back for further formal verification. If a change in the program breaks an AutoReq requirement, the prover will immediately notice this.

These advantages need stronger support in the form of successful industrial applications of AutoReq. Such applications may also uncover additional problems to solve. The application of AutoReq to the LGS example inherits the questionable assumptions (Section 11.3) from the original work by Arcaini et al. Applying AutoReq to an example with weaker assumptions would provide more evidence of its benefits.

The applicability studies will follow the LGS-based experiment that focuses on illustrating the approach alone. Combining the first description of AutoReq with its applicability studies would bear the risk of making the chapter difficult to read.

11.7.2 Existing formalisms

Reasoning about programs, imperative and concurrent, has been the focus of computer science researchers for decades [Jon03], and it traces back as early as Turing's work [Jon17]. Different techniques have been developed over time, and it soon became clear that, while *post facto* verification can be successful for small programs, an effective verification strategy should support and be part of the software development itself and be fully embedded in the process.

The AutoReq method follows this idea and relies on DbC verification; however, one should understand that DbC is not well suited for control software as it is. The possibility of unexpected changes in the values of environment-controlled variables introduces the gap between DbC and control software. Traditional DbC relies on invariant-based reasoning, on the principle of invariant stability [Pol+14]: it should be impossible for an operation to make an object inconsistent without modifying the object. This principle does not work with control software because of the unpredictable environment-controlled variables. In other words, any attempt to constrain an environment-controlled variable through a contract will inevitably lead to the contract's failure.

Control software communicates asynchronously with the environment. This introduces another gap with DbC, which is designed from the beginning to deal with synchronous software. For non-life-critical systems [JZ95] one may sacrifice the asynchrony under additional assumptions (Section 7.5.2), but the Landing Gear System does not fall into this category.

An interesting technique for including environment properties is the notion of monitor introduced by Zave [Zav82]. A monitor is an executable requirement that runs in a dedicated process and observes the system from outside logging possible anomalies. A monitor continuously polls the state of nondeterministic variables and checks if the system evolves accordingly. This is, however, a run-time mechanism; with AutoReq, we seek requirements techniques that lend themselves to static verification.

The general aspiration towards sound static verification resulted in numerous modeling approaches that rely on a declarative logic. Alloy [Jac06] is one of these declarative modeling languages, based on first-order logic, that are used to express complex behavior of software systems. Alloy is a successor of Z [ASM80] with its own formal syntax and semantics, that adds automatic verification and tool support to Z specifications. A model created in Alloy can indeed be automatically checked for correctness by using a dedicated tool: the *Alloy Analyzer*, a SAT-based constraint solver that provides fully automatic simulation and checking. Alloy is one of the tools used for *requirements verification*. There are several examples of successful applications of the modeling languages in different fields: from pedagogical to enterprise modeling to transportation. A list documenting some of these applications can be found in [Jac17].

The declarative view simplifies static reasoning, but the system will eventually have to physically operate. C. A. R. Hoare introduced an imperative logic to statically reason about software way back in 1969. This invention has been treated as a verification mechanism. We are interested in requirements specification notations. The notion of seamless requirements (Chapter 10) uses generalized Hoare triples, specification drivers (Chapter 8), as a requirements notation.

The AutoReq method steps forward by applying the idea of seamless requirements to the nondeterministic setting. It empowers the operational view of Pamela Zave on requirements with AutoProof – a Hoare logic based prover of Eiffel programs with contracts that relies on the Boogie technology [Bar+05]. In AutoReq a requirement is a routine enriched with `assume` statements capturing environment assumptions and `assert` statements that capture the obligations for AutoProof corresponding to the assumptions. The resulting method respects environment-controlled phenomena as monitors do but does not assume the requirements to physically run. The AutoReq method will benefit the development process even when there is no static prover like AutoProof: an operational requirement will become a subject to testing as a parameterized unit test (PUT) [TS05]. The testing will consist in this case of running the requirement in the simulated environment described in its `assume` statements.

11.7.3 Timing properties

Representation of real-time requirements, expressed in general or specific form, is a challenging task that has been attacked through several formalisms both in sequential and concurrent settings, and in a broad set of application domains. The difficulty (or impossibility) to fully represent general real-time requirements other than in natural language or making use of excessively complicated formalisms (unsuitable for software developers), has been recognized.

In [MB10] the domain of real-time reconfiguration of systems is discussed, emphasizing the necessity of adequate formalisms. The problem of modeling real time in the context of services orchestration in Business Process, and in presence of abnormal behavior has been examined in [Maz05] and [FBM14] by means, respectively, of process algebra and temporal logic. Modeling protocols also requires real-time aspects to be represented [BH00]. Event-B has also been used as a vector for real-time extension [Ili+12] to handle control software requirements.

In all these studies, the necessity emerged of focusing on specific typology of requirements using ad-hoc formalisms and techniques and making use of abstractions. The notion of *real-time* is often abstracted as *number of steps*, a metric commonly used.

The AutoReq method works with the explicit notion of time distance between events by stating operational assumptions on the environment; it also supports the abstraction of time as number of steps through finite loops with integer counters.

11.8 Summary

The AutoReq approach presented above is a comprehensive method for requirements analysis based on ideas from modern object-oriented software engineering and the application of a seamless software process that relies on the notation of a programming language as a modeling tool throughout the software process. AutoReq also clarifies the notion of verifying requirements and shows how to use a program prover to perform the verification. In addition, it connects fundamental concepts, heretofore considered independent, from two different areas of research: verification (`assume`/`assert`) and requirements (*environment*/*machine*).

AutoReq has the following limitations, also suggesting areas of improvement:

- While the idea of seamless requirements has been widely applied, its AutoReq development as described here needs more validation on diverse examples in an industrial setting, with actual stakeholders involved.

- The patterns given are not necessarily complete; here too experience with more examples is necessary to determine if there is a need for other patterns.

- The idea of using a programming language for requirements runs counter to accepted ideas; while there are strong arguments supporting it, and ample discussions in some of the OO literature, some people may still hesitate to adopt it.

- More work is required to determine how applicable AutoReq would be to a software process relying on technologies other than Eiffel and AutoProof. In line with this goal, we applied AutoReq [Gal18] to the London Ambulance System case [Alr+13], [Let01] and continue working on other examples.

- As discussed in Section 11.5, parts of the process may benefit from more automation. Such further tool support is currently under development.

With these reservations, we believe that AutoReq and the associated case study demonstrate the benefits and contributions listed in the introduction and point to a promising approach to producing and verifying effective requirements for control software.

Acknowledgment

We are indebted to the authors of the ASM version of the LGS case study [AGR17] for their careful work on this problem. We are particularly grateful to Professor Angelo Gargantini for his openness, patience and insights in discussing the ASM work with us.

Chapter 12

Making Seamlessness Reusable

Insufficient requirements reusability, understandability and verifiability jeopardize software projects. Empirical studies show little success in improving these qualities separately. Applying object-oriented thinking to requirements leads to their unified treatment. An online library of reusable requirement templates implements recurring requirement structures, offering a starting point for practicing the unified approach.

12.1 Introduction

The industry is not actively applying requirements reuse [PQF17], which is regrettable: it might help, if practiced, not only to save resources in the requirements specification phase, but also to obtain documents of better quality both in content and syntax. It might also decrease the risk of writing low quality requirements and lead to the reuse of design, code, and tests.

Bertrand Meyer in 1985 described seven understandability problems common to natural language specifications [Mey85] and proposed the process of passing them through a formal notation to produce their more understandable versions. He has later given a name to the approach – "The Formal Picnic Approach"[1]. Formal picnics should be practiced more actively and should be reusable across projects.

The general problem of reuse finds itself in requirements' verifiability too. Requirements' verifiable semantics follows several recurring patterns in most of the cases [DAC99]. If a pattern exists, it should be reused, and to be reused it should be encoded as a template. The template should also be connected to the main instruments of software verification – tests and contracts.

Applying object-oriented thinking to the problems of requirements reusability, understandability and verifiability draws a new roadmap towards addressing them simultaneously. A reusable library of requirement templates, taking the familiar form of object-oriented classes, provides a starting point for practicing the approach. Each template encodes a formal semantics pattern [DAC99] as a generic class reusable across

[1]`https://tinyurl.com/ycn526rm`

projects and components, for verifying candidate solutions through either testing or program proving.

12.2 The problem explained

Chapter 2 introduces some problems with reusability, understandability and verifiability of requirements. The present section refines these problems further. The preceding chapters of the dissertation target individual qualities of requirements. The discussion that follows the section explains how to address these concerns within a single requirements process at once.

12.2.1 Reusability

Reusability has become a success story in the reuse of code [Zai+15] and tests [TS05], but not requirements. On that side too, many patterns recur again and again, causing undue repetition of effort and mistakes. The practice of industrial projects, however, involves little reuse of requirements. Textual copy and subsequent modification of requirements from previous projects are still the most commonly used requirements reuse techniques [PQF17], which has already been long recognized as deficient in the world of code reuse.

The most critical factors inhibiting the industrial adoption of requirements reuse through software requirement patterns (SRP) catalogues are [PQF17]:

- The lack of a well-defined reuse method.
- The lack of quality and incompleteness of requirements to reuse.
- The lack of convenient tools and access facilities with suitable requirements classification.

Scientific literature studying requirements reuse approaches pays little attention to these factors when measuring the studied approaches [IPP18]. The degree of reuse is the most frequently measured variable, but it is measured under the assumption that the evaluated approach is fully practiced. This assumption does not meet the reality: most of the practitioners who declare to practice requirements reuse approaches, apply them very selectively [PQF17]. Secondary studies, which study other studies, equally ignore the factors that matter to practitioners [IPP18].

12.2.2 Understandability

Bertrand Meyer, in his work “On Formalism in Specifications”[Mey85], described “the seven sins of the specifier” – a classification of the frequently recurring flaws in requirements specifications. Analyzing a specification of a well-known text-processing problem illustrated that even a small and carefully written natural language requirements document may suffer from the following problems:

- *Noise* – the presence in the text of an element that does not carry information relevant to any feature of the problem. Variants: redundancy; remorse.
- *Silence* – the existence of a feature of the problem that is not covered by any element of the text.
- *Overspecification* – the presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
- *Contradiction* – the presence in the text of two or more elements that define a feature of the system in an incompatible way.
- *Ambiguity* – the presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.
- *Forward reference* – the presence in the text of an element that uses features of the problem not defined until later in the text.
- *Wishful thinking* – the presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

Identified in the times when software processes were following the Waterfall model, which takes good care of every software development lifecycle phase, these problems remain. Nowadays processes pursue continuity, and requirements analysts have little time to process new requirements before passing them to the developers. The processes are iterative and collecting requirements for another iteration often starts before the current iteration finishes. The pace of work lowers availability of expert developers for evaluating the new requirements' verifiability. The pervasiveness of Internet technologies like Google Search brings problems too. Many sources of unclear origins now offer tons of potentially unchecked information, which is sometimes overly trusted.

Denying the progress makes no sense, however. Requirements engineering tools should help the practitioners to improve the quality of information they consume and rely on. The improved information should be reusable across projects.

12.2.3 Verifiability

The reusability concern applies to requirements' verifiability as well. Dwyer et al. analyzed 555 specifications for finite-state verification from different domains and successfully matched 511 of them against 23 known patterns [DAC99]. The patterns were encoded in modeling notations without a guidance on how to reuse them across projects for verifying candidate solutions. The gap still exists, and the state-of-the-practice [PQF17] and literature reviews [IPP18] of requirements reuse approaches, as well as the studies they cite, do not evaluate requirements' verifiability in the studied approaches.

Requirements reuse approaches should properly address the verifiability aspect: reusing non-verifiable requirements makes little sense. The approaches should make it clear how to capture and reuse recurring verifiable semantics' structures.

12.3 Running example

Wikipedia represents a notable example of an intensely used and trusted Internet resource. The rest of the discussion relies on a Wikipedia page describing a 24-hour clock[2] as a requirements document example. The "24-hour clock" document is prone to the seven requirements understandability problems [Mey85]. It only has few statements relevant to clock behavior:

1. The 24-hour clock is a way of telling the time in which the day runs from midnight to midnight and is divided into 24 hours, numbered from 0 to 24.

2. A time in the 24-hour clock is written in the form hours:minutes (for example, 01:23), or hours:minutes:seconds (01:23:45).

3. Numbers under 10 usually have a zero in front (called a leading zero); e.g. 09:07.

4. Under the 24-hour clock system, the day begins at midnight, 00:00, and the last minute of the day begins at 23:59 and ends at 24:00, which is identical to 00:00 of the following day.

5. 12:00 can only be mid-day.

6. Midnight is called 24:00 and is used to mean the end of the day and 00:00 is used to mean the beginning of the day.

The rest of the text is *noise*. The "or" connective in Statement 2 results in *wishful thinking*: is it acceptable to decide between the two options for every clock object, or should the decision be taken once and uniformly applied to all objects? None of the requirements after Statement 2 talk about seconds, from which it follows that the author silently made the choice in favor of the "hours:minutes" format. This "sin" falls into the *silence* category. The "usually" qualification introduces the *wishful thinking* problem to Statement 3: how are the developers expected to check candidate solutions against this requirement? Statements 4 and 6 result in a *contradiction* each other: statement 4 says that midnight is 00:00, while statement 6 defines *24:00* as midnight and *00:00* as the beginning of the day. The contradiction may arise as a result of *forward referencing*: *24:00* and *00:00* are only defined in 6, while first used in 1 and 4. The last part of Statement 4 is a *remorse*: the author implicitly admits that the first part of the statement was not enough and adds the "which is…" part. Statement 5 introduces an *ambiguity*, since the document never defines the "mid-day". Moreover, terms like "mid-day", "midnight", "afternoon" should be defined through specific clock states; it is not clear then what the author means by saying that a specific state can only be mid-day/midnight/afternoon: it can be whatever, depending on the terminology.

The illustration of the object-oriented requirements approach handles a fragment of Statement 1: "the day runs from midnight to midnight", referred to as "Statement 1.1". Understanding this requirement's treatment will suffice to understand the approach. A GitHub repository [3] hosts the complete treatment of the "24-hour clock" example.

[2] https://tinyurl.com/ybocy485

[3] https://tinyurl.com/y6w7nlcs

12.4 Reuse methodology

Requirements reuse methodologies are bidimensional [IPP18]. The first dimension, known as *development for reuse*, describes the procedure of identifying and capturing new requirement patterns. The second dimension, known as *development with reuse*, describes the process of searching and reusing the captured patterns for specifying new requirements with lower efforts as compared to specifying them without the patterns.

12.4.1 Development for reuse

Given a collection of requirements:

1. Perform the standard commonality and variability analysis on the collection.
2. Capture the identified commonality in an object-oriented class.
3. Capture the semantical commonality through a specification driver (Chapter 8) to support verification.
4. Capture the structural commonality through a string function to support formal picnics.
5. Parameterize the identified variability points through abstraction and genericity.

12.4.2 Development with reuse

Given an informal requirement:

1. Analyze the requirement's meaning and structure.
2. Find the most appropriate requirement template class through the IDE's search facilities.
3. Inherit from the found template in a new class representing the requirement.
4. Refine the abstractions into domain definitions.
5. Replace the genericity with the specified types and domain definitions.
6. Perform a formal picnic to see if the new string representation of the requirement has a different meaning from the original one.
7. Verify candidate solutions through running [TS05] or proving (Section 8.4) the contracted routine.

12.5 Technical artifacts

Two major technical contributions support the method.

12.5.1 Library of templates

A ready-to-use online Eiffel library[4] of template classes captures known requirement patterns [DAC99]. The library represents a result of applying the *development for reuse* process to the patterns and provides basis for *development with reuse*. The library is written in Eiffel for readability, but the method scales to other object-oriented languages with support for genericity.

12.5.2 Library of multirequirement patterns

An online OneNote notebook [5] rearranges the original collection of patterns [6] in the form of multirequirements [Mey13] to support their understanding. Dwyer et al. have initially developed the patterns in 5 notations: LTL, CTL, GIL, Inca, QRE. Their online collection consists of 5 large pages corresponding to these notations. The alternative collection consists of 23 pages making it possible to study individual patterns in all the 5 notations simultaneously. The representations are clickable and lead to their sources in the original repository developed by Dwyer et al. Each page includes a link leading to the corresponding template in the online Eiffel library.

12.6 Applying a template

The following illustration handles the "Statement 1.1" requirement by applying a reusable template class from the Eiffel library. The requirement fits into the "Global Response" pattern [DAC99]. The pattern reads: "S responds to P globally", for events S and P. It is the most frequently used pattern: out of the 555 analyzed requirements [DAC99], 241 represented this pattern. For "Statement 1.1", both S and P map to the midnight event: "midnight responds to midnight globally". This new statement paraphrases the original one, "the day runs from midnight to midnight".

Class STATEMENT_1_1 (Figure 12.1(a)) captures the requirement. The class inherits from:

- A generic application of class RESPONSE_GLOBAL to classes CLOCK and MIDNIGHT, where RESPONSE_GLOBAL is a generic template encoding the "Global Response" pattern (Appendix A.20). The RESPONSE_GLOBAL [CLOCK, MIDNIGHT, MIDNIGHT] application reads: "for type CLOCK, MIDNIGHT response to MIDNIGHT globally".

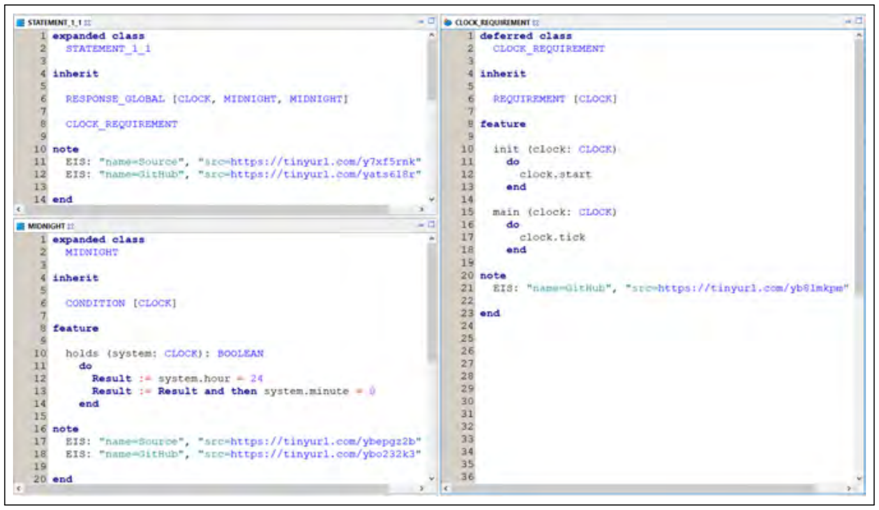
- Class CLOCK_REQUIREMENT recording domain information common to all clock requirements: the fact that the tick routine advances a clock's state, and the start routine initializes a new clock.

The CLOCK class is a candidate solution implementing the "clock" concept, and the MIDNIGHT class captures the definition of midnight through effecting the deferred holds Boolean function inherited from generic class CONDITION applied to the CLOCK class. The

[4]https://tinyurl.com/ybd4b5un

[5]https://1drv.ms/u/s!AsXOYPvbmuEyh4IsDdYj-i6V5yX0OA

[6]http://patterns.projects.cs.ksu.edu

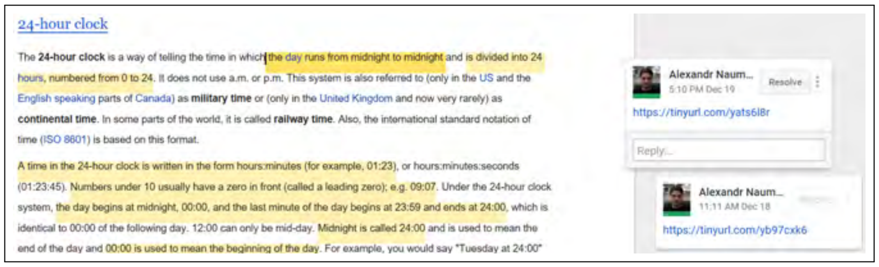


```
expanded class
  STATEMENT_1_1

inherit

  RESPONSE_GLOBAL [CLOCK, MIDNIGHT, MIDNIGHT]

  CLOCK_REQUIREMENT

note
  EIS: "name=Source", "src=https://tinyurl.com/y7xf5rnk"
  EIS: "name=GitHub", "src=https://tinyurl.com/yats618r"

end
```

```
expanded class
  MIDNIGHT

inherit

  CONDITION [CLOCK]

feature

  holds (system: CLOCK): BOOLEAN
    do
      Result := system.hour = 24
      Result := Result and then system.minute = 0
    end

note
  EIS: "name=Source", "src=https://tinyurl.com/ybepgz2b"
  EIS: "name=GitHub", "src=https://tinyurl.com/ybo232k3"

end
```

```
deferred class
  CLOCK_REQUIREMENT

inherit

  REQUIREMENT [CLOCK]

feature

  init (clock: CLOCK)
    do
      clock.start
    end

  main (clock: CLOCK)
    do
      clock.tick
    end

note
  EIS: "name=GitHub", "src=https://tinyurl.com/yb81mkpm"

end
```

(a) EiffelStudio with the STATEMENT_1_1 class representing the "Statement 1.1" requirement.



24-hour clock

The **24-hour clock** is a way of telling the time in which the day runs from midnight to midnight and is divided into 24 hours, numbered from 0 to 24. It does not use a.m. or p.m. This system is also referred to (only in the US and the English speaking parts of Canada) as **military time** or (only in the United Kingdom and now very rarely) as **continental time**. In some parts of the world, it is called **railway time**. Also, the international standard notation of time (ISO 8601) is based on this format.

A time in the 24-hour clock is written in the form hours:minutes (for example, 01:23), or hours:minutes:seconds (01:23:45). Numbers under 10 usually have a zero in front (called a leading zero); e.g. 09:07. Under the 24-hour clock system, the day begins at midnight, 00:00, and the last minute of the day begins at 23:59 and ends at 24:00, which is identical to 00:00 of the following day. 12:00 can only be mid-day. Midnight is called 24:00 and is used to mean the end of the day and 00:00 is used to mean the beginning of the day. For example, you would say "Tuesday at 24:00"

Alexandr Naum...
5:10 PM Dec 19
https://tinyurl.com/yats6l8r

Alexandr Naum...
11:11 AM Dec 18
https://tinyurl.com/yb97cxk6

(b) Google document with the contents of the "24-hour clock" Wikipedia page.

Figure 12.1: Requirement classes in EiffelStudio (Figure 12.1(a)), and the contents of the "24-hour clock" Wikipedia page copied to a Google document (Figure 12.1(b)). The "Source" link in the STATEMENT_1_1 class leads to the corresponding commented fragment in the Google document. The comment contains the GitHub location of the fragment's object-oriented version, equal to the location in the "GitHub" EIS link in STATEMENT_1_1.

generic application emphasizes the fact that the notion of midnight applies to the notion of clock.

The classes have something in common: the "note" section at the bottom with Web links of two kinds. Links named "Source", when followed, highlight the fragments in the original requirements documents from which the enclosing requirement classes were derived. Links named "GitHub", when followed, lead to the enclosing classes' locations on GitHub. The "Source" link in `STATEMENT_1_1`, for example, highlights, when followed, the "the day runs from midnight to midnight" phrase in the Google document[7], and brings the comment on this phrase to the reader's attention (Figure 12.1(b)). The comment contains the GitHub link leading back to the `STATEMENT_1_1` class on GitHub; this link is identical to the "GitHub" link in the `STATEMENT_1_1` class' "note" section.

12.7 Formal picnic

The `RESPONSE_GLOBAL` class (Figure 12.2) implements its string representation through redefining the standard `out` function present in all Eiffel classes. Any instruction that expects a string argument, such as `print`, automatically invokes this function to get the argument's string representation if the argument has a non-string type.

Routine `run` of class `TESTER` (Section 12.7) is a configurable entry point of the console application illustrating formal picnics and verification of object-oriented requirements.

Line 11 of `TESTER` outputs the structured string representation of the `STATEMENT_1_1` object-oriented requirement. The `.default` expression returns the default object of the `STATEMENT_1_1` class, and the `print` instruction puts the object's string representation to the "Output" window below the "TESTER" window. The requirement's name, `STATEMENT_1_1`, goes before the colon and its string representation goes after.

The requirements analyst now has two comparable string representations of the requirement: the original and the generated one. Comparing them facilitates analysis and may result in asking clarifying questions to the customer and in additional communication.

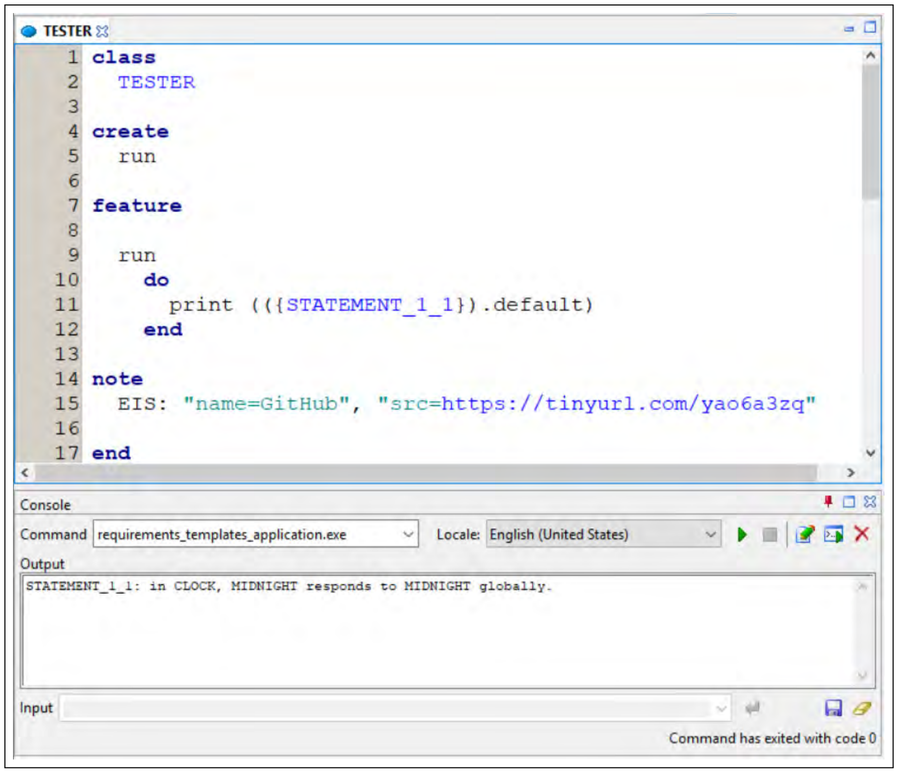
12.8 Verification

The template classes (Appendix A), including `RESPONSE_GLOBAL` (Appendix A.20), contain instruments of their own verification in the form of a contracted routine called `verify`. The `run` routine of the `TESTER` class may call `verify` to test a candidate solution.

Line 15 of the `TESTER` class (Figure 12.3) tests class `CLOCK` as a candidate solution of the `STATEMENT_1_1` requirement. Line 13 instantiates a `CLOCK` variable, while lines 14 and 15 use the variable as test input. The following discussion explains the nature of line 14. The line is commented to illustrate the problem that the line fixes when uncommented.

The `verify` routine has a precondition. For the `STATEMENT_1_1` class, the precondition becomes the `holds` Boolean function from the `MIDNIGHT` class. This function re-

[7]`https://tinyurl.com/y96rj2v3`



TESTER

```
class
  TESTER

create
  run

feature

  run
    do
      print (({STATEMENT_1_1}).default)
    end

note
  EIS: "name=GitHub", "src=https://tinyurl.com/yao6a3zq"

end
```

Console

Command: requirements_templates_application.exe   Locale: English (United States)

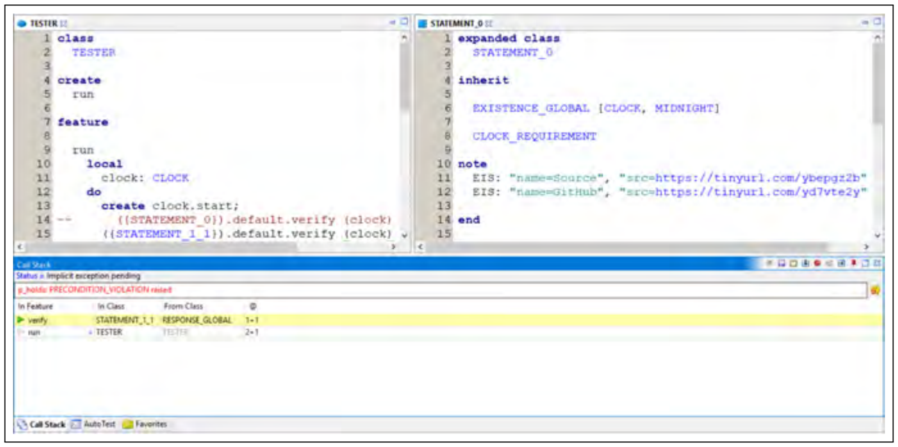
Output

```
STATEMENT_1_1: in CLOCK, MIDNIGHT responds to MIDNIGHT globally.
```

Input

Command has exited with code 0

Figure 12.2: The executable code (the upper window) outputs the automaticaly generated string representation of the requirement to the console (the lower window).



TESTER

```
class
  TESTER

create
  run

feature

  run
    local
      clock: CLOCK
    do
      create clock.start;
--      ({STATEMENT_0}).default.verify (clock)
      ({STATEMENT_1_1}).default.verify (clock)
```

STATEMENT_0

```
expanded class
  STATEMENT_0

inherit

  EXISTENCE_GLOBAL [CLOCK, MIDNIGHT]

  CLOCK_REQUIREMENT

note
  EIS: "name=Source", "src=https://tinyurl.com/ybepgz2b"
  EIS: "name=GitHub", "src=https://tinyurl.com/yd7vte2y"

end
```

Call Stack

Status = Implicit exception pending

p_holds: PRECONDITION_VIOLATION raised

In Feature	In Class	From Class	@
verify	STATEMENT_1_1	RESPONSE_GLOBAL	1-1
run	TESTER	TESTER	2-1

Figure 12.3: An exception caused by violating the requirement's verification precondition.

turns `True` only for the *24:00* time, and the newly instantiated `clock` variable is set to time *00:00*. Line 14 fixes this mismatch, and its removal crashes the execution. The "Call Stack" window provides information related to the failure: a precondition tagged `p_holds` is violated in `STATEMENT_1_1`, inherited from the `RESPONSE_GLOBAL` template class (Appendix A.20). The testing code should set the `clock` variable's state to time *24:00* before testing `STATEMENT_1_1`; line 14 does exactly this. `STATEMENT_0` is a requirement class saying that the midnight state should be in principle achievable by `CLOCK`. The `EXISTENCE_GLOBAL` pattern [DAC99] captures this semantics. Line 14 tests `CLOCK` against `STATEMENT_0` by trying to reach the midnight state on the input variable. Uncommenting the line will remove the precondition violation.

The process of deriving `STATEMENT_0` is an example of how the verification process may help identify a new requirement and learn a new template.

Program proving and Design by Contract may be used instead of testing. The automatic prover (AutoProof [Tsc+15] in the context of Eiffel) should be applied to the requirements classes, `STATEMENT_0` and `STATEMENT_1_1`. The prover will statically check the contracted `verify` routine according to the principles of Hoare logic [Hoa69]. The prover will only accept the routine if the `CLOCK` class has a strong enough and correct contract (Section 8.4.4). The illustration relies on testing because AutoProof, in its current state, requires a lot of additional annotations to check classes like `STATEMENT_1_1`, and explaining these annotations goes beyond the object-oriented requirements idea's essentials.

12.9 Summary

The approach helps to fix the identified problems undermining the lack of requirements reuse:

- *The lack of a well-defined reuse method*: the reuse method is object-oriented software construction, which is a well-defined method.
- *The lack of quality and incompleteness of requirements to reuse*: the templates library implements the existing collection of specification patterns proven to cover most of the cases, which makes the library complete and quality in that sense.
- *The lack of convenient tools and access facilities with suitable requirements classification*: the tools and access facilities are object-oriented IDEs and GitHub, with all their powerful features. The classification is that of the Dwyer et al.'s collection, proven to be practically relevant.

The approach helps to fix the requirements understandability problems:

- *Noise*: only those requirements remain that fall into an existing verifiable requirement template.
- *Silence*: an attempt to verify existing object-oriented requirements may uncover missing requirements, as it was the case with `STATEMENT_0`.

- *Overspecification*: only those requirements remain that fall into an existing verifiable requirement template. Implementation details cannot map to a requirement template.
- *Contradiction*: one notion may be defined in only one way, otherwise the IDE will raise a compilation error. The contradiction caused by two inconsistent definitions of midnight was resolved by defining this notion in the form of the `MIDNIGHT` class.
- *Ambiguity*: little can be done to remove the possibility for different interpretations – the requirements interpretation process is performed by a cognitive agent anyway. If an interpretation is identified as erroneous, however, switching to another template will automatically update both the generated string representation and the underlying verifiable semantics. In other words, the templates may help to reduce the effort spent on fixing the consequences of the misinterpretation.
- *Forward reference*: the approach removes this problem. There is no notion of requirements' order in the object-oriented approach, and meaningful statements are connected by the standard "client-supplier" relationship, extensively supported by the object-oriented IDEs.
- *Wishful thinking*: only those requirements remain that fall into an existing verifiable requirement template. The compiler will not accept a template's application in which the verifiable semantics is not fully defined.

The approach helps to fix the requirements verifiability problem. The library of Eiffel classes fixes the lack of reusable templates covering the identified verifiable specification patterns. The approach makes it possible to capture and reuse newly identified patterns using the existing object-oriented techniques complemented with contracts.

Besides the benefits, the approach has some limitations:

- Requirements analysts' familiarity with the principles of object-oriented analysis and design.
- Software developers' familiarity with the principles of Hoare logic based reasoning.

Intelligent tools should be embedded into existing text editors for:

- Detecting known patterns in what requirements analysts specify manually.
- Proposing reusable templates corresponding to the identified patterns.
- Identifying new patterns in requirements that do not map to existing patterns.

Natural language processing (NLP) would be an appropriate instrument for implementing these tools [Dal+18].

Part III

Discussion

Chapter 13

Qualitative Evaluation

The present chapter discusses qualitative arguments supporting the claim that the SOOR approach improves requirements' expressiveness, verifiability, reusability and understandability.

13.1 Expressiveness

SOORs can capture requirements of the following kinds:

- *ADT axioms*: specification drivers, which capture SOORs' formal semantics, inherit their syntax from the PUT-like tests: they are routines equipped with pre- and postconditions, and prover specific annotations [Pol+14]. The routines' implementations may contain as many command calls as necessary, which makes them suitable for capturing multicommand requirements.

- *Temporal properties*: specification drivers containing contracted loops in their implementations capture temporal properties. Loops' contracts consist of loop variants and invariants – constructs not present in most programming languages. According to Wikipedia, only Eiffel and Wiley programming languages have native support for loop invariants, and only Eiffel – for loop variants. PUT-like approaches emerged in the world of more widespread programming languages like Java and C# lacking support for loop contracts. This may explain the lack of support for temporal properties in these approaches.

- *Timing constraints*: specification drivers capturing temporal properties may capture timing constraints through the loop variants. A loop variant is a decreasing non-negative integer function. The loop variants' semantics maps to the notion of time, which monotonically goes in one direction. The rate at which a loop variant decreases corresponds to how the time flows in the problem space. The implementation of the loop variant may reflect the timing properties of the problem space.

13.2 Verifiability

The approach can verify what it can express: multicommand ADT axioms, temporal properties and timing constraints. Besides these, it helps remove the following requirements verifiability problems:

- *The modularity problem*: verification with SOORs does not assume instrumenting the verified code. Specification drivers, forming the verification core of SOORs, are clients of the verified components.
- *The lack of suitability for both testing and program proving*: specification drivers, unlike PUTs, are fully compatible with modular program proving after some tailoring consisting of adding some annotations [Pol+14] for AutoProof. The prover accepts a specification driver if the implementation contract is correct. AutoProof makes it possible for the two specification approaches to benefit each other; specification drivers, at the same time, remain applicable to the PUT-based testing. Well-definedness properties are specification drivers that call the same feature on two equal sets of inputs and assert preservation of the equivalence in their postconditions. AutoProof will only accept such specification drivers if the called feature's postcondition is well-defined. Contracts' inconsistency properties are specification drivers that assert False in their postconditions. The precondition and the implementation body depend on what is checked for an inconsistency – a feature's precondition, postcondition, or a class invariant. In any case, being able to prove False signals an inconsistency in the verified contract.
- *Lack of reusable templates covering the identified verification-oriented SRPs*: the online library of SOORTs captures exactly these SRPs.

13.3 Reusability

The approach helps fix the identified problems undermining requirements reuse:

- Copy and paste in requirements reuse: the SOOR approach builds on top of the object-oriented principles, which boosted software reuse and made the copy-and-paste approach a Stone Age practice. Applying object orientation to software requirements gives some hope for removing the copy-and-paste approach from requirements reuse.
- The lack of a well-defined reuse method: the reuse method is object-oriented software construction, which is a well-defined method.
- The lack of quality and incompleteness of requirements to reuse: the library of Eiffel SOORTs implements the existing SRP catalogues shown [DAC99], [KC05] to cover a significant portion of control software requirements, which makes the library complete and quality in that sense.
- The lack of convenient tools and access facilities with suitable requirements classification: the SOOR approach reuses the powerful tools and access facilities of

the object-oriented IDEs and GitHub. The requirements classification in the Eiffel library of SOORTs inherits the classification of the catalogue developed by Dwyer et al., proven to be practically relevant.

The two libraries of SOORTs implemented during the thesis work, have different contexts and levels of reuse:

- The control software SOORTs encode finalized behavior patterns; they are intended for being reused across projects, not for developing SOORTs. This maps well to the original SRP catalogues [DAC99], [KC05], in which the SRPs do not depend on each other.
- The software component SOORTs encode ADT specifications and may reuse each other, just like regular classes do in object-oriented programming.

13.4 Understandability

The formal picnic approach improves requirements' understandability [Mey85], [Mey18] by paraphrasing them by passing through an intermediate formal representation. SOORs provide a concrete tool support for this process in the form of functions that implement the paraphrasing parameterized with variable parts of requirement patterns.

Section 2.4 discusses how PUTs and multirequirements promote and inhibit the important characteristics of understandable requirements. Thought of as a combination of PUTs and multirequirements, SOORs inherit their best characteristics: implementation freedom, unambiguity, traceability, feasibility. Specification drivers, forming the semantical core of SOORs, additionally provide a formal framework for achieving completeness and detecting inconsistencies in the presence of a DbC-based program prover. We currently do not know how SOORs may affect the amount of noise in specifications.

13.5 Falsification experiment

The best way to test applicability of an approach is to apply it to unusual cases that were not considered from the beginning. One of the reviewers guessed that the approach applies only to atomic components, non-decomposable into finer-grained subcomponents:

> Reading the rest of the thesis and analyzing the examples, however, I have understood that, at least in its current form, the proposed approach does not aim at dealing with high level requirements expressed referring to a complex software made of many different components and subsystems acting as controllers. In fact, the approach appears to assume that there is a direct mapping between a SOOR and a single component (e.g., the bounded stack) which is either a control system, handling a finite and limited set of inputs, or a single data type (this is also mentioned at page 18 where it

is said that requirements to software components take the form of abstract data type, but no justification for this assumption is provided).

Indeed, individual SOORTs from the presented library apply only to one atomic component each. This does not necessarily imply, however, that the general approach does not scale to the multiple components case. Applying the approach to the following cases would help to test its applicability:

1. Software component composed of collaborating sub-components.
2. Software controller composed of collaborating sub-controllers.
3. Composition of software components collaborating with software controllers.

The section presents applying the SOOR approach to a software component composed of collaborating sub-components. The remaining two cases require more carefully designed experiments. Designing and conducting such experiments would be a perfect continuation of the present work.

Design patterns were the obvious choice for the case of a software component composed of collaborating sub-components. In this section we present a SOORT encoding the observer pattern as interpreted in an article by Polikarpova et al [Pol+14]. The listing below captures the corresponding SOORT.

```
note
  description: "Reusable abstract data type specification of the observer pattern."
  description: "Found in ‘‘Flexible Invariants Through Semantic Collaboration’’ by Polikarpova et al."
  EIS: "src= https://cseweb.ucsd.edu/~npolikarpova/publications/fm14.pdf"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y2x2xeat"

deferred class SUBJECT_OBSERVER [S, O, V]
  -- Types ‘‘S’’ and ‘‘O’’ form an observer pattern with shared state of type ‘‘V’’.

feature -- Deferred definitions.

  value (s: S): V
    deferred
    end

  subscribers (s: S): LIST [O]
    deferred
    end

  update (s: S; v: V)
    deferred
    end

  register (s: S; o: O)
    deferred
    end

  subject (o: O): S
    deferred
    end

  cache (o: O): V
```

```
    deferred
    end

  make (o: O; s: S)
    deferred
    end

  notify (o: O)
    deferred
    end

feature -- Abstract data type axioms.

  update_axiom (s: S; v: V; o: O)
    require
      subscribers (s).has (o)
      subject (o) = s
    do
      update (s, v)
    ensure
      subscribers (s).has (o)
      subject (o) = s
      value (s) = v
      cache (o) = v
    end

  register_axiom (s: S; o: O)
    require
      not subscribers (s).has (o)
      subject (o) = s
    do
      register (s, o)
    ensure
      subscribers (s).has (o)
      subject (o) = s
    end

  make_axiom (o: O; s: S)
    require
      not subscribers (s).has (o)
    do
      make (o, s)
    ensure
      subject (o) = s
      subscribers (s).has (o)
      cache (o) = value (s)
    end

  notify_axiom (o: O; s: S)
    require
      subscribers (s).has (o)
      subject (o) = s
    do
      notify (o)
    ensure
      subscribers (s).has (o)
      subject (o) = s
```

```
90        cache (o) = value (s)
91      end
92  end
```

Generic types S and O abstract the subject and observer concrete types. The V generic type abstracts the type of the state that the subject stores and broadcasts across its observers. Lines 12-42 declare the essential deferred features of the observer pattern. The specifier will need to implement these features in terms of the concrete types provided for S, O and V. Lines 46-91 capture the axioms of the observer pattern in the form of specification drivers.

The observer pattern consists of two equally important components – the subject and the list of observers. This plurality does not add, however, to the complexity of the method.

Chapter 14

Quantitative Evaluation

The present chapter discusses quantitative arguments showing that SOORs promote expressiveness, verifiability and reusability. We currently have no quantitative evidence for understandability; the benefits of SOORs for understandability may follow as self-evident, however, from the discussion in Chapter 13 and Chapter 2.

14.1 Expressiveness

The evidence of the SOORTs' expressiveness comes from the possibility to capture:

- The 23 temporal SRPs for control software [DAC99] (Section 4.1).
- The real-time semantics [KC05], as an optional feature inside the SOORTs for temporal properties.
- The 21 ADTs recurring in the requirements literature, some of which are essential components (Section 4.2).

Some of the control software SRPs have tricky formal semantics. For example, the "Bounded Existence Between Q and R" SRP, where the bound is at most 2 designated states, looks as follows in LTL:

$$\begin{aligned}\Box((Q \wedge \Diamond R) \Rightarrow ((\neg P \wedge \neg R)\mathcal{U}(R \vee ((P \wedge \neg R)\mathcal{U} \\ (R \vee ((\neg P \wedge \neg R)\mathcal{U}(R \vee ((P \wedge \neg R)\mathcal{U}(R \vee (\neg P\mathcal{U}R)))))\end{aligned} \tag{14.1}$$

We were able to encode this formula as a specification driver inside the `BOUNDED_EXISTENCE_BETWEEN` SOORT (Appendix A.6). Moreover, representing requirements' formal semantics as specification drivers allows us to generalize from the 2-states to the k-states case. Three out of the five notations used by Dwyer et al. – LTL, CTL and GIL – lack expressiveness for performing such generalization. Using the programming language as a requirements notation makes it possible to perform the generalization through enclosing the bounded existence semantics into an additional loop that runs exactly k times.

14.2 Verifiability

Modularity of the SOOR-based verification from its definitions, which is why it does not require evaluation.

Applicability of SOORs to both program proving and testing immediately follows from the definition as well. Specification drivers syntactically are PUTs equipped with the prover-specific annotations; the compiler ignores these annotations, which is why specification drivers may be used as PUTs without modification.

What may deserve an empirical evaluation is how useful specification drivers are for analyzing contracts' well-definedness and consistency. The EiffelBase2 library [PTF18] seems to be a perfect data set for such evaluation. We analyzed well-definedness of feature `copy_` in the EiffelBase2 classes. The feature copies the given object into the current one. Out of the 17 versions of the feature, 6 were underspecified. They come from the following classes:

- V_ARRAY2
- V_LINKED_QUEUE
- V_LINKED_STACK
- V_ARRAYED_LIST_ITERATOR
- V_ARRAY_ITERATOR
- V_HASH_SET_ITERATOR

Deeper analysis revealed that the most common problem was not taking into consideration the possibility of aliasing between the copied and the current objects. For the V_HASH_SET_ITERATOR class, however, AutoProof did not accept the well-definedness axiom even with the aliasing prohibited in the precondition. AutoProof did not terminate when checking the well-definedness axiom for the following 2 classes:

- V_DOUBLY_LINKED_LIST_ITERATOR
- V_LINKED_LIST_ITERATOR

The non-termination may be interpreted as if the features were underdefined. Summarizing the results of the analysis, out of the 17 versions AutoProof accepted the well-definedness axiom only for 9. Underdefined contracts may have security implications. Consider appending the following code to the implementation of feature `copy_` in class V_ARRAY2:

```
else
  array.wipe_out
  row_count := 0
  column_count := 0
end
```

The `else` clause describes the aliasing situation, which is ignored in the contract of the feature. The added code wipes out the current array's data. AutoProof accepts the modified implementation, which is not what we expect: a feature responsible for copying from another array should not erase the current one. The published presentation of EiffelBase2 claims well-definedness of the flawed classes [PTF18].

The EiffelBase2 library contains software components. As for control software, expressing their properties as specification drivers was also fruitful. Chapter 11 details uncovering an error in a published abstract state machine (ASM) implementation [AGR17] of the Landing Gear System (LGS) [BW14] – a commonly used example for evaluating applicability of formal specification and verification techniques.

14.3 Reusability

We might evaluate the extent to which the SOOR approach improves reusability by following the common approach – measuring the amount of duplication removed from requirements [IPP18]. Such evaluation would make little sense, however: the SOOR approach just applies the object-oriented principles to the construction of requirements. This makes the evaluation straightforward: the amount of duplication may be removed completely – this is exactly what happens to software built around the same principles. We prefer then to evaluate the extent to which the reuse approach simplifies specification of individual requirements.

Recall the "Bounded Existence Between Q and R" SRP (Equation (14.1)):

$$\begin{aligned}\Box((Q \wedge \Diamond R) \Rightarrow ((\neg P \wedge \neg R)\mathcal{U}(R \vee ((P \wedge \neg R)\mathcal{U} \\ (R \vee ((\neg P \wedge \neg R)\mathcal{U}(R \vee ((P \wedge \neg R)\mathcal{U}(R \vee (\neg P\mathcal{U}R)))))\end{aligned} \tag{14.2}$$

Repeatedly instantiating this SRP as it is and then translating it into unit tests may become challenging. In the SOOR approach, the complexity of specifying a SOOR does not depend on the SOORT's internal complexity. For example, a SOOR expressing requirement "equinox happens not more than two times during a year" for a calendar system would roughly look as follows:

```
class
  EQUINOX_FREQUENCY
inherit
  BOUNDED_EXISTENCE_BETWEEN [CALENDAR, EQUINOX, YEAR_BEGINNING, YEAR_END]
  CALENDAR _REQUIREMENT
end
```

where: class `CALENDAR` represents the specified type; `EQUINOX` captures the equinox condition; `YEAR_BEGINNING` and `YEAR_END` capture the beginning and the end of the year, respectively; `CALENDAR_REQUIREMENT` captures phenomena common to calendar requirements. Consider now requirement "the beginning of the year is always followed by the end of the year". This requirement represents the "Global Response" SRP, in LTL:

$$\Box(P \Rightarrow \Diamond S)$$

The complexity of this SRP is incomparably smaller than the complexity of the previous one. The SOOR capturing the new requirement would look as follows:

```
class
    YEAR_END_RESPONDS_TO_YEAR_BEGINNING
inherit
    RESPONSE_GLOBAL [CALENDAR, YEAR_END, YEAR_BEGINNING]
    CALENDAR _REQUIREMENT
end
```

This SOOR is simpler only in one way: it provides 3 actual generic parameters to its SOORT, while the previous one provides 4. We may say that the SOOR's complexity depends linearly on the number of formal generic parameters in the SOORT from which the SOOR inherits. For the existing control software SRP's, however, this number never exceeds 4.

As for specifying SOORs for software components from the ADT SOORTs: the number of ADT axioms depends quadratically on the number of operations in the specified ADT [Lam09]. Specifying a SOOR from an ADT SOORT requires only to connect the deferred features of the SOORT with the concrete features of the specified type. This does not remove the need to verify all the ADT axioms present in the SOORT in the form of specification drivers. Technologies like AutoProof and AutoTest solve the verification problem, however. The approach replaces the specification complexity from quadratic to linear.

14.4 Understandability

We currently have no empirical evidence that the SOOR approach improves requirements understandability: it requires feedback from people applying the SOOR approach. We consider industrial evaluation of this aspect as important part of the future work.

Chapter 15

Thesis Summary

The present chapter wraps-up the discussion by drawing conclusions, admitting some limitations and showing future research directions.

15.1 Conclusions

The dissertation presents Seamless Object-Oriented Requirements (SOOR) – a requirements approach that treats software requirements as regular input to the object-oriented analysis and design. The approach makes requirements full citizens of integrated development environments (IDEs) and removes the notational gap between requirements and their implementations. The object-oriented treatment makes requirements:

- *Expressive*, through the expressive power of Design by Contract (Chapter 7, Chapter 10).
- *Verifiable*, through specification drivers contained inside the requirement classes and contracts inside the implementation classes (Chapter 8, Chapter 9, Chapter 11).
- *Reusable*, through the standard object-oriented techniques – genericity and abstraction (Chapter 11, Chapter 12).
- *Understandable*, through the automatic paraphrasing mechanism embedded into requirement classes (Chapter 12).

Both qualitative (Chapter 13) and quantitative (Chapter 14) arguments show the improvements in expressiveness, verifiability and reusability, while understandability is currently lacking a supporting quantitative data. We expect to have such data soon, however, from applying the SOOR approach in an industrial setting.

The SOOR approach comes with a ready-to-use library of Seamless Object-Oriented Requirement Templates (SOORTs) – deferred generic requirement classes capturing known software requirement patterns (SRPs). Studying several SOORTs' internals will give intuition behind constructing new SOORTs. The thinking discipline behind

SOORTs' construction and reuse is identical to the object-oriented thinking, which decreases the learning curve for seasoned developers. In the SOOR approach, requirements become the junction point of the software process:

- The automatic paraphrasing implemented as part of the SOORTs supports requirements validation and understandability. A developer looking at a requirement and its paraphrased form may consider rewriting it.
- The SOORTs also contain a reusable requirements verification mechanism. This mechanism may either drive specification of strong enough and correct contracts (Chapter 8) or serve as input to a Parameterized Test-Driven Development (PTDD)-style [DTS10] construction process.
- The SOOR's verification mechanism makes it possible to instantly see if the existing solution correctly implements an added or updated requirement (Chapter 10).
- Symmetrically, the same verification mechanism makes it possible to instantly see if an updated solution correctly implements the existing requirements (Chapter 10).
- Analysis of a set of SOORs may reveal commonality among them, leading to creation of a new SOORT capturing the identified commonality. The original SOORs will become descendants of the new SOORT and will only include the variable part; as a consequence, they will look simpler.
- Seamless requirements may literally be programmed and reused for programming other requirements, as Chapter 11 shows.
- They may serve as building blocks for contracted implementations (Chapter 7) in the context of Design by Contract.

All these technical traits of Seamless Object-Oriented Requirements will bring the following sensible business effects:

- Better reusability of requirements.
- Higher responsiveness of the software process to changing requirements.
- Decreased learning curve for software developers.
- Higher confidentiality in requirements validity.
- Higher confidentiality in software correctness.
- Smaller number of software tools to buy and maintain.
- Possibility to formally prove software correct, even in the context of Agile development.
- Faster detection of defects in requirements.

> "Different tasks will of course remain. To take extreme examples, you are not doing the same thing when defining general properties of a system that has yet to be built and performing the last rounds of debugging. But the idea of seamlessness is to downplay differences where the traditional approach exaggerated them; to recognize, behind the technical variations, the fundamental unity of the software process. Throughout development the same issues arise, the same intellectual challenges must be addressed, the same structuring mechanisms are needed, the same forms of reasoning apply and, as shown in this book, the same notation can be used." [Mey97]

This citation from the OOSC book remains valid. The SOOR approach adapts the inital object-oriented software process to the everchanging nature of modern requirements. The approach treats requirements as early input to the object-oriented analysis process, resulting in the requirements enjoying the traditional benefits of object orientation and leading to higher levels of seamlessness.

15.2 Limitations

The SOOR approach has limitations. It applies at the conceptual level to any general-purpose programming language with "assume"/"assert" statements, as Chapter 11 demonstrates. Having a native support for contracts, however, would greatly simplify its application. Having a program prover like AutoProof would maximize outcome from practicing the approach, though it remains powerful even with testing-based verification. Successfully applying contracts and program proving will require some additional training; luckily, there are plenty of resources on these topics, from online tutorials [Lei13] to fundamental literature [Mey97].

The necessity to learn contracts and proof-based verification is better, in our opinion, than the existing necessity to learn separate requirements notations disjoint from the solution space. We justify this opinion as follows: the skills required for applying the SOOR approach may pay not only at the requirements but also at the construction. Developers that master contracts and program proving for requirements may start applying these techniques in their programming activities, which will result in better documented and verified programs.

We could not come up with any practical limitations for applying the SOOR approach. This should not surprise – the objective was exactly to remove the existing limitations. Industrial studies of the approach should be conducted, however, to reveal possible concerns from the practitioners. We leave this important task as the future work.

15.3 Future work

The following work is necessary to show stronger evidence of the SOOR's approach benefits:

- Applying and measuring the approach in an industrial setting.

- Proving formally that the presented library of seamless requirement templates correctly resembles the encoded SRPs' semantics.

The dissertation creates opens up the following research directions:

- Automatic generation of seamless requirement templates for a given programming language from a given pattern expressed in a mathematical formalism. The dissertation present an Eiffel library of seamless requirement templates that. The library encodes requirement patterns from an existing catalogue. The input patterns are encoded in several mathematical formalisms. Because the concepts behind the templates' construction apply not only to Eiffel, it makes sense to develop tools that would automatically generate similar templates for other widely used programming languages. Such tools might accept the mathematical formalisms on input and produce the corresponding seamless requirement templates on output.

- Extending the existing IDEs for better support of seamless requirements and their templates. Specification of seamless requirements currently relies on the typical programming-style activities, such as inheritance and generic derivation. While software engineers with programming background may find this process comfortable, former requirements engineers may need a "friendlier" environment that would give them more familiar user experience. Because the ultimate long-term goal of our research is to unify requirements engineering and software construction, catering to practitioners from the both camps is important.

- Detecting known patterns in natural language requirements with their subsequent translation to seamless requirement templates. A huge body of software requirements exists expressed in numerous notations: natural language, UML diagrams, goal diagrams, temporal logics, Z notations and so forth. Manually converting them to the seamless form will take ages and will be considered as waste in the software engineering world dominated by agile methodologies. Also, we think that early requirements will still be captured in natural language anyway. Early requirements elicitation sessions' success relies mainly on the quality of human communication. Nothing beats quickly drafted natural language notes in their ability to capture the conversation's context and the participants' mood and perception.

- Identifying new patterns in recurring requirements that do not map to existing patterns. Natural language requirements' meaning may recur and still not map to any existing pattern. Tools should exist that would identify new patterns and propose ways to capture them in the form of seamless requirement templates.

- Enriching the Eiffel's contract layer with annotations corresponding to the SOORTs. Ait-Ameur and Méry [AM16] propose integrating domain knowledge and design models through annotations. The annotations enrich the models with semantic information from the target domain. The verification process then automatically takes these annotations into account. In the world of programming languages, this idea maps to the single product principle [Mey09] violated when one states requirements separately from the source code, which is the case with SOORs. With a programming language, design models will map to contracts, and domain knowledge will map to SOORTs. The task will be to develop the annotations corresponding to SOORTs at the level of contracts and thus obey the single product principle. The next task will be to update the existing verification mechanisms, AutoTest and AutoProof, to take the new annotations into account.

- Investigate the possibility of developing general rules for translating temporal properties to a programming language. The SOORTs presented in the dissertation are translations of entire requirement patterns, but it is not clear yet how to generalize the translation to the level of distinct temporal operators. Having such a translation scheme would make it possible to translate arbitrary temporal properties to SOORs and SOORTs.

Part IV

Appendices

Appendix A

Control Software SOORTs

A.1 Absence After

```
note
  description: "P is false after Q"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/yxcwu8vw"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y5h2pw8o"

deferred class
  ABSENCE_AFTER [S, expanded P →CONDITION [S], expanded Q →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        p_does_not_hold: not ({P}).default.holds (system)
      until
        timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is false after " + ({Q}).name
  end

end
```

A.2 Absence Before

```
note
  description: "P is false before R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y28f8xxg"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yykmrtwe"

deferred class
  ABSENCE_BEFORE [S, expanded P →CONDITION [S], expanded R →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
      invariant
        p_does_not_hold_or_else_r_holds: not ({P}).default.holds (system) or else ({R}).default.
             holds (system)
      until
        ({R}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is false before " + ({R}).name
    end

end
```

A.3 Absence Between

```
note
  description: "P is false between Q and R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y4nkt92q"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y4ltn92p"

deferred class
  ABSENCE_BETWEEN [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded R →
       CONDITION [S]]
```

```
inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
      r_does_not_hold: not ({R}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        p_does_not_hold_or_else_r_holds: not ({P}).default.holds (system) or else ({R}).default.
             holds (system)
      until
        ({R}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is false between " + ({Q}).name + " and " + ({R}).name
    end

end
```

A.4 Absence Global

```
note
  description: "P is false globally"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y5a6bb8u"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yx9cd6va"

deferred class
  ABSENCE_GLOBAL [S, expanded P −>CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
        init (system)
      invariant
        p_does_not_hold: not ({P}).default.holds (system)
      until
        timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is false globally"
    end

end
```

A.5 Absence Until

```
note
  description: "P is false after Q until R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y3onr2bn"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y69x5dlr"

deferred class
  ABSENCE_UNTIL [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded R →
       CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
      r_does_not_hold: not ({R}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        p_does_not_hold_or_else_r_holds: not ({P}).default.holds (system) or else ({R}).default.
             holds (system)
      until
        ({R}).default.holds (system) or else timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end
```

```
feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is false after " + ({Q}).name + " until " + ({R}).name
    end

end
```

A.6 Bounded Existence Between

```
note
  description: "Transitions to P occur at most 2 times between Q and R."
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y4nr2h8x"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yypy2pgb"

deferred class
  BOUNDED_EXISTENCE_BETWEEN [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded
        R →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
    do
      from
        timer := time_boundary
      until
        ({R}).default.holds (system) or else not ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({R}).default.holds (system) or else ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({R}).default.holds (system) or else not ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({R}).default.holds (system) or else ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({R}).default.holds (system) or else not ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      invariant
        ({R}).default.holds (system) or else not ({P}).default.holds (system)
      until
        ({R}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    ensure
      r_holds: ({R}).default.holds (system)
    end

feature

  requirement_specific_output: STRING
    do
      Result := "transitions to " + ({P}).name + " occur at most 2 times between " + ({Q}).name + " and "
           + ({R}).name
    end

end
```

A.7 Condition

```
note
  description: "Condition over S."
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yya4fncg"

deferred class
  CONDITION [S]

feature

  holds (system: S): BOOLEAN
    deferred
    end

end
```

A.8 Existence After

```
note
  description: "P becomes true after Q"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y644k9hl"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y2psqqzk"

deferred class
  EXISTENCE_AFTER [S, expanded P →CONDITION [S], expanded Q →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
        init (system)
      until
        ({Q}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " becomes true after " + ({Q}).name
    end

end
```

A.9 Existence Before

```
note
  description: "P becomes true before R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y584yaqr"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yyufzv2g"

deferred class
  EXISTENCE_BEFORE [S, expanded P →CONDITION [S], expanded R →CONDITION [S]]
```

```
inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
      invariant
        r_does_not_hold: not ({R}).default.holds (system)
      until
        ({P}).default.holds (system) or else timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " becomes true before " + ({R}).name
    end

end
```

A.10 Existence Between

```
note
  description: "P becomes true between Q and R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y2prdopt"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y5vq9hg9"

deferred class
  EXISTENCE_BETWEEN [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded R →
       CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
      r_does_not_hold: not ({R}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
```

```
        r_does_not_hold: not ({R}).default.holds (system)
      until
        ({P}).default.holds (system) or else timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " becomes true between " + ({Q}).name + " and " + ({R}).name
    end

end
```

A.11 Existence Global

```
note
  description: "P becomes true globally"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y5rrbsrk"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yxgvkktt"

deferred class
  EXISTENCE_GLOBAL [S, expanded P →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
        init (system)
      until
        ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " becomes true globally"
    end

end
```

A.12 Existence Until

```
note
  description: "P becomes true after Q until R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y55xy2aq"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yy5we3xq"

deferred class
  EXISTENCE_UNTIL [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded R →
       CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
      r_does_not_hold: not ({R}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        r_does_not_hold: not ({R}).default.holds (system)
      until
        ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " becomes true after " + ({Q}).name + " until " + ({R}).name
    end

end
```

A.13 Precedence After

```
note
  description: "S precedes P after Q"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y54958zw"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y4bqc3eb"

deferred class
  PRECEDENCE_AFTER [G, expanded S →CONDITION [G], expanded P →CONDITION [G], expanded Q →
       CONDITION [G]]

inherit
```

```
  REQUIREMENT [G]

feature

  frozen verify (system: G)
    require
      q_holds: ({Q}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        p_does_not_hold_or_else_s: not ({P}).default.holds (system) or else ({S}).default.holds (
              system)
      until
        ({S}).default.holds (system) or else timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({S}).name + " precedes " + ({P}).name + " after " + ({Q}).name
    end

end
```

A.14 Precedence Chain Global

```
note
  description: "S, T precedes P globally."
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y22s7fed"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yxgqazkn"

deferred class
  PRECEDENCE_CHAIN_GLOBAL [G, expanded S →CONDITION [G], expanded T →CONDITION [G], expanded P
        →CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    do
      from
        timer := time_boundary
        init (system)
      invariant
        not ({P}).default.holds (system)
      until
        ({S}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
        iterate (system)
      invariant
        not ({P}).default.holds (system) or else ({T}).default.holds (system)
      until
        ({T}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({S}).name + ", " + ({T}).name + " precedes " + ({P}).name + " globally"
    end

end
```

A.15 Precedence Global

```
note
  description: "S precedes P globally"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y5rmuwef"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y3d6xscj"

deferred class
  PRECEDENCE_GLOBAL [G, expanded S →CONDITION [G], expanded P →CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    do
      from
      timer := time_boundary
      init (system)
    invariant
      p_does_not_hold_or_else_s_holds: not ({P}).default.holds (system) or else ({S}).default.
           holds (system)
    until
      ({S}).default.holds (system) or else timer = 0
    loop
      iterate (system)
    variant
      timer
    end
  end

feature

  requirement_specific_output: STRING
    do
      Result := ({S}).name + " precedes " + ({P}).name + " globally"
    end

end
```

A.16 Requirement

```
note
  description: "Verifiable requirement over S."
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y5rqwzs9"

deferred class
  REQUIREMENT [S]

inherit

  ANY
    undefine
      out
    end

feature

  init (system: S)
    deferred
    end

  main (system: S)
    deferred
    end

  iterate (system: S)
    do
      main (system)
      timer := timer − time_growth
    end

  out: STRING
```

```
    do
      Result := Current.generating_type.name + ": in "
      Result := Result + ({S}).name + ", "
      Result := Result + requirement_specific_output + ". "
      Result := Result + "The effect should be observed within " + time_boundary.out + " "
      Result := Result + time_unit
      if time_boundary > 1 then
        Result := Result + "s"
      end
      Result := Result + ".%N"
    end

  requirement_specific_output: STRING
    deferred
    end

feature

  timer: INTEGER

  time_boundary: INTEGER
    do
      Result := {INTEGER}.max_value
    end

  time_growth: INTEGER
    do
      Result := 1
    end

  time_unit: STRING
    do
      Result := "time unit"
    end

end
```

A.17 Response After

```
note
  description: "S responds to P after Q"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/yyso3qn8"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yyl4ebge"

deferred class
  RESPONSE_AFTER [G, expanded S →CONDITION [G], expanded P →CONDITION [G], expanded Q →
       CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    require
```

```
      q_holds: ({Q}).default.holds (system)
    do
      from
        timer := time_boundary
      until
        ({P}).default.holds (system) or else timer = 0
      loop
        iterate (system)
      variant
        timer
      end
      check
        assume: ({P}).default.holds (system)
      end
      from
      until
        ({S}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({S}).name + " responds to " + ({P}).name + " after " + ({Q}).name + "."
    end

end
```

A.18 Response Before

```
note
  description: "S responds to P before R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y2b69k9o"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y56g5ok5"

deferred class
  RESPONSE_BEFORE [G, expanded S →CONDITION [G], expanded P →CONDITION [G], expanded R →
       CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    require
      p_holds: ({P}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        r_does_not_hold: not ({R}).default.holds (system)
      until
        ({S}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({R}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({S}).name + " responds to " + ({P}).name + " before " + ({R}).name
    end

end
```

A.19 Response Chain Global

```
note
  description: "P responds to S, T globally."
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y32tgtcm"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yyr8xw2b"

deferred class
  RESPONSE_CHAIN_GLOBAL [G, expanded P →CONDITION [G], expanded S →CONDITION [G], expanded T
        →CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    require
      ({S}).default.holds (system)
    do
      from
        timer := time_boundary
        iterate (system)
      until
        ({T}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
      from
      until
        ({P}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " responds to " + ({S}).name + ", " + ({T}).name + " globally"
    end

end
```

A.20 Response Global

```
note
  description: "S responds to P globally"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y44wbnbs"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y2crlkjc"

deferred class
  RESPONSE_GLOBAL [G, expanded S →CONDITION [G], expanded P →CONDITION [G]]

inherit

  REQUIREMENT [G]

feature

  frozen verify (system: G)
    require
      p_holds: ({P}).default.holds (system)
    do
      from
        timer := time_boundary
      until
        ({S}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
```

```
    do
      Result := ({S}).name + " responds to " + ({P}).name + " globally"
    end

end
```

A.21 Universality After

```
note
  description: "P is true after Q"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y3e7vrvx"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y35pz34n"

deferred class
  UNIVERSALITY_AFTER [S, expanded P →CONDITION [S], expanded Q →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        p_holds: ({P}).default.holds (system)
      until
        timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is true after " + ({Q}).name
    end

end
```

A.22 Universality Before

```
note
  description: "P is true before R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/yxmn65yo"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y42m9uth"
```

```
deferred class
  UNIVERSALITY_BEFORE [S, expanded P →CONDITION [S], expanded R →CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
      invariant
        ({P}).default.holds (system) or else ({R}).default.holds (system)
      until
        ({R}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is true before " + ({R}).name
    end

end
```

A.23 Universality Between

```
note
  description: "P is true between Q and R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/yxmkw6s5"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/yypj6uhf"

deferred class
  UNIVERSALITY_BETWEEN [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded R →
        CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
      r_does_not_hold: not ({R}).default.holds (system)
    do
      from
```

```
        timer := time_boundary
      invariant
        p_holds_or_else_r: ({P}).default.holds (system) or else ({R}).default.holds (system)
      until
        ({R}).default.holds (system)
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result := ({P}).name + " is true between " + ({Q}).name + " and " + ({R}).name
    end

end
```

A.24 Universality Global

```
note
  description: "P is true globally"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y3hrpltn"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y46rbz87"

deferred class
  UNIVERSALITY_GLOBAL [S, expanded P ->CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    do
      from
        timer := time_boundary
        init (system)
      invariant
        p_holds: ({P}).default.holds (system)
      until
        timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
```

```
    do
      Result :=  ({P}).name + " is true globally"
    end

end
```

A.25 Universality Until

```
note
  description: "P is true after Q until R"
  EIS: "name= Multirequirement", "src= http://tinyurl.com/y3mgklvw"
  EIS: "name= Location on GitHub", "src= http://tinyurl.com/y65zzxke"

deferred class
  UNIVERSALITY_UNTIL [S, expanded P →CONDITION [S], expanded Q →CONDITION [S], expanded R →
        CONDITION [S]]

inherit

  REQUIREMENT [S]

feature

  frozen verify (system: S)
    require
      q_holds: ({Q}).default.holds (system)
      r_does_not_hold: not ({R}).default.holds (system)
    do
      from
        timer := time_boundary
      invariant
        p_holds_or_else_r_holds: ({P}).default.holds (system) or else ({R}).default.holds (
             system)
      until
        ({R}).default.holds (system) or else timer = 0
      loop
        iterate (system)
      variant
        timer
      end
    end

feature

  requirement_specific_output: STRING
    do
      Result :=  ({P}).name + " is true after " + ({Q}).name + " until " + ({R}).name
    end

end
```

Appendix B

Software Components SOORTs

B.1 Array

```
note
  description: "Reusable abstract data type specification of array."
  description: "Found in ‘‘Abstract Data Types and Software Validation ’’ by Guttag, Horowitz and
       Musser:"
  EIS: "src= https://pdfs.semanticscholar.org/372d/4f331d0a6cd5fb4ee0c04d4a0753b8eb659f.pdf"
  description: "Found in ‘‘Abstract Data Types and the Development of Data Structures’’ by Guttag:"
  EIS: "src= http://tinyurl.com/y45o32hq"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y69xc6fy"

deferred class
  ARRAY_ADT [A, E]
  -- Arrays ‘‘A’’ contain elements of ‘‘E’’.

inherit

  EQUALITY_ADT [A]

feature
  -- Deferred definitions.

  make (new_first, new_last: INTEGER): A
    deferred
    end

  put (array: A; index: INTEGER; element: E)
    deferred
    end

  first (array: A): INTEGER
    deferred
    end

  last (array: A): INTEGER
    deferred
    end
  eval (array: A; index: INTEGER): E
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (new_first, new_last: INTEGER)
    local
      new_array: A
    do
      new_array := make (new_first, new_last)
      check
        first (new_array) ~ new_first
      end
    end

  frozen a_2 (array: A; index: INTEGER; element: E; old_first: INTEGER)
    require
      first (array) ~ old_first
    do
      put (array, index, element)
    ensure
      first (array) ~ old_first
    end

  frozen a_3 (new_first, new_last: INTEGER)
    local
      new_array: A
    do
      new_array := make (new_first, new_last)
      check
        last (new_array) ~ new_last
      end
    end

  frozen a_4 (array: A; index: INTEGER; element: E; old_last: INTEGER)
    require
      last (array) ~ old_last
    do
      put (array, index, element)
    ensure
      last (array) ~ old_last
    end

  frozen a_5 (new_first, new_last: INTEGER; index: INTEGER; element: E)
    local
      array: A
    do
      array := make (new_first, new_last)
      check
        eval (array, index) /~ element
      end
    end

  frozen a_6 (array: A; index_put, index_eval: INTEGER; element_1, element_2: E)
```

```
    require
      index_eval < first (array)
    do
      put (array, index_put, element_1)
    ensure
      eval (array, index_eval) /~ element_2
    end

  frozen a_7 (array: A; index_put, index_eval: INTEGER; element_1, element_2: E)
    require
      index_eval > last (array)
    do
      put (array, index_put, element_1)
    ensure
      eval (array, index_eval) /~ element_2
    end

  frozen a_8 (array: A; index: INTEGER; element: E)
    require
      index ≥ first (array)
      index ≤ last (array)
    do
      put (array, index, element)
    ensure
      eval (array, index) ~ element
    end

  frozen a_9 (array: A; index_put: INTEGER; element: E; index_eval: INTEGER; old_element: E)
    require
      index_eval ≥ first (array)
      index_eval ≤ last (array)
      index_put /~ index_eval
      eval (array, index_eval) ~ old_element
    do
      put (array, index_put, element)
    ensure
      eval (array, index_eval) ~ old_element
    end

feature

  frozen make_well_defined (new_first, new_last: INTEGER)
    local
      array_1, array_2: A
    do
      array_1 := make (new_first, new_last)
      array_2 := make (new_first, new_last)
      check
        array_1 ≠ array_2
      end
      check
        array_1 ~ array_2
      end
    end

  frozen put_well_defined (array_1, array_2: A; index: INTEGER; element: E)
    require
```

```
      array_1 ~ array_2
    do
      put (array_1, index, element)
      put (array_2, index, element)
    ensure
      array_1 ~ array_2
    end

  frozen first_well_defined (array_1, array_2: A)
    require
      array_1 ~ array_2
    do
    ensure
      first (array_1) ~ first (array_2)
    end

  frozen last_well_defined (array_1, array_2: A)
    require
      array_1 ~ array_2
    do
    ensure
      last (array_1) ~ last (array_2)
    end

  frozen eval_well_defined (array_1, array_2: A; index: INTEGER)
    require
      array_1 ~ array_2
    do
    ensure
      eval (array_1, index) ~ eval (array_2, index)
    end

end
```

B.2 Bag

```
note
  description: "Reusable abstract data type specification of bag."
  description: "Found in ‘‘The Algebraic Specification of Abstract Data Types’’ by Guttag and Horning:
       "
  EIS: "src= https://link.springer.com/article/10.1007/BF00260922"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yyensvjt"

deferred class
  BAG_ADT [B, E]
  -- Bags ‘‘B’’ contain elements of ‘‘E’’.

inherit

  EQUALITY_ADT [B]

feature
  -- Deferred definitions.

  empty_bag: B
    deferred
    end

  insert (bag: B; element: E)
    deferred
    end

  delete (bag: B; element: E)
    deferred
    end

  member_of (bag: B; element: E): BOOLEAN
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (element: E)
    local
      bag: B
    do
      bag := empty_bag
      check
        not member_of (bag, element)
      end
    end

  frozen a_2_1 (bag: B; element: E)
    do
      insert (bag, element)
    ensure
      member_of (bag, element)
    end

  frozen a_2_2 (bag: B; element_1, element_2: E; old_member_of: BOOLEAN)
    require
      element_1 /~ element_2
      member_of (bag, element_2) ~ old_member_of
    do
      insert (bag, element_1)
    ensure
      member_of (bag, element_2) ~ old_member_of
    end

  frozen a_3 (element: E)
    local
      bag_1, bag_2: B
    do
      bag_1 := empty_bag
      bag_2 := empty_bag
      delete (bag_1, element)
      check
        bag_1 ~ bag_2
      end
    end

  frozen a_4_1 (bag_1, bag_2: B; element: E)
```

```
    do
      check
        assume: bag_1 ~ bag_2
      end
      insert (bag_1, element)
      delete (bag_1, element)
      check
        assert: bag_1 ~ bag_2
      end
    end

  frozen a_4_2 (bag_1, bag_2: B; element_1, element_2: E)
    require
      bag_1 ~ bag_2
      element_1 /~ element_2
    do
      insert (bag_1, element_1)
      delete (bag_1, element_2)
      delete (bag_2, element_2)
      insert (bag_2, element_1)
    ensure
      bag_1 ~ bag_2
    end

feature
  -- Well-definedness axioms.

  frozen empty_bag_well_defined
    local
      bag_1, bag_2: B
    do
      bag_1 := empty_bag
      bag_2 := empty_bag
      check
        assert: bag_1 ≠ bag_2
      end
      check
        assert: bag_1 ~ bag_2
      end
    end

  frozen insert_well_defined (bag_1, bag_2: B; element: E)
    require
      bag_1 ~ bag_2
    do
      insert (bag_1, element)
      insert (bag_2, element)
    ensure
      bag_1 ~ bag_2
    end

  frozen delete_well_defined (bag_1, bag_2: B; element: E)
    require
      bag_1 ~ bag_2
    do
      delete (bag_1, element)
      delete (bag_2, element)
    ensure
      bag_1 ~ bag_2
    end

  frozen member_of_well_defined (bag_1, bag_2: B; element: E)
    require
      bag_1 ~ bag_2
    do
    ensure
      member_of (bag_1, element) ~ member_of (bag_2, element)
    end

end
```

B.3 Binary Tree

```
note
  description: "Reusable abstract data type specification of binary tree."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y2rhrktn"

deferred class
  BINARY_TREE_ADT [B, I]
  -- Binary trees ‘‘B’’ contain elements of ‘‘I’’.

inherit

  EQUALITY_ADT [B]

feature
  -- Deferred definitions.

  empty_tree: B
    deferred
    end

  make (b_tree_left: B; item: I; b_tree_right: B): B
    deferred
    end

  is_empty_tree (b_tree: B): BOOLEAN
    deferred
    end

  left (b_tree: B): B
    deferred
    end

  data (b_tree: B): I
    deferred
    end

  right (b_tree: B): B
    deferred
    end
  is_in (b_tree: B; item: I): BOOLEAN
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1
    local
      b_tree: B
    do
      b_tree := empty_tree
      check
        is_empty_tree (b_tree)
      end
    end

  frozen a_2 (b_tree_left: B; item: I; b_tree_right: B)
    local
      b_tree: B
    do
      b_tree := make (b_tree_left, item, b_tree_right)
      check
        not is_empty_tree (b_tree)
      end
    end

  frozen a_3
    local
      b_tree_1, b_tree_2: B
    do
      b_tree_1 := empty_tree
      b_tree_2 := empty_tree
      check
        left (b_tree_1) ~ b_tree_2
      end
    end

  frozen a_4 (b_tree_left: B; item: I; b_tree_right: B)
    local
      b_tree: B
    do
      b_tree := make (b_tree_left, item, b_tree_right)
      check
        left (b_tree) ~ b_tree_left
      end
    end

  frozen a_5 (item: I)
    local
      b_tree: B
    do
      b_tree := empty_tree
      check
        data (b_tree) /~ item
      end
    end

  frozen a_6 (b_tree_left: B; item: I; b_tree_right: B)
    local
      b_tree: B
    do
      b_tree := make (b_tree_left, item, b_tree_right)
      check
        data (b_tree) ~ item
      end
    end

  frozen a_7
    local
      b_tree_1, b_tree_2: B
    do
      b_tree_1 := empty_tree
      b_tree_2 := empty_tree
      check
        right (b_tree_1) ~ b_tree_2
      end
    end

  frozen a_8 (b_tree_left: B; item: I; b_tree_right: B)
    local
      b_tree: B
    do
      b_tree := make (b_tree_left, item, b_tree_right)
      check
        right (b_tree) ~ b_tree_right
      end
    end

  frozen a_9 (item: I)
    local
      b_tree: B
    do
      b_tree := empty_tree
      check
        not is_in (b_tree, item)
      end
    end

  frozen a_10 (b_tree_left: B; item_1, item_2: I; b_tree_right: B)
    local
      b_tree: B
    do
      b_tree := make (b_tree_left, item_1, b_tree_right)
      check
        is_in (b_tree, item_2) = (item_1 ~ item_2 or is_in (b_tree_left, item_2) or is_in (
             b_tree_right, item_2))
      end
    end

feature
  -- Well-definedness axioms.
```

```
frozen empty_tree_well_defined
  local
    b_tree_1, b_tree_2: B
  do
    b_tree_1 := empty_tree
    b_tree_2 := empty_tree
    check
      b_tree_1 ≠ b_tree_2
    end
    check
      b_tree_1 ~ b_tree_2
    end
  end

frozen make_well_defined (b_tree_left: B; item: I; b_tree_right: B)
  local
    b_tree_1, b_tree_2: B
  do
    b_tree_1 := make (b_tree_left, item, b_tree_right)
    b_tree_2 := make (b_tree_left, item, b_tree_right)
    check
      b_tree_1 ≠ b_tree_2
    end
    check
      b_tree_1 ~ b_tree_2
    end
  end

frozen is_empty_tree_well_defined (b_tree_1, b_tree_2: B)
  require
    b_tree_1 ~ b_tree_2
  do
  ensure
    is_empty_tree (b_tree_1) ~ is_empty_tree (b_tree_2)
  end

frozen left_well_defined (b_tree_1, b_tree_2: B)
  require
    b_tree_1 ~ b_tree_2
  do
  ensure
    left (b_tree_1) ~ left (b_tree_2)
  end

frozen data_well_defined (b_tree_1, b_tree_2: B)
  require
    b_tree_1 ~ b_tree_2
  do
  ensure
    data (b_tree_1) ~ data (b_tree_2)
  end

frozen right_well_defined (b_tree_1, b_tree_2: B)
  require
    b_tree_1 ~ b_tree_2
  do
  ensure
```

```
      right (b_tree_1) ~ right (b_tree_2)
    end

  frozen is_in_well_defined (b_tree_1, b_tree_2: B; item: I)
    require
      b_tree_1 ~ b_tree_2
    do
    ensure
      is_in (b_tree_1, item) ~ is_in (b_tree_2, item)
    end

end
```

B.4 Binary Tree with Inord

```
note
  description: "Reusable abstract data type specification of binary tree with ‘‘inord’’ operation."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y3peoll5"

deferred class
  BINARY_TREE_WITH_INORD_ADT [B, I, Q, QS −→QUEUE_WITH_APPEND_ADT [Q, I]]
  -- Binary trees ‘‘B’’ contain elements of ‘‘I’’.
  -- They rely on queues ‘‘Q’’ with elements of ‘‘I’’ conforming to the
  -- ‘‘QUEUE_WITH_APPEND_ADT’’ specification.

inherit

  BINARY_TREE_ADT [B, I]

feature
  -- Deferred definitions.

  in_ord (b_tree: B): Q
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_11
    local
      b_tree: B
    do
      b_tree := empty_tree
      check
        in_ord (b_tree) ~ ({QS}).default.newq
      end
    end

  frozen a_12 (b_tree_left: B; item: I; b_tree_right: B; q_left, q_right: Q)
    require
      in_ord (b_tree_left) ~ q_left
      in_ord (b_tree_right) ~ q_right
    local
```

```
      b_tree: B
    do
      b_tree := make (b_tree_left, item, b_tree_right)
      ({QS}).default.addq (q_left, item)
      ({QS}).default.appendq (q_left, q_right)
      check
        in_ord (b_tree) ~ q_left
      end
    end

feature
  -- Well-definedness axioms.

  frozen in_ord_well_defined (b_tree_1, b_tree_2: B)
    require
      b_tree_1 ~ b_tree_2
    do
    ensure
      in_ord (b_tree_1) ~ in_ord (b_tree_2)
    end

end
```

B.5 Book Directory

```
note
  description: "Reusable abstract data type specification of searchable book directory."
  description: "Found in ‘‘Requirements engineering: From system goals to UML models to software.’’
        by van Lamsweerde:"
  EIS: "src= http://tinyurl.com/yxd3zxd2"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y6ft5d3a"

deferred class
  BOOK_DIRECTORY_ADT [D, B, BC, T, L, LS →LIST_ADT [L, B]]
  -- Book directories ‘‘D’’ contain books ‘‘B’’ with topics ‘‘T’’ and book copies ‘‘BC’’.
  -- Searching by topics returns lists ‘‘L’’ of books ‘‘B’’ conforming to the
  -- ‘‘LIST_ADT’’ specification.

inherit

  EQUALITY_ADT [D]

feature
  -- Deferred definitions.

  empty_dir: D
    deferred
    end

  add_entry (d: D; b: B; bc: BC; t: T)
    deferred
    end

  biblio_search (d: D; t: T): L
    deferred
    end
```

```
feature
  -- Abstract data type axioms.

  frozen a_1 (tp: T)
    local
      dir: D
    do
      dir := empty_dir
      check
        biblio_search (dir, tp) ~ ({LS}).default.nil
      end
    end

  frozen a_2 (dir: D; b: B; bc: BC; tp: T; bs: L)
    require
      biblio_search (dir, tp) ~ bs
    do
      ({LS}).default.cons (bs, b)
      add_entry (dir, b, bc, tp)
    ensure
      biblio_search (dir, tp) ~ bs
    end

  frozen a_3 (dir: D; b: B; bc: BC; tp_1, tp_2: T; bs: L)
    require
      biblio_search (dir, tp_1) ~ bs
    do
      add_entry (dir, b, bc, tp_2)
    ensure
      biblio_search (dir, tp_1) ~ bs
    end

feature
  -- Well-definedness axioms

  frozen empty_dir_well_defined
    local
      d_1, d_2: D
    do
      d_1 := empty_dir
      d_2 := empty_dir
      check
        d_1 ≠ d_2
      end
      check
        d_1 ~ d_2
      end
    end

  frozen add_entry_well_defined (dir_1, dir_2: D; b: B; bc: BC; tp: T)
    require
      dir_1 ~ dir_2
    do
      add_entry (dir_1, b, bc, tp)
      add_entry (dir_2, b, bc, tp)
    ensure
```

```
      dir_1 ~ dir_2
    end

  frozen biblio_search_well_defined (dir_1, dir_2: D; tp: T)
    require
      dir_1 ~ dir_2
    do
    ensure
      biblio_search (dir_1, tp) ~ biblio_search (dir_2, tp)
    end

end
```

B.6 Bounded Queue

```
note
  description: "Reusable abstract data type specification of bounded queue."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yybezkrm"

deferred class
  BOUNDED_QUEUE_ADT [B, I]
  -- Bounded queues ‘‘B’’ contain elements of ‘‘I’’.

inherit

  EQUALITY_ADT [B]

feature
  -- Deferred definitions.

  newq (capacity: INTEGER): B
    deferred
    end

  addq (bounded_queue: B; item: I)
    deferred
    end

  deleteq (bounded_queue: B)
    deferred
    end

  frontq (bounded_queue: B): I
    deferred
    end

  isnewq (bounded_queue: B): BOOLEAN
    deferred
    end

  appendq (bounded_queue, other: B)
    deferred
    end
  size (bounded_queue: B): INTEGER
    deferred
    end

  limit (bounded_queue: B): INTEGER
    deferred
    end

  enq (bounded_queue: B; item: I)
    deferred
    end

  deq (bounded_queue: B): I
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (capacity: INTEGER)
    local
      bounded_queue: B
    do
      bounded_queue := newq (capacity)
      check
        isnewq (bounded_queue)
      end
    end

  frozen a_2 (bounded_queue: B; item: I)
      -- ISNEWQ(ADDQ(q,i)) = false
    do
      addq (bounded_queue, item)
    ensure
      not isnewq (bounded_queue)
    end

  frozen a_3 (capacity: INTEGER)
    local
      bounded_queue_1, bounded_queue_2: B
    do
      bounded_queue_1 := newq (capacity)
      bounded_queue_2 := newq (capacity)
      deleteq (bounded_queue_1)
      check
        bounded_queue_1 ~ bounded_queue_2
      end
    end

  frozen a_4 (bounded_queue: B; item: I; capacity: INTEGER)
    require
      isnewq (bounded_queue)
    local
      new_queue: B
    do
      new_queue := newq (capacity)
      addq (bounded_queue, item)
      deleteq (bounded_queue)
      check
        bounded_queue ~ new_queue
      end
    end

  frozen a_5 (bounded_queue_1, bounded_queue_2: B; item: I)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
      addq (bounded_queue_1, item)
      deleteq (bounded_queue_1)
      deleteq (bounded_queue_2)
      addq (bounded_queue_2, item)
    ensure
      bounded_queue_1 ~ bounded_queue_2
    end

  frozen a_6 (capacity: INTEGER; item: I)
    local
      bounded_queue: B
    do
      bounded_queue := newq (capacity)
      check
        frontq (bounded_queue) /~ item
      end
    end

  frozen a_7 (bounded_queue: B; item: I)
    require
      isnewq (bounded_queue)
    do
      addq (bounded_queue, item)
    ensure
      frontq (bounded_queue) ~ item
    end

  frozen a_8 (bounded_queue: B; item: I; old_frontq: I)
    require
      not isnewq (bounded_queue)
      frontq (bounded_queue) ~ old_frontq
    do
      addq (bounded_queue, item)
    ensure
      frontq (bounded_queue) ~ old_frontq
    end

  frozen a_9 (bounded_queue_1, bounded_queue_2: B; capacity: INTEGER)
    require
      bounded_queue_1 ~ bounded_queue_2
    local
      new_queue: B
    do
      new_queue := newq (capacity)
      appendq (bounded_queue_1, new_queue)
    ensure
      bounded_queue_1 ~ bounded_queue_2
  end

frozen a_10 (bounded_queue_1, bounded_queue_2, other_1, other_2: B; item: I)
  require
    bounded_queue_1 ~ bounded_queue_2
    other_1 ~ other_2
  do
    addq (other_1, item)
    appendq (bounded_queue_1, other_1)
    appendq (bounded_queue_2, other_2)
    addq (bounded_queue_2, item)
  ensure
    bounded_queue_1 ~ bounded_queue_2
  end

frozen a_11 (capacity: INTEGER)
  local
    new_queue: B
  do
    new_queue := newq (capacity)
    check
      limit (new_queue) ~ capacity
    end
  end

frozen a_12 (bounded_queue: B; item: I; old_limit: INTEGER)
  require
    limit (bounded_queue) ~ old_limit
  do
    addq (bounded_queue, item)
  ensure
    limit (bounded_queue) ~ old_limit
  end

frozen a_13 (bounded_queue_1, bounded_queue_2: B; item: I)
  require
    size (bounded_queue_1) < limit (bounded_queue_1)
    bounded_queue_1 ~ bounded_queue_2
  do
    enq (bounded_queue_1, item)
    addq (bounded_queue_2, item)
  ensure
    bounded_queue_1 ~ bounded_queue_2
  end

frozen a_14 (bounded_queue_1, bounded_queue_2: B; item: I)
  require
    size (bounded_queue_1) = limit (bounded_queue_1)
  do
    enq (bounded_queue_1, item)
  ensure
    bounded_queue_1 /~ bounded_queue_2
  end

frozen a_15 (bounded_queue_1, bounded_queue_2: B)
  require
    bounded_queue_1 ~ bounded_queue_2
    local
      item: I
    do
      item := deq (bounded_queue_1)
      deleteq (bounded_queue_2)
      check
        item ~ frontq (bounded_queue_2)
      end
    end

  frozen a_16 (capacity: INTEGER)
    local
      bounded_queue: B
    do
      bounded_queue := newq (capacity)
      check
        size (bounded_queue) ~ 0
      end
    end

  frozen a_17 (bounded_queue: B; item: I; old_size: INTEGER)
    require
      size (bounded_queue) ~ old_size
    do
      addq (bounded_queue, item)
    ensure
      size (bounded_queue) ~ 1 + old_size
    end

feature
  -- Well-definedness axioms.

  frozen newq_well_defined (capacity: INTEGER)
    local
      bounded_queue_1, bounded_queue_2: B
    do
      bounded_queue_1 := newq (capacity)
      bounded_queue_2 := newq (capacity)
      check
        bounded_queue_1 ~ bounded_queue_2
      end
    end

  frozen addq_well_defined (bounded_queue_1, bounded_queue_2: B; item: I)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
      addq (bounded_queue_1, item)
      addq (bounded_queue_2, item)
    ensure
      bounded_queue_1 ~ bounded_queue_2
    end

  frozen deleteq_well_defined (bounded_queue_1, bounded_queue_2: B)
    require
      bounded_queue_1 ~ bounded_queue_2
      bounded_queue_1 ≠ bounded_queue_2
      not isnewq (bounded_queue_1)
      not isnewq (bounded_queue_2)
    do
      deleteq (bounded_queue_1)
      deleteq (bounded_queue_2)
    ensure
      bounded_queue_1 ~ bounded_queue_2
    end

  frozen frontq_well_defined (bounded_queue_1, bounded_queue_2: B)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
    ensure
      frontq (bounded_queue_1) ~ frontq (bounded_queue_2)
    end

  frozen isnewq_well_defined (bounded_queue_1, bounded_queue_2: B)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
    ensure
      isnewq (bounded_queue_1) ~ isnewq (bounded_queue_2)
    end

  frozen appendq_well_defined (bounded_queue_1, bounded_queue_2, other: B)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
      appendq (bounded_queue_1, other)
      appendq (bounded_queue_2, other)
    ensure
      bounded_queue_1 ~ bounded_queue_2
    end

  frozen size_well_defined (bounded_queue_1, bounded_queue_2: B)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
    ensure
      size (bounded_queue_1) ~ size (bounded_queue_2)
    end

  frozen limit_well_defined (bounded_queue_1, bounded_queue_2: B)
    require
      bounded_queue_1 ~ bounded_queue_2
    do
    ensure
      limit (bounded_queue_1) ~ limit (bounded_queue_2)
    end

  frozen enq_well_defined (bounded_queue_1, bounded_queue_2: B; item: I)
    require
      bounded_queue_1 ~ bounded_queue_2
      size (bounded_queue_1) < limit (bounded_queue_1)
      size (bounded_queue_2) < limit (bounded_queue_2)
    do
```

```
      enq (bounded_queue_1, item)
      enq (bounded_queue_2, item)
    ensure
      bounded_queue_1 ~ bounded_queue_2
    end

  frozen deq_well_defined (bounded_queue_1, bounded_queue_2: B)
    require
      bounded_queue_1 ~ bounded_queue_2
    local
      item_1, item_2: I
    do
      item_1 := deq (bounded_queue_1)
      item_2 := deq (bounded_queue_2)
      check
        item_1 ~ item_2
      end
    ensure
      bounded_queue_1 ~ bounded_queue_2
    end

end
```

B.7 Bounded Stack

```
note
  description: "Reusable abstract data type specification of bounded stack."
  description: "Found in ‘‘Abstract Data Types and Software Validation ’’ by Guttag, Horowitz and
          Musser:"
  EIS: "src= https://pdfs.semanticscholar.org/372d/4f331d0a6cd5fb4ee0c04d4a0753b8eb659f.pdf"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y4wswh3t"

deferred class
  BOUNDED_STACK_ADT [B, E]
  -- Bounded stacks ‘‘B’’ contain elements of ‘‘E’’.

inherit

  EQUALITY_ADT [B]

feature
  -- Deferred definitions.

  new_stack (lim: INTEGER): B
    deferred
    end

  push (stack: B; element: E)
    deferred
    end

  pop (stack: B)
    deferred
    end

  top (stack: B): E
```

```
    deferred
    end

  size (stack: B): INTEGER
    deferred
    end

  limit (stack: B): INTEGER
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (lim: INTEGER)
    local
      stack_1, stack_2: B
    do
      stack_1 := new_stack (lim)
      stack_2 := new_stack (lim)
      pop (stack_1)
      check
        stack_1 ~ stack_2
      end
    end

  frozen a_2 (stack_1, stack_2: B; element: E)
    require
      stack_1 ~ stack_2
      size (stack_1) <limit (stack_1)
    do
      push (stack_1, element)
      pop (stack_1)
    ensure
      stack_1 ~ stack_2
    end

  frozen a_3 (stack_1, stack_2: B; element: E)
    require
      stack_1 ~ stack_2
      size (stack_1) ~ limit (stack_1)
    do
      push (stack_1, element)
    ensure
      stack_1 ~ stack_2
    end

  frozen a_4 (lim: INTEGER; element: E)
    local
      stack: B
    do
      stack := new_stack (lim)
      check
        top (stack) /~ element
      end
    end
```

```
frozen a_5 (stack: B; element: E)
  require
    size (stack) < limit (stack)
  do
    push (stack, element)
  ensure
    top (stack) ~ element
  end

frozen a_6 (stack: B; element_1, element_2: E)
  require
    top (stack) ~ element_2
    size (stack) ~ limit (stack)
  do
    push (stack, element_1)
  ensure
    top (stack) ~ element_2
  end

frozen a_7 (lim: INTEGER)
  local
    stack: B
  do
    stack := new_stack (lim)
    check
      limit (stack) ~ lim
    end
  end

frozen a_8 (stack: B; element: E; old_limit: INTEGER)
  require
    limit (stack) ~ old_limit
    size (stack) ~ limit (stack)
  do
    push (stack, element)
  ensure
    limit (stack) ~ old_limit
  end

frozen a_9 (lim: INTEGER)
  local
    stack: B
  do
    stack := new_stack (lim)
    check
      size (stack) ~ 0
    end
  end

frozen a_10 (stack: B; element: E; old_size: INTEGER)
  require
    size (stack) ~ old_size
    size (stack) < limit (stack)
  do
    push (stack, element)
  ensure
    size (stack) ~ old_size + 1
    end

  frozen a_11 (stack: B; element: E; old_size: INTEGER)
    require
      size (stack) ~ old_size
      size (stack) ~ limit (stack)
    do
      push (stack, element)
    ensure
      size (stack) ~ old_size
    end

feature
  -- Well-definedness axioms.

  frozen new_stack_well_defined (lim: INTEGER)
    local
      stack_1, stack_2: B
    do
      stack_1 := new_stack (lim)
      stack_2 := new_stack (lim)
      check
        stack_1 ≠ stack_2
      end
      check
        stack_1 ~ stack_2
      end
    end

  frozen push_well_defined (stack_1, stack_2: B; element: E)
    require
      stack_1 ~ stack_2
    do
      push (stack_1, element)
      push (stack_2, element)
    ensure
      stack_1 ~ stack_2
    end

  frozen pop_well_defined (stack_1, stack_2: B)
    require
      stack_1 ~ stack_2
      stack_1 ≠ stack_2
    do
      pop (stack_1)
      pop (stack_2)
    ensure
      stack_1 ~ stack_2
    end

  frozen top_well_defined (stack_1, stack_2: B)
    require
      stack_1 ~ stack_2
    do
    ensure
      top (stack_1) ~ top (stack_2)
    end
  frozen size_well_defined (stack_1, stack_2: B)
    require
      stack_1 ~ stack_2
    do
    ensure
      size (stack_1) ~ size (stack_2)
    end

  frozen limit_well_defined (stack_1, stack_2: B)
    require
      stack_1 ~ stack_2
    do
    ensure
      limit (stack_1) ~ limit (stack_2)
    end

end
```

B.8 Commutative Ring

```
note
  description: "Reusable abstract data type specification of commutative ring."
  description: "Found in Wikipedia:"
  EIS: "src= https://en.wikipedia.org/wiki/Commutative_ring#Definition"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yytyfmqj"

deferred class
  COMMUTATIVE_RING_ADT [R]
  -- ‘‘R’’ is a mathematical commutative ring.

inherit

  EQUALITY_ADT [R]

feature
  -- Deferred definitions.

  one: R
    deferred
    end

  zero: R
    deferred
    end

  sum (summand_1, summand_2: R): R
    deferred
    end

  product (multiplier_1, multiplier_2: R): R
    deferred
    end

  additive_inverse (a: R): R
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (a, b, c: R)
    do
    ensure
      sum (sum (a, b), c) ~ sum (a, sum (b, c))
    end

  frozen a_2 (a, b: R)
    do
    ensure
      sum (a, b) ~ sum (b, a)
    end

  frozen a_3 (a: R)
    do
    ensure
      sum (a, zero) ~ a
    end

  frozen a_4 (a: R)
    do
    ensure
      sum (a, additive_inverse (a)) ~ zero
    end

  frozen a_5 (a, b, c: R)
    do
    ensure
      product (product (a, b), c) ~ product (a, product (b, c))
    end

  frozen a_6 (a: R)
    do
    ensure
      product (a, one) ~ a
      product (one, a) ~ a
    end

  frozen a_7 (a, b, c: R)
    do
    ensure
      product (a, sum (b, c)) ~ sum (product (a, b), product (a, c))
    end

  frozen a_8 (a, b, c: R)
    do
    ensure
      product (sum (b, c), a) ~ sum (product (b, a), product (c, a))
    end

  frozen a_9 (a, b: R)
    do
    ensure
```

```
      product (a, b) ~ product (b, a)
    end

feature
  -- Well-definedness axioms.

  frozen one_well_defined
    local
      r_1, r_2: R
    do
      r_1 := one
      r_2 := one
      check
        r_1 ~ r_2
      end
    end

  frozen zero_well_defined
    local
      r_1, r_2: R
    do
      r_1 := zero
      r_2 := zero
      check
        r_1 ~ r_2
      end
    end

  frozen sum_well_defined (summand_1, summand_2, other: R)
    require
      summand_1 ~ summand_2
    do
    ensure
      sum (summand_1, other) ~ sum (summand_2, other)
    end

  frozen product_well_defined (multiplier_1, multiplier_2, other: R)
    require
      multiplier_1 ~ multiplier_2
    do
    ensure
      product (multiplier_1, other) ~ product (multiplier_2, other)
    end

  frozen additive_inverse_well_defined (r_1, r_2: R)
    require
      r_1 ~ r_2
    do
    ensure
      additive_inverse (r_1) ~ additive_inverse (r_2)
    end

end
```

B.9 Edge

```
note
  description: "Reusable abstract data type specification of graph edge."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y4g93wpo"

deferred class
  EDGE_ADT [E, N]
  -- Edges ‘‘E’’ connect nodes ‘‘N’’.

inherit

  EQUALITY_ADT [E]

feature
  -- Deferred definitions.

  rel (node_1, node_2: N): E
    deferred
    end

feature
  -- Well-definedness axioms.

  frozen rel_well_defined (node_1, node_2: N)
    local
      edge_1, edge_2: E
    do
      edge_1 := rel (node_1, node_2)
      edge_2 := rel (node_1, node_2)
      check
        edge_1 ~ edge_2
      end
    end

end
```

B.10 Environment

```
note
  description: "Reusable abstract data type specification of environment."
  description: "Found in ‘‘An abstract data type for name analysis’’ by Kastens and Waite:"
  EIS: "src= http://tinyurl.com/y2ghqjq7"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yyb6uwxy"

deferred class
  ENVIRONMENT_ADT [E, I, D]
  -- Environments ‘‘E’’ contain keys ‘‘D’’ inentified by elements of ‘‘I’’.

inherit

  EQUALITY_ADT [E]

feature
  -- Deferred definitions.
  new_env: E
    deferred
    end

  new_scope (env: E)
    deferred
    end

  add (env: E; id: I; key: D)
    deferred
    end

  key_in_scope (env: E; id: I): D
    deferred
    end

  key_in_env (env: E; id: I): D
    deferred
    end

  add_idn (env: E; id: I; key: D): BOOLEAN
    deferred
    end

  define_idn (env: E; id: I): D
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (id: I; key: D)
    local
      env: E
    do
      env := new_env
      check
        key_in_scope (env, id) /~ key
      end
    end

  frozen a_2 (env: E; id: I; key: D)
    do
      new_scope (env)
    ensure
      key_in_scope (env, id) /~ key
    end

  frozen a_3_1 (env: E; id: I; key: D)
    do
      add (env, id, key)
    ensure
      key_in_scope (env, id) ~ key
    end

  frozen a_3_2 (env: E; id_1, id_2: I; key, old_key_in_scope: D)
    require
```

```
      id_1 /~ id_2
      key_in_scope (env, id_2) ~ old_key_in_scope
    do
      add (env, id_1, key)
    ensure
      key_in_scope (env, id_2) ~ old_key_in_scope
    end

  frozen a_4 (id: I; key: D)
    local
      env: E
    do
      env := new_env
      check
        key_in_env (env, id) /~ key
      end
    end

  frozen a_5 (env: E; id: I; old_key_in_env: D)
    require
      key_in_env (env, id) ~ old_key_in_env
    do
      new_scope (env)
    ensure
      key_in_env (env, id) ~ old_key_in_env
    end

  frozen a_6_1 (env: E; id: I; key: D)
    do
      add (env, id, key)
    ensure
      key_in_env (env, id) ~ key
    end

  frozen a_6_2 (env: E; id_1, id_2: I; key, old_key_in_env: D)
    require
      id_1 /~ id_2
      key_in_env (env, id_2) ~ old_key_in_env
    do
      add (env, id_1, key)
    ensure
      key_in_env (env, id_2) ~ old_key_in_env
    end

feature
  -- Well-definedness axioms.

  frozen new_env_well_defined
    local
      env_1, env_2: E
    do
      env_1 := new_env
      env_2 := new_env
      check
        env_1 ≠ env_2
      end
      check
```

```
        env_1 ~ env_2
      end
    end

  frozen new_scope_well_defined (env_1, env_2: E)
    require
      env_1 ~ env_2
    do
      new_scope (env_1)
      new_scope (env_2)
    ensure
      env_1 ~ env_2
    end

  frozen add_well_defined (env_1, env_2: E; id: I; key: D)
    require
      env_1 ~ env_2
    do
      add (env_1, id, key)
      add (env_2, id, key)
    ensure
      env_1 ~ env_2
    end

  frozen key_in_scope_well_defined (env_1, env_2: E; id: I)
    require
      env_1 ~ env_2
    do
    ensure
      key_in_scope (env_1, id) ~ key_in_scope (env_2, id)
    end

  frozen key_in_env_well_defined (env_1, env_2: E; id: I)
    require
      env_1 ~ env_2
    do
    ensure
      key_in_env (env_1, id) ~ key_in_env (env_2, id)
    end

end
```

B.11 Equality

```
note
  description: "Reusable abstract data type specification of a type with equality."
  description: "Found in Wikipedia:"
  EIS: "src= http://tinyurl.com/pfafsvd"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yxb98wrq"

deferred class
  EQUALITY_ADT [G]
  -- Elements of ‘‘G’’ form an equivalence relation.

feature
  -- Abstract data type axioms.
  frozen equality_reflexivity (v: G)
    do
    ensure
      v ~ v
    end

  frozen equality_commutativity (v_1, v_2: G)
    require
      v_1 ~ v_2
    do
    ensure
      v_2 ~ v_1
    end

  frozen equality_transitivity (v_1, v_2, v_3: G)
    require
      v_1 ~ v_2
      v_2 ~ v_3
    do
    ensure
      v_1 ~ v_3
    end

end
```

B.12 File

```
note
  description: "Reusable abstract data type specification of file."
  description: "Found in ``The design of data type specifications'' by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y5phfw2h"

deferred class
  FILE_ADT [F, R]
  -- Files ``F'' contain records ``R''.

inherit

  EQUALITY_ADT [F]

feature
  -- Deferred definitions.

  empty_file: F
    deferred
    end

  write (file: F; record: R)
    deferred
    end

  skip (file: F; gap: INTEGER)
    deferred
    end
  reset (file: F)
    deferred
    end

  is_eof (file: F): BOOLEAN
    deferred
    end

  read (file: F): R
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (gap: INTEGER)
    local
      file_1, file_2: F
    do
      file_1 := empty_file
      file_2 := empty_file
      skip (file_1, gap)
      check
        file_1 ~ file_2
      end
    end

  frozen a_2 (file_1, file_2: F; gap_1, gap_2: INTEGER)
    require
      file_1 ~ file_2
    do
      skip (file_1, gap_1)
      skip (file_1, gap_2)
      skip (file_2, gap_1 + gap_2)
    ensure
      file_1 ~ file_2
    end

  frozen a_3
    local
      file_1, file_2: F
    do
      file_1 := empty_file
      file_2 := empty_file
      reset (file_1)
      check
        file_1 ~ file_2
      end
    end

  frozen a_4 (file_1, file_2: F; record: R)
    require
      file_1 ~ file_2
    do
      write (file_1, record)
      reset (file_1)
      write (file_2, record)
      skip (file_2, 0)
    ensure
      file_1 ~ file_2
    end

  frozen a_5 (file_1, file_2: F; record: R; gap: INTEGER)
    require
      file_1 ~ file_2
    do
      write (file_1, record)
      skip (file_1, gap)
      reset (file_1)
      write (file_2, record)
      skip (file_2, 0)
    ensure
      file_1 ~ file_2
    end

  frozen a_6
    local
      file: F
    do
      file := empty_file
      check
        is_eof (file)
      end
    end

  frozen a_7 (file: F; record: R)
    do
      write (file, record)
    ensure
      is_eof (file)
    end

  frozen a_8 (file: F; record: R)
    do
      write (file, record)
      skip (file, 0)
    ensure
      not is_eof (file)
    end

  frozen a_9 (file_1, file_2: F; record: R; gap: INTEGER)
    require
      gap /~ 0
      file_1 ~ file_2
    do
      write (file_1, record)
      skip (file_1, gap)
      skip (file_2, gap − 1)
    ensure
      file_1 ~ file_2
    end

  frozen a_10 (file_1, file_2: F; record: R; gap: INTEGER)
```

```
    require
      file_1 ~ file_2
    do
      write (file_1, record)
      skip (file_1, gap)
      skip (file_2, gap)
      check
        assume: is_eof (file_2)
      end
    ensure
      read (file_1) ~ record
    end

  frozen a_11 (file_1, file_2: F; record: R; gap: INTEGER)
    require
      file_1 ~ file_2
    do
      write (file_1, record)
      skip (file_1, gap)
      skip (file_2, gap)
      check
        assume: not is_eof (file_2)
      end
    ensure
      read (file_1) ~ read (file_2)
    end

  frozen a_12 (file_1, file_2, file_3: F; record_1, record_2: R; gap: INTEGER)
    require
      file_1 ~ file_2
      file_2 ~ file_3
    do
      write (file_1, record_1)
      skip (file_1, gap)
      write (file_1, record_2)
      skip (file_2, gap)
      check
        assume: is_eof (file_2)
      end
      write (file_3, record_2)
    ensure
      file_1 ~ file_3
    end

  frozen a_13 (file_1, file_2: F; record_1, record_2: R; gap: INTEGER)
    require
      file_1 ~ file_2
    do
      write (file_1, record_1)
      skip (file_1, gap)
      write (file_1, record_2)
      skip (file_2, gap)
      check
        assume: not is_eof (file_2)
      end
      write (file_2, record_2)
    ensure
```

```
      file_1 ~ file_2
    end

feature
  -- Well-definedness axioms.

  frozen empty_file_well_defined
    local
      file_1, file_2: F
    do
      file_1 := empty_file
      file_2 := empty_file
      check
        file_1 ≠ file_2
      end
      check
        file_1 ~ file_2
      end
    end

  frozen write_well_defined (file_1, file_2: F; record: R)
    require
      file_1 ~ file_2
    do
      write (file_1, record)
      write (file_2, record)
    ensure
      file_1 ~ file_2
    end

  frozen skip_well_defined (file_1, file_2: F; gap: INTEGER)
    require
      file_1 ~ file_2
    do
      skip (file_1, gap)
      skip (file_2, gap)
    ensure
      file_1 ~ file_2
    end

  frozen reset_well_defined (file_1, file_2: F)
    require
      file_1 ~ file_2
    do
      reset (file_1)
      reset (file_2)
    ensure
      file_1 ~ file_2
    end

  frozen is_eof_well_defined (file_1, file_2: F): BOOLEAN
    require
      file_1 ~ file_2
    do
    ensure
      is_eof (file_1) ~ is_eof (file_2)
    end
  frozen read_well_defined (file_1, file_2: F)
    require
      file_1 ~ file_2
    do
    ensure
      read (file_1) ~ read (file_2)
    end

end
```

B.13 Graph

```
note
  description: "Reusable abstract data type specification of graph."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y5y6l6ax"

deferred class
  GRAPH_ADT [G, N, E, SN, SE, ES →EDGE_ADT [E, N], SNS →SET_WITH_ISEMPTYSET_ADT [SN, N], SES →
        SET_WITH_ISEMPTYSET_ADT [SE, E]]
  -- Graphs ‘‘G’’ contain edges ‘‘E’’ connecting nodes ‘‘N’’
  -- conforming to the ‘‘EDGE_ADT’’ specification.
  -- They depend on sets ‘‘SE’’ of edges ‘‘E’’ conforming to the
  -- ‘‘SET_WITH_ISEMPTYSET_ADT’’ specification.
  -- They depend on sets ‘‘SN’’ of nodes ‘‘N’’ conforming to the
  -- ‘‘SET_WITH_ISEMPTYSET_ADT’’ specification.

inherit

  EQUALITY_ADT [G]

feature
  -- Deferred definitions.

  empty_graph: G
    deferred
    end

  add_node (graph: G; node: N)
    deferred
    end

  add_edge (graph: G; edge: E)
    deferred
    end

  nodes (graph: G): SN
    deferred
    end

  edges (graph: G): SE
    deferred
    end
  adjac (graph: G; node: N): SN
    deferred
    end

  nod_out (graph: G; node: N)
    deferred
    end

  edge_out (graph: G; edge: E)
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1
    local
      graph: G
      nodes_set: SN
    do
      graph := empty_graph
      nodes_set := ({SNS}).default.empty_set
      check
        nodes (graph) ~ nodes_set
      end
    end

  frozen a_2 (graph: G; node: N; nodes_set: SN)
    require
      nodes (graph) ~ nodes_set
    do
      add_node (graph, node)
      ({SNS}).default.insert (nodes_set, node)
    ensure
      nodes (graph) ~ nodes_set
    end

  frozen a_3 (graph: G; node_1, node_2: N; edge: E; nodes_set: SN)
    require
      ({ES}).default.rel (node_1, node_2) ~ edge
      nodes (graph) ~ nodes_set
    do
      add_edge (graph, edge)
      ({SNS}).default.insert (nodes_set, node_1)
      ({SNS}).default.insert (nodes_set, node_2)
    ensure
      nodes (graph) ~ nodes_set
    end

  frozen a_4
    local
      graph: G
      edges_set: SE
    do
      graph := empty_graph
      edges_set := ({SES}).default.empty_set
      check
      edges (graph) ~ edges_set
    end
  end

frozen a_5 (graph: G; node: N; old_edges: SE)
  require
    edges (graph) ~ old_edges
  do
    add_node (graph, node)
  ensure
    edges (graph) ~ old_edges
  end

frozen a_6 (graph: G; node_1, node_2: N; edge: E; old_edges: SE)
  require
    ({ES}).default.rel (node_1, node_2) ~ edge
    edges (graph) ~ old_edges
  do
    add_edge (graph, edge)
    ({SES}).default.insert (old_edges, edge)
  ensure
    edges (graph) ~ old_edges
  end

frozen a_7 (node: N)
  local
    graph: G
    nodes_set: SN
  do
    graph := empty_graph
    nodes_set := ({SNS}).default.empty_set
    check
      adjac (graph, node) ~ nodes_set
    end
  end

frozen a_8 (graph: G; node_1, node_2: N; old_adjac: SN)
  require
    adjac (graph, node_2) ~ old_adjac
  do
    add_node (graph, node_1)
  ensure
    adjac (graph, node_2) ~ old_adjac
  end

frozen a_9 (graph: G; node_1, node_2: N; edge: E; old_adjac: SN)
  require
    ({ES}).default.rel (node_1, node_2) ~ edge
    adjac (graph, node_1) ~ old_adjac
  do
    add_edge (graph, edge)
    ({SNS}).default.insert (old_adjac, node_2)
  ensure
    adjac (graph, node_1) ~ old_adjac
  end

frozen a_10 (graph: G; node_1, node_2: N; edge: E; old_adjac: SN)
```

```
  require
    ({ES}).default.rel (node_1, node_2) ~ edge
    adjac (graph, node_2) ~ old_adjac
  do
    add_edge (graph, edge)
    ({SNS}).default.insert (old_adjac, node_2)
  ensure
    adjac (graph, node_2) ~ old_adjac
  end

frozen a_11 (graph: G; node_1, node_2: N; edge: E; old_adjac: SN)
  require
    ({ES}).default.rel (node_1, node_2) ~ edge
    adjac (graph, node_2) ~ old_adjac
  do
    add_edge (graph, edge)
    ({SNS}).default.insert (old_adjac, node_1)
  ensure
    adjac (graph, node_2) ~ old_adjac
  end

frozen a_12 (graph: G; node_1, node_2, node_3: N; edge: E; old_adjac: SN)
  require
    node_3 /~ node_1
    node_3 /~ node_2;
    ({ES}).default.rel (node_1, node_2) ~ edge
    adjac (graph, node_3) ~ old_adjac
  do
    add_edge (graph, edge)
    ({SNS}).default.insert (old_adjac, node_1)
  ensure
    adjac (graph, node_3) ~ old_adjac
  end

frozen a_13 (node: N)
  local
    graph_1, graph_2: G
  do
    graph_1 := empty_graph
    graph_2 := empty_graph
    nod_out (graph_1, node)
    check
      graph_1 ~ graph_2
    end
  end

frozen a_14 (graph_1, graph_2: G; node: N)
  require
    graph_1 ~ graph_2
  do
    add_node (graph_1, node)
    nod_out (graph_1, node)
    nod_out (graph_2, node)
  ensure
    graph_1 ~ graph_2
  end
```

```
frozen a_15 (graph_1, graph_2: G; node_1, node_2: N)
  require
    node_1 /~ node_2
    graph_1 ~ graph_2
  do
    add_node (graph_1, node_1)
    nod_out (graph_1, node_2)
    nod_out (graph_2, node_2)
    add_node (graph_2, node_1)
  ensure
    graph_1 ~ graph_2
  end

frozen a_16 (graph_1, graph_2: G; node_1, node_2: N; edge: E)
  require
    ({ES}).default.rel (node_1, node_2) ~ edge
    graph_1 ~ graph_2
  do
    add_edge (graph_1, edge)
    nod_out (graph_1, node_1)
    nod_out (graph_2, node_1)
  ensure
    graph_1 ~ graph_2
  end

frozen a_17 (graph_1, graph_2: G; node_1, node_2: N; edge: E)
  require
    ({ES}).default.rel (node_1, node_2) ~ edge
    graph_1 ~ graph_2
  do
    add_edge (graph_1, edge)
    nod_out (graph_1, node_2)
    nod_out (graph_2, node_2)
  ensure
    graph_1 ~ graph_2
  end

frozen a_18 (graph_1, graph_2: G; node_1, node_2, node_3: N; edge: E)
  require
    node_3 /~ node_1
    node_3 /~ node_2;
    ({ES}).default.rel (node_1, node_2) ~ edge
    graph_1 ~ graph_2
  do
    add_edge (graph_1, edge)
    nod_out (graph_1, node_3)
    nod_out (graph_2, node_3)
    add_edge (graph_2, edge)
  ensure
    graph_1 ~ graph_2
  end

frozen a_19 (edge: E)
  local
    graph_1, graph_2: G
  do
    graph_1 := empty_graph
```

```
      graph_2 := empty_graph
      edge_out (graph_1, edge)
      check
        graph_1 ~ graph_2
      end
    end

  frozen a_20 (graph_1, graph_2: G; node: N; edge: E)
    require
      graph_1 ~ graph_2
    do
      add_node (graph_1, node)
      edge_out (graph_1, edge)
      edge_out (graph_2, edge)
      add_node (graph_2, node)
    ensure
      graph_1 ~ graph_2
    end

  frozen a_21 (graph_1, graph_2: G; edge: E)
    require
      graph_1 ~ graph_2
    do
      add_edge (graph_1, edge)
      edge_out (graph_1, edge)
    ensure
      graph_1 ~ graph_2
    end

  frozen a_22 (graph_1, graph_2: G; edge_1, edge_2: E)
    require
      edge_1 /~ edge_2
      graph_1 ~ graph_2
    do
      add_edge (graph_1, edge_1)
      edge_out (graph_1, edge_2)
      edge_out (graph_2, edge_2)
      add_edge (graph_2, edge_1)
    ensure
      graph_1 ~ graph_2
    end

feature
  -- Well-definedness axioms.

  frozen empty_graph_well_defined
    local
      graph_1, graph_2: G
    do
      graph_1 := empty_graph
      graph_2 := empty_graph
      check
        graph_1 ≠ graph_2
      end
      check
        graph_1 ~ graph_2
      end
  end

frozen add_node_well_defined (graph_1, graph_2: G; node: N)
  require
    graph_1 ~ graph_2
  do
    add_node (graph_1, node)
    add_node (graph_2, node)
  ensure
    graph_1 ~ graph_2
  end

frozen add_edge_well_defined (graph_1, graph_2: G; edge: E)
  require
    graph_1 ~ graph_2
  do
    add_edge (graph_1, edge)
    add_edge (graph_2, edge)
  ensure
    graph_1 ~ graph_2
  end

frozen nodes_well_defined (graph_1, graph_2: G)
  require
    graph_1 ~ graph_2
  do
  ensure
    nodes (graph_1) ~ nodes (graph_2)
  end

frozen edges_well_defined (graph_1, graph_2: G)
  require
    graph_1 ~ graph_2
  do
  ensure
    edges (graph_1) ~ edges (graph_2)
  end

frozen adjac_well_defined (graph_1, graph_2: G; node: N)
  require
    graph_1 ~ graph_2
  do
  ensure
    adjac (graph_1, node) ~ adjac (graph_2, node)
  end

frozen nod_out_well_defined (graph_1, graph_2: G; node: N)
  require
    graph_1 ~ graph_2
  do
    nod_out (graph_1, node)
    nod_out (graph_2, node)
  ensure
    graph_1 ~ graph_2
  end

frozen edge_out_well_defined (graph_1, graph_2: G; edge: E)
```

```
    require
      graph_1 ~ graph_2
    do
      edge_out (graph_1, edge)
      edge_out (graph_2, edge)
    ensure
      graph_1 ~ graph_2
    end

end
```

B.14 Library

```
note
  description: "Reusable abstract data type specification of library."
  description: "Found in ‘‘Requirements engineering: From system goals to UML models to software.’’
        by van Lamsweerde:"
  EIS: "src= http://tinyurl.com/yxd3zxd2"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y4jnocr4"

deferred class
  LIBRARY_ADT [L, B]
  -- Libraries ‘‘L’’ contain books ‘‘B’’.

inherit

  EQUALITY_ADT [L]

feature
  -- Deferred definitions.

  empty_lib: L
    deferred
    end

  add_copy (l: L; b: B)
    deferred
    end

  remove_copy (l: L; b: B)
    deferred
    end

  check_out (l: L; b: B)
    deferred
    end

  return (l: L; b: B)
    deferred
    end

  copy_exists (l: L; b: B): BOOLEAN
    deferred
    end

  copy_borrowed (l: L; b: B): BOOLEAN
```

```
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (lib: L; bc: B)
    local
      new_lib: L
    do
      new_lib := empty_lib
      check
        not copy_exists (new_lib, bc)
      end
    end

  frozen a_2 (lib: L; bc: B)
    do
      add_copy (lib, bc)
    ensure
      copy_exists (lib, bc)
    end

  frozen a_3 (lib: L; bc_1, bc_2: B; bc_2_exists: BOOLEAN)
    require
      bc_1 /~ bc_2
      copy_exists (lib, bc_2) ~ bc_2_exists
    do
      add_copy (lib, bc_1)
    ensure
      copy_exists (lib, bc_2) ~ bc_2_exists
    end

  frozen a_4 (lib: L; bc_1, bc_2: B; bc_2_exists: BOOLEAN)
    require
      copy_exists (lib, bc_2) ~ bc_2_exists
      not copy_borrowed (lib, bc_1)
    do
      check_out (lib, bc_1)
    ensure
      copy_exists (lib, bc_2) ~ bc_2_exists
    end

  frozen a_5 (lib: L; bc: B)
    local
      new_lib: L
    do
      new_lib := empty_lib
      check
        not copy_borrowed (lib, bc)
      end
    end

  frozen a_6 (lib: L; bc_1, bc_2: B; bc_2_borrowed: BOOLEAN)
    require
      copy_borrowed (lib, bc_2) ~ bc_2_borrowed
    do
```

```
      add_copy (lib, bc_1)
    ensure
      copy_borrowed (lib, bc_2) ~ bc_2_borrowed
    end

  frozen a_7 (lib: L; bc: B)
    require
      not copy_borrowed (lib, bc)
    do
      check_out (lib, bc)
    ensure
      copy_borrowed (lib, bc)
    end

  frozen a_8 (lib: L; bc_1, bc_2: B; bc_2_borrowed: BOOLEAN)
    require
      bc_1 /~ bc_2
      copy_borrowed (lib, bc_2) ~ bc_2_borrowed
      not copy_borrowed (lib, bc_1)
    do
      check_out (lib, bc_1)
    ensure
      copy_borrowed (lib, bc_2) ~ bc_2_borrowed
    end

  frozen a_9 (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
    do
      add_copy (lib_1, bc)
      remove_copy (lib_1, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen a_10 (lib_1, lib_2: L; bc_1, bc_2: B)
    require
      lib_1 ≠ lib_2
      lib_1 ~ lib_2
      bc_1 /~ bc_2
      copy_exists (lib_1, bc_2)
      copy_exists (lib_2, bc_2)
    do
      add_copy (lib_1, bc_1)
      remove_copy (lib_1, bc_2)
      remove_copy (lib_2, bc_2)
      add_copy (lib_2, bc_1)
    ensure
      lib_1 ~ lib_2
    end

  frozen a_11 (lib_1, lib_2: L; bc: B)
    require
      lib_1 ≠ lib_2
      lib_1 ~ lib_2
      not copy_borrowed (lib_1, bc)
      copy_exists (lib_1, bc)
      copy_exists (lib_2, bc)
    do
      check_out (lib_1, bc)
      remove_copy (lib_1, bc)
      remove_copy (lib_2, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen a_12 (lib_1, lib_2: L; bc_1, bc_2: B)
    require
      lib_1 ≠ lib_2
      lib_1 ~ lib_2
      bc_1 /~ bc_2
      not copy_borrowed (lib_1, bc_1)
      copy_exists (lib_1, bc_2)
      copy_exists (lib_2, bc_2)
      not copy_borrowed (lib_2, bc_1)
    do
      check_out (lib_1, bc_1)
      remove_copy (lib_1, bc_2)
      remove_copy (lib_2, bc_2)
      check_out (lib_2, bc_1)
    ensure
      lib_1 ~ lib_2
    end

  frozen a_13 (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
      not copy_borrowed (lib_1, bc)
    do
      check_out (lib_1, bc)
      return (lib_1, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen a_14 (lib_1, lib_2: L; bc_1, bc_2: B)
    require
      lib_1 ≠ lib_2
      lib_1 ~ lib_2
      bc_1 /~ bc_2
      not copy_borrowed (lib_1, bc_1)
      copy_borrowed (lib_1, bc_2)
      copy_borrowed (lib_2, bc_2)
      not copy_borrowed (lib_2, bc_1)
    do
      check_out (lib_1, bc_1)
      return (lib_1, bc_2)
      return (lib_2, bc_2)
      check_out (lib_2, bc_1)
    ensure
      lib_1 ~ lib_2
    end

  frozen a_15 (lib_1, lib_2: L; bc_1, bc_2: B)
```

```
    require
      lib_1 ≠ lib_2
      lib_1 ~ lib_2
      bc_1 /~ bc_2
      copy_borrowed (lib_1, bc_2)
      copy_borrowed (lib_2, bc_2)
    do
      add_copy (lib_1, bc_1)
      return (lib_1, bc_2)
      return (lib_2, bc_2)
      add_copy (lib_2, bc_1)
    ensure
      lib_1 ~ lib_2
    end

feature
  -- Well-definedness axioms.

  frozen empty_lib_well_defined
    local
      lib_1, lib_2: L
    do
      lib_1 := empty_lib
      lib_2 := empty_lib
      check
        lib_1 ≠ lib_2
      end
      check
        lib_1 ~ lib_2
      end
    end

  frozen add_copy_well_defined (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
    do
      add_copy (lib_1, bc)
      add_copy (lib_2, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen remove_copy_well_defined (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
      lib_1 ≠ lib_2
      copy_exists (lib_1, bc)
      copy_exists (lib_2, bc)
    do
      remove_copy (lib_1, bc)
      remove_copy (lib_2, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen check_out_well_defined (lib_1, lib_2: L; bc: B)
    require
```

```
      lib_1 ~ lib_2
      lib_1 ≠ lib_2
      not copy_borrowed (lib_1, bc)
      not copy_borrowed (lib_2, bc)
    do
      check_out (lib_1, bc)
      check_out (lib_2, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen return_well_defined (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
      lib_1 ≠ lib_2
      copy_borrowed (lib_1, bc)
      copy_borrowed (lib_2, bc)
    do
      return (lib_1, bc)
      return (lib_2, bc)
    ensure
      lib_1 ~ lib_2
    end

  frozen copy_exists_well_defined (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
    do
    ensure
      copy_exists (lib_1, bc) ~ copy_exists (lib_2, bc)
    end

  frozen copy_borrowed_well_defined (lib_1, lib_2: L; bc: B)
    require
      lib_1 ~ lib_2
    do
    ensure
      copy_borrowed (lib_1, bc) ~ copy_borrowed (lib_2, bc)
    end

end
```

B.15 List

```
note
  description: "Reusable abstract data type specification of list."
  description: "Found in Wikipedia:"
  EIS: "src= http://tinyurl.com/yxu9yze9"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yym548bu"

deferred class
  LIST_ADT [L, E]
  -- Lists ‘‘L’’ contain elements of ‘‘E’’.

inherit
```

```
  EQUALITY_ADT [L]

feature
  -- Deferred definitions.

  nil: L
    deferred
    end

  cons (l: L; e: E)
    deferred
    end

  first (l: L): E
    deferred
    end

  rest (l: L): L
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (l: L; e: E)
    do
      cons (l, e)
    ensure
      first (l) ~ e
    end

  frozen a_2 (l: L; e: E; old_l: L)
    require
      l ~ old_l
    do
      cons (l, e)
    ensure
      rest (l) ~ old_l
    end

feature
  -- Well-definedness axioms.

  frozen nil_well_defined
    local
      list_1, list_2: L
    do
      list_1 := nil
      list_2 := nil
      check
        assert: list_1 ≠ list_2
      end
      check
        assert: list_1 ~ list_2
      end
    end
  frozen cons_well_defined (list_1, list_2: L; element: E)
    require
      list_1 ~ list_2
    do
      cons (list_1, element)
      cons (list_2, element)
    ensure
      list_1 ~ list_2
    end

  frozen first_well_defined (list_1, list_2: L)
    require
      list_1 ~ list_2
    do
    ensure
      first (list_1) ~ first (list_2)
    end

  frozen rest_well_defined (list_1, list_2: L)
    require
      list_1 ~ list_2
    do
    ensure
      rest (list_1) ~ rest (list_2)
    end

end
```

B.16 Mapping

```
note
  description: "Reusable abstract data type specification of mapping."
  description: "Found in ‘‘Abstract Data Types and Software Validation’’ by Guttag, Horowitz and
        Musser:"
  EIS: "src= https://pdfs.semanticscholar.org/372d/4f331d0a6cd5fb4ee0c04d4a0753b8eb659f.pdf"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/yxnkehv8"

deferred class
  MAPPING_ADT [M, D, R]
  -- Mappings ‘‘M’’ map domains ‘‘D’’ to ranges ‘‘R’’.

inherit

  EQUALITY_ADT [M]

feature
  -- Deferred definitions.

  new_map: M
    deferred
    end

  def_map (map: M; dval: D; rval: R)
    deferred
    end
  ev_map (map: M; dval: D): R
    deferred
    end

  is_defined (map: M; dval: D): BOOLEAN
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (dval: D; rval: R)
    local
      map: M
    do
      map := new_map
      check
        ev_map (map, dval) /~ rval
      end
    end

  frozen a_2 (map: M; dval: D; rval: R)
    do
      def_map (map, dval, rval)
    ensure
      ev_map (map, dval) ~ rval
    end

  frozen a_3 (map: M; dval_1, dval_2: D; rval, old_ev_map: R)
    require
      ev_map (map, dval_2) ~ old_ev_map
      dval_1 /~ dval_2
    do
      def_map (map, dval_1, rval)
    ensure
      ev_map (map, dval_2) ~ old_ev_map
    end

  frozen a_4 (dval: D)
    local
      map: M
    do
      map := new_map
      check
        not is_defined (map, dval)
      end
    end

  frozen a_5 (map: M; dval: D; rval: R)
    do
      def_map (map, dval, rval)
    ensure
      is_defined (map, dval)
    end

  frozen a_6 (map: M; dval_1, dval_2: D; rval: R; old_is_defined: BOOLEAN)
    require
```

```
      is_defined (map, dval_2) ~ old_is_defined
      dval_1 /~ dval_2
    do
      def_map (map, dval_1, rval)
    ensure
      is_defined (map, dval_2) ~ old_is_defined
    end

feature
  -- Well-definedness axioms.

  frozen new_map_well_defined
    local
      map_1, map_2: M
    do
      map_1 := new_map
      map_2 := new_map
      check
        map_1 ≠ map_2
      end
      check
        map_1 ~ map_2
      end
    end

  frozen def_map_well_defined (map_1, map_2: M; dval: D; rval: R)
    require
      map_1 ~ map_2
    do
      def_map (map_1, dval, rval)
      def_map (map_2, dval, rval)
    ensure
      map_1 ~ map_2
    end

  frozen ev_map_well_defined (map_1, map_2: M; dval: D)
    require
      map_1 ~ map_2
    do
    ensure
      ev_map (map_1, dval) ~ ev_map (map_2, dval)
    end

  frozen is_defined_well_defined (map_1, map_2: M; dval: D)
    require
      map_1 ~ map_2
    do
    ensure
      is_defined (map_1, dval) ~ is_defined (map_2, dval)
    end

end
```

B.17 Polynomial

```
note
```

```
  description: "Reusable abstract data type specification of polynomial."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y5u8hubc"

deferred class
  POLYNOMIAL_ADT [P, C, CS →COMMUTATIVE_RING_ADT [C]]
  -- Polynomials ‘‘P’’ have coefficients from the commutative ring ‘‘C’’.

inherit

  COMMUTATIVE_RING_ADT [P]
    rename
      sum as add,
      product as mult
    end

feature
  -- Deferred definitions.

  add_term (polynomial: P; coefficient: C; exponent: INTEGER)
    deferred
    end

  rem_term (polynomial: P; exponent: INTEGER)
    deferred
    end

  mult_term (polynomial: P; coefficient: C; exponent: INTEGER)
    deferred
    end

  reductum (polynomial: P)
    deferred
    end

  is_zero (polynomial: P): BOOLEAN
    deferred
    end

  coef (polynomial: P; exponent: INTEGER): C
    deferred
    end

  degree (polynomial: P): INTEGER
    deferred
    end

  ldcf (polynomial: P): C
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_11 (exponent: INTEGER)
    local
    polynomial_1, polynomial_2: P
  do
    polynomial_1 := zero
    polynomial_2 := zero
    rem_term (polynomial_1, exponent)
    check
      polynomial_1 ~ polynomial_2
    end
  end

frozen a_12 (polynomial_1, polynomial_2: P; coefficient: C; exponent: INTEGER)
  require
    polynomial_1 ~ polynomial_2
  do
    add_term (polynomial_1, coefficient, exponent)
    rem_term (polynomial_1, exponent)
    rem_term (polynomial_2, exponent)
  ensure
    polynomial_1 ~ polynomial_2
  end

frozen a_13 (polynomial_1, polynomial_2: P; coefficient: C; exponent_1, exponent_2: INTEGER)
  require
    exponent_1 /~ exponent_2
    polynomial_1 ~ polynomial_2
  do
    add_term (polynomial_1, coefficient, exponent_1)
    rem_term (polynomial_1, exponent_2)
    rem_term (polynomial_2, exponent_2)
    add_term (polynomial_2, coefficient, exponent_1)
  ensure
    polynomial_1 ~ polynomial_2
  end

frozen a_14 (coefficient: C; exponent: INTEGER)
  local
    polynomial_1, polynomial_2: P
  do
    polynomial_1 := zero
    polynomial_2 := zero
    mult_term (polynomial_1, coefficient, exponent)
    check
      polynomial_1 ~ polynomial_2
    end
  end

frozen a_15 (polynomial_1, polynomial_2: P; coefficient_1, coefficient_2: C; exponent_1,
     exponent_2: INTEGER)
  require
    polynomial_1 ~ polynomial_2
  do
    add_term (polynomial_1, coefficient_1, exponent_1)
    mult_term (polynomial_1, coefficient_2, exponent_2)
    mult_term (polynomial_2, coefficient_2, exponent_2)
    add_term (polynomial_2, ({CS}).default.product (coefficient_1, coefficient_2), exponent_1
          + exponent_2)
  ensure
```

```
      polynomial_1 ~ polynomial_2
    end

  frozen a_16 (polynomial: P)
    local
      zero_p: P
    do
      zero_p := zero
      check
        add (polynomial, zero_p) ~ polynomial
      end
    end

  frozen a_17 (p, q, s: P; d: C; f: INTEGER)
    require
      add (p, q) ~ s
    do
      add_term (q, d, f)
      add_term (s, d, f)
    ensure
      add (p, q) ~ s
    end

  frozen a_18 (polynomial: P)
      -- MULT(p,ZERO) = ZERO
    local
      zero_p: P
    do
      zero_p := zero
      check
        mult (polynomial, zero_p) ~ zero_p
      end
    end

  frozen a_19 (p_1, p_2, p_3, q_1, q_2: P; d: C; f: INTEGER)
    require
      p_1 ~ p_2
      q_1 ~ q_2
    do
      add_term (q_1, d, f)
      mult_term (p_2, d, f)
    ensure
      mult (p_1, q_1) ~ add (mult (p_1, q_2), p_2)
    end

  frozen a_20 (polynomial_1, polynomial_2: P)
    require
      polynomial_1 ~ polynomial_2
    do
      reductum (polynomial_1)
      rem_term (polynomial_2, degree (polynomial_2))
    ensure
      polynomial_1 ~ polynomial_2
    end

  frozen a_21
    local
```

```
      polynomial: P
    do
      polynomial := zero
      check
        is_zero (polynomial)
      end
    end

  frozen a_22 (polynomial_1, polynomial_2: P; coefficient: C; exponent: INTEGER)
    require
      coef (polynomial_1, exponent) ~ ({CS}).default.additive_inverse (coefficient)
      polynomial_1 ~ polynomial_2
    do
      add_term (polynomial_1, coefficient, exponent)
      rem_term (polynomial_2, exponent)
    ensure
      is_zero (polynomial_1) ~ is_zero (polynomial_2)
    end

  frozen a_23 (polynomial: P; coefficient: C; exponent: INTEGER)
    require
      coef (polynomial, exponent) /~ ({CS}).default.additive_inverse (coefficient)
    do
      add_term (polynomial, coefficient, exponent)
    ensure
      not is_zero (polynomial)
    end

  frozen a_24 (exponent: INTEGER)
    local
      polynomial: P
    do
      polynomial := zero
      check
        coef (polynomial, exponent) ~ ({CS}).default.zero
      end
    end

  frozen a_25 (polynomial: P; coefficient: C; exponent: INTEGER; old_coefficient: C)
    require
      coef (polynomial, exponent) ~ old_coefficient
    do
      add_term (polynomial, coefficient, exponent)
    ensure
      coef (polynomial, exponent) ~ ({CS}).default.sum (coefficient, old_coefficient)
    end

  frozen a_26 (polynomial: P; coefficient: C; exponent_1, exponent_2: INTEGER; old_coefficient: C
      )
    require
      coef (polynomial, exponent_2) ~ old_coefficient
    do
      add_term (polynomial, coefficient, exponent_1)
    ensure
      coef (polynomial, exponent_2) ~ old_coefficient
    end
```

```
frozen a_27
  local
    polynomial: P
  do
    polynomial := zero
    check
      degree (polynomial) ~ 0
    end
  end

frozen a_28 (polynomial: P; coefficient: C; exponent: INTEGER)
  require
    exponent > degree (polynomial)
  do
    add_term (polynomial, coefficient, exponent)
  ensure
    degree (polynomial) ~ exponent
  end

frozen a_29 (polynomial: P; coefficient: C; exponent: INTEGER; old_degree: INTEGER)
  require
    exponent <degree (polynomial)
    degree (polynomial) ~ old_degree
  do
    add_term (polynomial, coefficient, exponent)
  ensure
    degree (polynomial) ~ old_degree
  end

frozen a_30 (polynomial_1, polynomial_2: P; coefficient: C; exponent: INTEGER; old_degree:
     INTEGER)
  require
    exponent = degree (polynomial_1)
    coef (polynomial_1, exponent) ~ ({CS}).default.additive_inverse (coefficient)
    polynomial_1 ~ polynomial_2
  do
    add_term (polynomial_1, coefficient, exponent)
    reductum (polynomial_2)
  ensure
    degree (polynomial_1) ~ degree (polynomial_2)
  end

frozen a_31 (polynomial: P; coefficient: C; exponent: INTEGER; old_degree: INTEGER)
  require
    exponent = degree (polynomial)
    coef (polynomial, exponent) /~ ({CS}).default.additive_inverse (coefficient)
    degree (polynomial) ~ old_degree
  do
    add_term (polynomial, coefficient, exponent)
  ensure
    degree (polynomial) ~ old_degree
  end

frozen a_32 (polynomial: P)
  do
  ensure
    ldcf (polynomial) ~ coef (polynomial, degree (polynomial))
    end

feature
  -- Well-definedness axioms.

  frozen add_term_well_defined (polynomial_1, polynomial_2: P; coefficient: C; exponent:
        INTEGER)
    require
      polynomial_1 ~ polynomial_2
    do
      add_term (polynomial_1, coefficient, exponent)
      add_term (polynomial_2, coefficient, exponent)
    ensure
      polynomial_1 ~ polynomial_2
    end

  frozen rem_term_well_defined (polynomial_1, polynomial_2: P; exponent: INTEGER)
    require
      polynomial_1 ~ polynomial_2
    do
      rem_term (polynomial_1, exponent)
      rem_term (polynomial_2, exponent)
    ensure
      polynomial_1 ~ polynomial_2
    end

  frozen mult_term_well_defined (polynomial_1, polynomial_2: P; coefficient: C; exponent:
        INTEGER)
    require
      polynomial_1 ~ polynomial_2
    do
      mult_term (polynomial_1, coefficient, exponent)
      mult_term (polynomial_2, coefficient, exponent)
    ensure
      polynomial_1 ~ polynomial_2
    end

  frozen reductum_well_defined (polynomial_1, polynomial_2: P)
    require
      polynomial_1 ~ polynomial_2
    do
      reductum (polynomial_1)
      reductum (polynomial_2)
    ensure
      polynomial_1 ~ polynomial_2
    end

  frozen is_zero_well_defined (polynomial_1, polynomial_2: P)
    require
      polynomial_1 ~ polynomial_2
    do
    ensure
      is_zero (polynomial_1) ~ is_zero (polynomial_2)
    end

  frozen coef_well_defined (polynomial_1, polynomial_2: P; exponent: INTEGER)
    require
```

```
      polynomial_1 ~ polynomial_2
    do
    ensure
      coef (polynomial_1, exponent) ~ coef (polynomial_2, exponent)
    end

  frozen degree_well_defined (polynomial_1, polynomial_2: P)
    require
      polynomial_1 ~ polynomial_2
    do
    ensure
      degree (polynomial_1) ~ degree (polynomial_2)
    end

  frozen ldcf_well_defined (polynomial_1, polynomial_2: P)
    require
      polynomial_1 ~ polynomial_2
    do
    ensure
      ldcf (polynomial_1) ~ ldcf (polynomial_2)
    end

end
```

B.18 Queue

```
note
  description: "Reusable abstract data type specification of queue."
  description: "Found in ‘‘The Algebraic Specification of Abstract Data Types’’ by Guttag and Horning:
        "
  EIS: "src= https://link.springer.com/article/10.1007/BF00260922"
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  description: "Found in ‘‘Implementing Algebraically Specified Abstract Data Types in an imperative
        Programming Language ’’ by Thomas:"
  EIS: "src= http://www.dcs.gla.ac.uk/~muffy/papers/Tapsoft_87.pdf"
  description: "Found in ‘‘Abstract Data Types and the Development of Data Structures’’ by Guttag:"
  EIS: "src= http://tinyurl.com/y45o32hq"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y4qv86kz"

deferred class
  QUEUE_ADT [Q, T]
  -- Queues ‘‘Q’’ contain elements of ‘‘T’’.

inherit

  EQUALITY_ADT [Q]

feature
  -- Deferred definitions.

  newq: Q
    deferred
    end

  addq (q: Q; t: T)
```

```
    deferred
    end

  deleteq (q: Q)
    deferred
    end

  frontq (q: Q): T
    deferred
    end

  isnewq (q: Q): BOOLEAN
    deferred
    end

  size (q: Q): INTEGER
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_3_empty (q_1, q_2: Q)
    require
      q_1 ~ q_2
    local
      empty_: BOOLEAN
    do
      empty_ := isnewq (q_1)
    ensure
      q_1 ~ q_2
    end

  frozen a_3_size (q_1, q_2: Q)
    require
      q_1 ~ q_2
    local
      size_: INTEGER
    do
      size_ := size (q_1)
    ensure
      q_1 ~ q_2
    end

  frozen a_3_front (q_1, q_2: Q)
    require
      q_1 ~ q_2
    local
      front_: T
    do
      front_ := frontq (q_1)
    ensure
      q_1 ~ q_2
    end

  frozen a_4_if (q: Q)
    require
```

```
        size (q) ~ 0
      do
      ensure
        isnewq (q)
      end

    frozen a_4_only_if (q: Q)
      require
        isnewq (q)
      do
      ensure
        size (q) ~ 0
      end

    frozen a_5 (q: Q; n: INTEGER; t: T)
      require
        size (q) ~ n
      local
        i: INTEGER
      do
        addq (q, t)
        from
          i := 0
        until
          i ~ n
        loop
          deleteq (q)
          i := i + 1
        end
      ensure
        frontq (q) ~ t
      end

    frozen a_6 (q: Q; t: T; old_size: INTEGER)
      require
        size (q) ~ old_size
      do
        addq (q, t)
      ensure
        size (q) ~ old_size + 1
      end

    frozen a_7 (q: Q; t: T; old_size: INTEGER)
      require
        size (q) ~ old_size
        not isnewq (q)
      do
        deleteq (q)
      ensure
        size (q) ~ old_size − 1
      end

    frozen a_9 (q: Q; t: T)
      do
        addq (q, t)
      ensure
        not isnewq (q)
```

```
  end

frozen a_10
  local
    q: Q
  do
    q := newq
    check
      isnewq (q)
    end
  end

frozen a_11
  local
    q: Q
  do
    q := newq
    check
      size (q) ~ 0
    end
  end

frozen a_12
  local
    q_1, q_2: Q
  do
    q_1 := newq
    q_2 := newq
    deleteq (q_1)
    check
      q_1 ~ q_2
    end
  end

frozen a_13 (t: T)
  local
    q_1, q_2: Q
  do
    q_1 := newq
    q_2 := newq
    addq (q_1, t)
    deleteq (q_1)
    check
      q_1 ~ q_2
    end
  end

frozen a_14 (q_1, q_2: Q; t_1, t_2: T)
  require
    q_1 ~ q_2
    q_1 ≠ q_2
  do
    addq (q_1, t_1)
    addq (q_1, t_2)
    deleteq (q_1)
    addq (q_2, t_1)
    deleteq (q_2)
```

```
      addq (q_2, t_2)
    ensure
      q_1 ~ q_2
    end

  frozen a_15 (t: T)
    local
      q: Q
    do
      q := newq
      addq (q, t)
      check
        frontq (q) ~ t
      end
    end

  frozen a_16 (q_1, q_2: Q; t_1, t_2: T)
    require
      q_1 ~ q_2
      q_1 ≠ q_2
    do
      addq (q_1, t_1)
      addq (q_1, t_2)
      addq (q_2, t_1)
    ensure
      frontq (q_1) ~ frontq (q_2)
    end

  frozen a_17 (t: T)
    local
      q: Q
    do
      q := newq
      check
        frontq (q) /~ t
      end
    end

feature
  -- Well-definedness axioms.

  frozen new_well_defined
    local
      q_1, q_2: Q
    do
      q_1 := newq
      q_2 := newq
      check
        q_1 ~ q_2
      end
    end

  frozen add_well_defined (q_1, q_2: Q; t: T)
    require
      q_1 ~ q_2
    do
      addq (q_1, t)
      addq (q_2, t)
    ensure
      q_1 ~ q_2
    end

  frozen dequeue_well_defined (q_1, q_2: Q)
    require
      q_1 ~ q_2
      q_1 ≠ q_2
      not isnewq (q_1)
      not isnewq (q_2)
    do
      deleteq (q_1)
      deleteq (q_2)
    ensure
      q_1 ~ q_2
    end

  frozen front_well_defined (q_1, q_2: Q)
    require
      q_1 ~ q_2
    do
    ensure
      frontq (q_1) ~ frontq (q_2)
    end

  frozen empty_well_defined (q_1, q_2: Q)
    require
      q_1 ~ q_2
    do
    ensure
      isnewq (q_1) ~ isnewq (q_2)
    end

  frozen size_well_defined (q_1, q_2: Q)
    require
      q_1 ~ q_2
    do
    ensure
      size (q_1) ~ size (q_2)
    end

end
```

B.19 Queue with Append

```
note
  description: "Reusable abstract data type specification of queue with the ‘‘append’’ operation."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y44w738n"

deferred class
  QUEUE_WITH_APPEND_ADT [Q, T]
  -- Queues ‘‘Q’’ with appending other queues contain elements of ‘‘T’’.
```

```
inherit

  QUEUE_ADT [Q, T]

feature
  -- Deferred definitions.

  appendq (queue, other: Q)
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_18 (queue_1, queue_2: Q)
    require
      queue_1 ~ queue_2
    local
      other: Q
    do
      other := newq
      appendq (queue_1, other)
    ensure
      queue_1 ~ queue_2
    end

  frozen a_19 (queue_1, queue_2, other_1, other_2: Q; element: T)
    require
      queue_1 ~ queue_2
      other_1 ~ other_2
    do
      addq (other_1, element)
      appendq (queue_1, other_1)
      appendq (queue_2, other_2)
      addq (queue_2, element)
    ensure
      queue_1 ~ queue_2
    end

feature
  -- Well-definedness axioms.

  frozen appendq_well_defined (queue_1, queue_2, other: Q)
    require
      queue_1 ~ queue_2
    do
      appendq (queue_1, other)
      appendq (queue_2, other)
    ensure
      queue_1 ~ queue_2
    end

end
```

B.20 Set

```
note
```

```
  description: "Reusable abstract data type specification of set."
  description: "Found in ‘‘The Algebraic Specification of Abstract Data Types’’ by Guttag and Horning:
        "
  EIS: "src= https://link.springer.com/article/10.1007/BF00260922"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y2thcfbr"

deferred class
  SET_ADT [S, E]
  -- Sets ‘‘S’’ contain elements of ‘‘E’’.

inherit

  EQUALITY_ADT [S]

feature
  -- Deferred definitions.

  empty_set: S
    deferred
    end

  insert (set: S; element: E)
    deferred
    end

  delete (set: S; element: E)
    deferred
    end

  member_of (set: S; element: E): BOOLEAN
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (element: E)
    local
      set: S
    do
      set := empty_set
      check
        not member_of (set, element)
      end
    end

  frozen a_2_1 (set: S; element: E)
    do
      insert (set, element)
    ensure
      member_of (set, element)
    end

  frozen a_2_2 (set: S; element_1, element_2: E; old_member_of: BOOLEAN)
    require
      element_1 /~ element_2
      member_of (set, element_2) ~ old_member_of
    do
      insert (set, element_1)
    ensure
      member_of (set, element_2) ~ old_member_of
    end

  frozen a_3 (element: E)
    local
      set_1, set_2: S
    do
      set_1 := empty_set
      set_2 := empty_set
      delete (set_1, element)
      check
        set_1 ~ set_2
      end
    end

  frozen a_4_1 (set_1, set_2: S; element: E)
    require
      set_1 ~ set_2
    do
      insert (set_1, element)
      delete (set_1, element)
      delete (set_2, element)
    ensure
      set_1 ~ set_2
    end

  frozen a_4_2 (set_1, set_2: S; element_1, element_2: E)
    require
      set_1 ~ set_2
      element_1 /~ element_2
    do
      insert (set_1, element_1)
      delete (set_1, element_2)
      delete (set_2, element_2)
      insert (set_2, element_1)
    ensure
      set_1 ~ set_2
    end

feature
  -- Well-definedness axioms.

  frozen empty_set_well_defined
    local
      set_1, set_2: S
    do
      set_1 := empty_set
      set_2 := empty_set
      check
        assert: set_1 ≠ set_2
      end
      check
        assert: set_1 ~ set_2
      end
    end

  frozen insert_well_defined (set_1, set_2: S; element: E)
    require
      set_1 ~ set_2
    do
      insert (set_1, element)
      insert (set_2, element)
    ensure
      set_1 ~ set_2
    end

  frozen delete_well_defined (set_1, set_2: S; element: E)
    require
      set_1 ~ set_2
    do
      delete (set_1, element)
      delete (set_2, element)
    ensure
      set_1 ~ set_2
    end

  frozen member_of_well_defined (set_1, set_2: S; element: E)
    require
      set_1 ~ set_2
    do
    ensure
      member_of (set_1, element) ~ member_of (set_2, element)
    end

end
```

B.21 Set with IsEmptySet

```
note
  description: "Reusable abstract data type specification of set with ‘‘is_empty_set’’ operation."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y5lh2hro"

deferred class
  SET_WITH_ISEMPTYSET_ADT [S, I]
  -- Sets ‘‘S’’ contain elements of ‘‘I’’.

inherit

  EQUALITY_ADT [S]

feature
  -- Deferred definitions.

  empty_set: S
    deferred
    end

  is_empty_set (set: S): BOOLEAN
```

```
    deferred
    end

  insert (set: S; item: I)
    deferred
    end

  del_set (set: S; item: I)
    deferred
    end

  has (set: S; item: I): BOOLEAN
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1
    local
      set: S
    do
      set := empty_set
      check
        is_empty_set (set)
      end
    end

  frozen a_2 (set: S; item: I)
    do
      insert (set, item)
    ensure
      not is_empty_set (set)
    end

  frozen a_3 (item: I)
    local
      set: S
    do
      set := empty_set
      check
        not has (set, item)
      end
    end

  frozen a_4 (set: S; item: I)
    do
      insert (set, item)
    ensure
      has (set, item)
    end

  frozen a_5 (set: S; item_1, item_2: I; old_has: BOOLEAN)
    require
      item_1 /~ item_2
      has (set, item_2) ~ old_has
    do
```

```
      insert (set, item_1)
    ensure
      has (set, item_2) ~ old_has
    end

  frozen a_6 (set: S; item: I)
    local
      e_set: S
    do
      e_set := empty_set
      del_set (e_set, item)
      check
        e_set /~ set
      end
    end

  frozen a_7 (set_1, set_2: S; item: I)
    require
      set_1 ~ set_2
    do
      insert (set_1, item)
      del_set (set_1, item)
      del_set (set_2, item)
    ensure
      set_1 ~ set_2
    end

  frozen a_8 (set_1, set_2: S; item_1, item_2: I)
    require
      item_1 /~ item_2
      set_1 ~ set_2
    do
      insert (set_1, item_1)
      del_set (set_1, item_2)
      del_set (set_2, item_2)
      insert (set_2, item_1)
    ensure
      set_1 ~ set_2
    end

feature
  -- Well-definedness axioms.

  frozen empty_set_well_defined
    local
      set_1, set_2: S
    do
      set_1 := empty_set
      set_2 := empty_set
      check
        set_1 ≠ set_2
      end
      check
        set_1 ~ set_2
      end
    end
  frozen is_empty_set_well_defined (set_1, set_2: S)
    require
      set_1 ~ set_2
    do
    ensure
      is_empty_set (set_1) ~ is_empty_set (set_2)
    end

  frozen insert_well_defined (set_1, set_2: S; item: I)
    require
      set_1 ~ set_2
    do
      insert (set_1, item)
      insert (set_2, item)
    ensure
      set_1 ~ set_2
    end

  frozen del_set_well_defined (set_1, set_2: S; item: I)
    require
      set_1 ~ set_2
    do
      del_set (set_1, item)
      del_set (set_2, item)
    ensure
      set_1 ~ set_2
    end

  frozen has_well_defined (set_1, set_2: S; item: I)
    require
      set_1 ~ set_2
    do
    ensure
      has (set_1, item) ~ has (set_2, item)
    end

end
```

B.22 Stack

```
note
  description: "Reusable abstract data type specification of stack."
  description: "Found in ‘‘The design of data type specifications’’ by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  description: "Found in ‘‘Abstract Data Types and the Development of Data Structures’’ by Guttag:"
  EIS: "src= http://tinyurl.com/y45o32hq"
  description: "Found in ‘‘Programming with Abstract Data Types’’ by Liskov and Zilles:"
  EIS: "src= http://tinyurl.com/y5dc5k9h"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y62gkzyz"

deferred class
  STACK_ADT [S, T]
  -- Stacks ‘‘S’’ contain elements of ‘‘T’’.

inherit
```

```
  EQUALITY_ADT [S]

feature
  -- Deferred definitions.

  new: S
    deferred
    end

  push (s: S; t: T)
    deferred
    end

  pop (s: S)
    deferred
    end

  top (s: S): T
    deferred
    end

  is_new (s: S): BOOLEAN
    deferred
    end

  size (s: S): INTEGER
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1 (t: T; other: S)
    local
      s: S
    do
      s := new
      pop (s)
      check
        s /~ other
      end
    end

  frozen a_2 (t: T)
    local
      s: S
    do
      s := new
      check
        top (s) /~ t
      end
    end

  frozen a_3_empty (s_1, s_2: S)
    require
      s_1 ~ s_2
    local
```

```
      empty_: BOOLEAN
    do
      empty_ := is_new (s_1)
    ensure
      s_1 ~ s_2
    end

  frozen a_3_size (s_1, s_2: S)
    require
      s_1 ~ s_2
    local
      size_: INTEGER
    do
      size_ := size (s_1)
    ensure
      s_1 ~ s_2
    end

  frozen a_3_top (s_1, s_2: S)
    require
      s_1 ~ s_2
    local
      top_: T
    do
      top_ := top (s_1)
    ensure
      s_1 ~ s_2
    end

  frozen a_4_if (s: S)
    require
      size (s) ~ 0
    do
    ensure
      is_new (s)
    end

  frozen a_4_only_if (s: S)
    require
      is_new (s)
    do
    ensure
      size (s) ~ 0
    end

  frozen a_5 (s_1, s_2: S; t: T)
    require
      s_1 ~ s_2
    do
      push (s_1, t)
      pop (s_1)
    ensure
      s_1 ~ s_2
    end

  frozen a_6 (s: S; t: T)
    do
```

```
      push (s, t)
    ensure
      top (s) ~ t
    end

  frozen a_7 (s: S; t: T; old_size: INTEGER)
    require
      size (s) ~ old_size
    do
      push (s, t)
    ensure
      size (s) ~ old_size + 1
    end

  frozen a_8 (s: S; t: T; old_size: INTEGER)
    require
      size (s) ~ old_size
      not is_new (s)
    do
      pop (s)
    ensure
      size (s) ~ old_size − 1
    end

  frozen a_9 (s: S; t: T)
    do
      push (s, t)
    ensure
      not is_new (s)
    end

  frozen a_10
    local
      s: S
    do
      s := new
      check
        is_new (s)
      end
    end

  frozen a_11
    local
      s: S
    do
      s := new
      check
        size (s) ~ 0
      end
    end

feature
  -- Well-definedness axioms.

  frozen new_well_defined
    local
      s_1, s_2: S
    do
      s_1 := new
      s_2 := new
      check
        s_1 ~ s_2
      end
    end

  frozen push_well_defined (s_1, s_2: S; t: T)
    require
      s_1 ~ s_2
    do
      push (s_1, t)
      push (s_2, t)
    ensure
      s_1 ~ s_2
    end

  frozen pop_well_defined (s_1, s_2: S)
    require
      s_1 ~ s_2
      s_1 ≠ s_2
      not is_new (s_1)
      not is_new (s_2)
    do
      pop (s_1)
      pop (s_2)
    ensure
      s_1 ~ s_2
    end

  frozen top_well_defined (s_1, s_2: S)
    require
      s_1 ~ s_2
    do
    ensure
      top (s_1) ~ top (s_2)
    end

  frozen empty_well_defined (s_1, s_2: S)
    require
      s_1 ~ s_2
    do
    ensure
      is_new (s_1) ~ is_new (s_2)
    end

  frozen size_well_defined (s_1, s_2: S)
    require
      s_1 ~ s_2
    do
    ensure
      size (s_1) ~ size (s_2)
    end

end
```

B.23 Stack with Replace

```
note
   description: "Reusable abstract data type specification of stack with ‘‘replace’’ operation."
   description: "Found in ‘‘Abstract Data Types and Software Validation ’’ by Guttag, Horowitz and
          Musser:"
   EIS: "src= https://pdfs.semanticscholar.org/372d/4f331d0a6cd5fb4ee0c04d4a0753b8eb659f.pdf"
   description: "Found in ‘‘Abstract Data Types and the Development of Data Structures’’ by Guttag:"
   EIS: "src= http://tinyurl.com/y45o32hq"
   EIS: "name= Location on GitHub", "src= https://tinyurl.com/y4ql2lgn"

deferred class
   STACK_WITH_REPLACE_ADT [S, E]
   -- Stacks ‘‘S’’ with replacing contain elements of ‘‘E’’.

inherit

   STACK_ADT [S, E]

feature
   -- Deferred definitions.

   replace (stk: S; elm: E)
      deferred
      end

feature
   -- Abstract data type axioms.

   frozen a_12
      local
         stk_1, stk_2: S
      do
         stk_1 := new
         stk_2 := new
         pop (stk_1)
         check
            stk_1 ~ stk_2
         end
      end

   frozen a_13 (elm: E)
      local
         stk: S
      do
         stk := new
         check
            top (stk) /~ elm
         end
      end

   frozen a_14 (stk_1, stk_2: S; elm: E)
      require
         stk_1 ~ stk_2
      do
         replace (stk_1, elm)
         pop (stk_2)
```

```
    push (stk_2, elm)
  ensure
    stk_1 ~ stk_2
  end

feature
  -- Well-definedness axioms.

  frozen replace_well_defined (stk_1, stk_2: S; elm: E)
    require
      stk_1 ~ stk_2
    do
      replace (stk_1, elm)
      replace (stk_2, elm)
    ensure
      stk_1 ~ stk_2
    end

end
```

B.24 String

```
note
  description: "Reusable abstract data type specification of stting."
  description: "Found in ``The design of data type specifications'' by Guttag, Horowitz and Musser:"
  EIS: "src= http://tinyurl.com/yxmnv23w"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y3ezsvro"

deferred class
  STRING_ADT [S, C]
  -- Strings ``S'' contain characters ``C''.

inherit

  EQUALITY_ADT [S]

feature
  -- Deferred definitions.

  null: S
    deferred
    end

  is_null (string: S): BOOLEAN
    deferred
    end

  len (string: S): INTEGER
    deferred
    end

  add_char (string: S; character: C)
    deferred
    end

  concat (string_1, string_2: S)
```

```
    deferred
    end

  substr (string: S; start, finish: INTEGER): S
    deferred
    end

  index (string_1, string_2: S): INTEGER
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_1
    local
      string: S
    do
      string := null
      check
        is_null (string)
      end
    end

  frozen a_2 (string: S; character: C)
    do
      add_char (string, character)
      check
        not is_null (string)
      end
    end

  frozen a_3
    local
      string: S
    do
      string := null
      check
        len (string) ~ 0
      end
    end

  frozen a_4 (string: S; character: C; old_len: INTEGER)
    require
      len (string) ~ old_len
    do
      add_char (string, character)
    ensure
      len (string) ~ old_len + 1
    end

  frozen a_5 (string_1, string_2: S)
    require
      string_1 ~ string_2
    local
      string: S
    do
```

```
      string := null
      concat (string_1, string)
    ensure
      string_1 ~ string_2
    end

  frozen a_6 (string_1, string_2, string_3, string_4: S; character: C)
    require
      string_1 ~ string_4
      string_2 ~ string_3
    do
      add_char (string_1, character)
      concat (string_2, string_1)
      concat (string_3, string_4)
      add_char (string_3, character)
    ensure
      string_2 ~ string_3
    end

  frozen a_7 (start, finish: INTEGER)
    local
      string_1, string_2: S
    do
      string_1 := null
      string_2 := null
      check
        substr (string_2, start, finish) ~ string_1
      end
    end

  frozen a_8 (string: S; start, finish: INTEGER; character: C)
    require
      finish ~ 0
    local
      null_string: S
    do
      null_string := null
      add_char (string, character)
      check
        substr (string, start, finish) ~ null_string
      end
    end

  frozen a_9 (string_1, string_2: S; start, finish: INTEGER; character: C)
    require
      finish /~ 0
      finish ~ len (string_1) − start + 2
      string_2 ~ substr (string_1, start, finish − 1)
    do
      add_char (string_1, character)
      add_char (string_2, character)
    ensure
      string_2 ~ substr (string_1, start, finish)
    end

  frozen a_10 (string_1, string_2: S; start, finish: INTEGER; character: C)
    require
    finish /~ 0
    finish /~ len (string_1) − start + 2
    string_1 ~ string_2
  do
    add_char (string_1, character)
  ensure
    substr (string_1, start, finish) ~ substr (string_2, start, finish)
  end

frozen a_11 (string: S)
  local
    null_string: S
  do
    null_string := null
    check
      index (string, null_string) ~ len (string) + 1
    end
  end

frozen a_12 (string: S; character: C)
  local
    null_string: S
  do
    null_string := null
    add_char (string, character)
    check
      index (null_string, string) ~ 0
    end
  end

frozen a_13 (string_1, string_2, string_3: S; character_1, character_2: C)
  require
    string_1 ~ string_3
  do
    add_char (string_1, character_1)
    add_char (string_2, character_2)
    check
      assume: index (string_3, string_2) /~ 0
    end
  ensure
    index (string_1, string_2) ~ index (string_3, string_2)
  end

frozen a_14 (string_1, string_2, string_3, string_4: S; character_1, character_2: C)
  require
    string_1 ~ string_3
    string_2 ~ string_4
    character_1 ~ character_2
    index (string_1, string_2) ~ len (string_1) − len (string_2) + 1
  do
    add_char (string_1, character_1)
    add_char (string_2, character_2)
    check
      assume: index (string_3, string_2) ~ 0
    end
  ensure
    index (string_1, string_2) ~ index (string_3, string_4)
    end

  frozen a_15 (string_1, string_2, string_3: S; character_1, character_2: C)
    require
      string_1 ~ string_3
      character_1 /~ character_2
    do
      add_char (string_1, character_1)
      add_char (string_2, character_2)
      check
        assume: index (string_3, string_2) ~ 0
      end
    ensure
      index (string_1, string_2) ~ 0
    end

  frozen a_16 (string_1, string_2, string_3: S; character_1, character_2: C)
    require
      string_1 ~ string_3
      index (string_1, string_2) /~ len (string_1) − len (string_2) + 1
    do
      add_char (string_1, character_1)
      add_char (string_2, character_2)
      check
        assume: index (string_3, string_2) ~ 0
      end
    ensure
      index (string_1, string_2) ~ 0
    end

feature
  -- Well-definedness axioms.

  frozen null_well_defined
    local
      string_1, string_2: S
    do
      string_1 := null
      string_2 := null
      check
        string_1 ≠ string_2
      end
      check
        string_1 ~ string_2
      end
    end

  frozen is_null_well_defined (string_1, string_2: S)
    require
      string_1 ~ string_2
    do
    ensure
      is_null (string_1) ~ is_null (string_2)
    end

  frozen len_well_defined (string_1, string_2: S)
    require
```

```
      string_1 ~ string_2
    do
    ensure
      len (string_1) ~ len (string_2)
    end

  frozen add_char_well_defined (string_1, string_2: S; character: C)
    require
      string_1 ~ string_2
    do
      add_char (string_1, character)
      add_char (string_2, character)
    ensure
      string_1 ~ string_2
    end

  frozen concat_well_defined (string_1, string_2, string: S)
    require
      string_1 ~ string_2
    do
      concat (string_1, string)
      concat (string_2, string)
    ensure
      string_1 ~ string_2
    end

  frozen substr_well_defined (string_1, string_2: S; start, finish: INTEGER)
    require
      string_1 ~ string_2
    do
    ensure
      substr (string_1, start, finish) ~ substr (string_2, start, finish)
    end

  frozen index_well_defined (string_1, string_2, string: S)
    require
      string_1 ~ string_2
    do
    ensure
      index (string_1, string) ~ index (string_2, string)
    end

end
```

B.25 Symbol Table

```
note
  description: "Reusable abstract data type specification of symbol table."
  description: "Found in ‘‘Abstract Data Types and Software Validation’’ by Guttag, Horowitz and
       Musser:"
  EIS: "src= https://pdfs.semanticscholar.org/372d/4f331d0a6cd5fb4ee0c04d4a0753b8eb659f.pdf"
  description: "Found in ‘‘Abstract Data Types and the Development of Data Structures’’ by Guttag:"
  EIS: "src= http://tinyurl.com/y45o32hq"
  EIS: "name= Location on GitHub", "src= https://tinyurl.com/y3sja4uj"

deferred class
```

```
  SYMBOL_TABLE_ADT [S, I, A]
  -- Symbol tables ‘‘S’’ contain elements of ‘‘A’’
  -- indexed by elements of ‘‘I’’.

inherit

  EQUALITY_ADT [S]

feature
  -- Deferred definitions.

  init: S
    deferred
    end

  enter_block (s_t: S)
    deferred
    end

  leave_block (s_t: S)
    deferred
    end

  is_in_block (s_t: S; id: I): BOOLEAN
    deferred
    end

  add (s_t: S; id: I; attr: A)
    deferred
    end

  retrieve (s_t: S; id: I): A
    deferred
    end

feature
  -- Abstract data type axioms.

  frozen a_2 (s_t_1, s_t_2: S)
    require
      s_t_1 ~ s_t_2
    do
      enter_block (s_t_1)
      leave_block (s_t_1)
    ensure
      s_t_1 ~ s_t_2
    end

  frozen a_3 (s_t_1, s_t_2: S; id: I; attr: A)
    require
      s_t_1 ~ s_t_2
      s_t_1 ≠ s_t_2
    do
      add (s_t_1, id, attr)
      leave_block (s_t_1)
      leave_block (s_t_2)
    ensure
    s_t_1 ~ s_t_2
  end

frozen a_4 (id: I)
  local
    s_t: S
  do
    s_t := init
    check
      not is_in_block (s_t, id)
    end
  end

frozen a_5 (symtab: S; id: I)
  do
    enter_block (symtab)
  ensure
    not is_in_block (symtab, id)
  end

frozen a_6_1 (symtab: S; id: I; attrs: A)
  do
    add (symtab, id, attrs)
  ensure
    is_in_block (symtab, id)
  end

frozen a_6_2 (symtab: S; id_1, id_2: I; attrs: A; old_is_in_block: BOOLEAN)
  require
    id_1 /~ id_2
    is_in_block (symtab, id_1) ~ old_is_in_block
  do
    add (symtab, id_1, attrs)
  ensure
    is_in_block (symtab, id_2) ~ old_is_in_block
  end

frozen a_8 (symtab: S; id: I; attrs: A)
  require
    retrieve (symtab, id) ~ attrs
  do
    enter_block (symtab)
  ensure
    retrieve (symtab, id) ~ attrs
  end

frozen a_9_1 (symtab: S; id: I; attrs: A)
  do
    add (symtab, id, attrs)
  ensure
    retrieve (symtab, id) ~ attrs
  end

frozen a_9_2 (symtab: S; id_1, id_2: I; attrs, old_retrieve: A)
  require
    id_1 /~ id_2
    retrieve (symtab, id_2) ~ old_retrieve
    do
      add (symtab, id_1, attrs)
    ensure
      retrieve (symtab, id_2) ~ old_retrieve
    end

feature
  -- Well-definedness axioms.

  frozen init_well_defined
    local
      symtab_1, symtab_2: S
    do
      symtab_1 := init
      symtab_2 := init
      check
        symtab_1 ~ symtab_2
      end
    end

  frozen enter_block_well_defined (symtab_1, symtab_2: S)
    require
      symtab_1 ~ symtab_2
    do
      enter_block (symtab_1)
      enter_block (symtab_2)
    ensure
      symtab_1 ~ symtab_2
    end

  frozen leave_block_well_defined (symtab_1, symtab_2: S)
    require
      symtab_1 ~ symtab_2
    do
      leave_block (symtab_1)
      leave_block (symtab_2)
    ensure
      symtab_1 ~ symtab_2
    end

  frozen is_in_block_well_defined (symtab_1, symtab_2: S; id: I)
    require
      symtab_1 ~ symtab_2
    do
    ensure
      is_in_block (symtab_1, id) ~ is_in_block (symtab_2, id)
    end

  frozen add_well_defined (symtab_1, symtab_2: S; id: I; attr: A)
    require
      symtab_1 ~ symtab_2
    do
      add (symtab_1, id, attr)
      add (symtab_2, id, attr)
    ensure
      symtab_1 ~ symtab_2
    end
  frozen retrieve_well_defined (symtab_1, symtab_2: S; id: I)
    require
      symtab_1 ~ symtab_2
    do
    ensure
      retrieve (symtab_1, id) ~ retrieve (symtab_2, id)
    end

end
```

Bibliography

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. ISBN: 978-0-521-89556-9.

[AGR17] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. "Rigorous development process of a safety-critical system: from ASM models to Java code". In: *STTT* 19.2 (2017), pp. 247–269.

[AH94] Rajeev Alur and Thomas A. Henzinger. "A Really Temporal Logic". In: *J. ACM* 41.1 (1994), pp. 181–204.

[Alr+13] Dalal Alrajeh et al. "Elaborating Requirements Using Model Checking and Inductive Learning". In: *IEEE Trans. Software Eng.* 39.3 (2013), pp. 361–383.

[AM16] Yamine Aït Ameur and Dominique Méry. "Making explicit domain knowledge in formal system development". In: *Sci. Comput. Program.* 121 (2016), pp. 100–127.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. "Specification Language". In: *On the Construction of Programs*. 1980, pp. 343–410.

[Ban17] Richard Banach. "The landing gear system in multi-machine Hybrid Event-B". In: *STTT* 19.2 (2017), pp. 205–228.

[Bar+05] Michael Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. ISBN: 3-540-36749-7.

[Bar10] Mike Barnett. "Code Contracts for .NET: Runtime Verification and So Much More". In: *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. Ed. by Howard Barringer et al. Vol. 6418. Lecture Notes in Computer Science. Springer, 2010, pp. 16–17. ISBN: 978-3-642-16611-2.

[BDZF14] Bernard Berthomieu, Silvano Dal Zilio, and Łukasz Fronc. "Model-checking Real-Time Properties of an Aircraft Landing Gear System Using Fiacre". In: ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. Lecture Notes in Computer Science. Springer, 2014, pp. 110–125. ISBN: 978-3-662-43651-6.

[BH00] Martin Berger and Kohei Honda. "The Two-Phase Commitment Protocol in an Extended pi-Calculus". In: *Electr. Notes Theor. Comput. Sci.* 39.1 (2000), pp. 21–46.

[Bla+09] Sue Black et al. "Formal Versus Agile: Survival of the Fittest". In: *IEEE Computer* 42.9 (2009), pp. 37–45.

[Bli89] Wayne D. Blizard. "Multiset Theory". In: *Notre Dame Journal of Formal Logic* 30.1 (1989), pp. 36–66.

[BPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. "The Temporal Logic of Branching Time". In: *Acta Inf.* 20 (1983), pp. 207–226.

[BW14] Frédéric Boniol and Virginie Wiels. "The Landing Gear System Case Study". In: ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. Lecture Notes in Computer Science. Springer, 2014, pp. 1–18. ISBN: 978-3-662-43651-6.

[Ciu+08] Ilinca Ciupa et al. "Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports". In: *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*. IEEE Computer Society, 2008, pp. 157–166. ISBN: 978-0-7695-3405-3.

[Ciu+11] Ilinca Ciupa et al. "On the number and nature of faults found by random testing". In: *Softw. Test., Verif. Reliab.* 21.1 (2011), pp. 3–28.

[CK04] David R. Cok and Joseph Kiniry. "ESC/Java2: Uniting ESC/Java and JML". In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*. Ed. by Gilles Barthe et al. Vol. 3362. Lecture Notes in Computer Science. Springer, 2004, pp. 108–128. ISBN: 3-540-24287-2.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in Property Specifications for Finite-State Verification". In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.* Ed. by Barry W. Boehm, David Garlan, and Jeff Kramer. ACM, 1999, pp. 411–420. ISBN: 1-58113-074-0.

[Dal+18] Fabiano Dalpiaz et al. "Natural Language Processing for Requirements Engineering: The Best Is Yet to Come". In: *IEEE Software* 35.5 (2018), pp. 115–119.

[DNR04] Jerry Drobka, David Noftz, and Rekha Raghu. "Piloting XP on Four Mission-Critical Projects". In: *IEEE Software* 21.6 (2004), pp. 70–75.

[DT14] Philippe Dhaussy and Ciprian Teodorov. "Context-Aware Verification of a Landing Gear System". In: ed. by Yamine Aït Ameur and Klaus-Dieter Schewe. Vol. 8477. Lecture Notes in Computer Science. Springer, 2014, pp. 52–65. ISBN: 978-3-662-43651-6.

[DTS10] Jonathan Paul De Halleux, Nikolai Tillmann, and Wolfram Schulte. *Parameterized test driven development*. 2010.

[Ebe+16] Christof Ebert et al. "DevOps". In: *IEEE Software* 33.3 (2016), pp. 94–100.

[Eif] *Eiffel Community*. https://www.eiffel.org/.

[Fah+09a] Dirk Fahland et al. "Declarative versus Imperative Process Modeling Languages: The Issue of Maintainability". In: *Business Process Management Workshops, BPM 2009 International Workshops, Ulm, Germany, September 7, 2009. Revised Papers*. Ed. by Stefanie Rinderle-Ma, Shazia Wasim Sadiq, and Frank Leymann. Vol. 43. Lecture Notes in Business Information Processing. Springer, 2009, pp. 477–488. ISBN: 978-3-642-12185-2.

[Fah+09b] Dirk Fahland et al. "Declarative versus Imperative Process Modeling Languages: The Issue of Understandability". In: *Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings*. Ed. by Terry A. Halpin et al. Vol. 29. Lecture Notes in Business Information Processing. Springer, 2009, pp. 353–366. ISBN: 978-3-642-01861-9.

[FBM14] Luca Ferrucci, Marcello M Bersani, and Manuel Mazzara. "An LTL semantics of business workflows with recovery". In: *2014 9th International Conference on Software Paradigm Trends (ICSOFT-PT)*. IEEE. 2014, pp. 29–40.

[Fra+03] Steven Fraser et al. "Test Driven Development (TDD)". In: *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003 Proceedings*. Ed. by Michele Marchesi and Giancarlo Succi. Vol. 2675. Lecture Notes in Computer Science. Springer, 2003, pp. 459–462. ISBN: 3-540-40215-2.

[Gal18] Florian Galinier. *Specification of the London Ambulance System in AutoReq*. https://gitlab.com/fgalinier/LAS. 2018.

[GH78] John V. Guttag and James J. Horning. "The Algebraic Specification of Abstract Data Types". In: *Acta Inf.* 10 (1978), pp. 27–52.

[GH94] Yuri Gurevich and James K. Huggins. "Evolving Algebras and Partial Evaluation". In: *Technology and Foundations - Information Processing '94, Volume 1, Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August - 2 September, 1994*. Ed. by Björn Pehrson and Imre Simon. Vol. A-51. IFIP Transactions. North-Holland, 1994, pp. 587–592. ISBN: 0-444-81989-4.

[GHM76] John V Guttag, Ellis Horowitz, and David R Musser. “The Design of Data Type Specifications”. In: *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*. Ed. by Raymond T Yeh and C V Ramamoorthy. {IEEE} Computer Society, 1976, pp. 414–420.

[GHM78] John V Guttag, Ellis Horowitz, and David R Musser. “Abstract Data Types and Software Validation”. In: *Commun. {ACM}* 21.12 (1978), pp. 1048–1064.

[Gli] Martin Glinz. “Should Requirements Be Objects?” In: *Tutorial Position Paper, 14th Annual International Symposium on Systems Engineering*.

[Gur00] Yuri Gurevich. “Sequential abstract-state machines capture sequential algorithms”. In: *ACM Trans. Comput. Log.* 1.1 (2000), pp. 77–111.

[Gut76] John V. Guttag. “Abstract Data Types and the Development of Data Structures”. In: *Proceedings of the SIGPLAN ’76 Conference on Data: Abstraction, Definition and Structure, Salt Lake City, Utah, USA, March 22-24, 1976*. ACM, 1976, p. 72.

[HF01] Ivy F Hooks and Kristin A Farry. *Customer-centered products: creating successful products through smart requirements management*. Amacom Books, 2001.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[Hoa72] C. A. R. Hoare. “Proof of Correctness of Data Representations”. In: *Acta Inf.* 1 (1972), pp. 271–281.

[Ili+12] Alexei Iliasov et al. “Augmenting Event-B modelling with real-time verification”. In: *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*. Ed. by Stefania Gnesi et al. IEEE, 2012, pp. 51–57. ISBN: 978-1-4673-1906-5.

[IPP18] Mohsin Irshad, Kai Petersen, and Simon M. Poulding. “A systematic literature review of software requirements reuse approaches”. In: *Information & Software Technology* 93 (2018), pp. 223–245.

[ISO11] ISO/IEC/IEEE. “ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes –Requirements engineering”. In: *ISO/IEC/IEEE 29148:2011(E)* (2011).

[Jac06] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. ISBN: 978-0-262-10114-1.

[Jac14] Michael Jackson. “Topsy-turvy requirements”. In: *Requir. Eng.* 19.1 (2014), pp. 107–111.

[Jac17] Daniel Jackson. *Alloy Applications*. 2017. URL: `http://alloy.mit.edu/alloy/citations/case-studies.html`.

[Jac95] Michael Jackson. “The World and the Machine”. In: *17th International Conference on Software Engineering, Seattle, Washington, USA, April 23-30, 1995, Proceedings.* Ed. by Dewayne E. Perry, Ross Jeffery, and David Notkin. ACM, 1995, pp. 283–292. ISBN: 0-89791-708-1.

[Jon03] Cliff B. Jones. “The Early Search for Tractable Ways of Reasoning about Programs”. In: *IEEE Annals of the History of Computing* 25.2 (2003), pp. 26–49.

[Jon17] Cliff B. Jones. “Turing’s 1949 Paper in Context”. In: *Unveiling Dynamics and Complexity - 13th Conference on Computability in Europe, CiE 2017, Turku, Finland, June 12-16, 2017, Proceedings*. Ed. by Jarkko Kari, Florin Manea, and Ion Petre. Vol. 10307. Lecture Notes in Computer Science. Springer, 2017, pp. 32–41. ISBN: 978-3-319-58740-0.

[Jor08] Paul C. Jorgensen. *Software testing - a craftsman’s approach (3. ed.)* Taylor & Francis, 2008. ISBN: 978-0-8493-7475-3.

[JZ95] Michael Jackson and Pamela Zave. “Deriving Specifications from Requirements: An Example”. In: *17th International Conference on Software Engineering, Seattle, Washington, USA, April 23-30, 1995, Proceedings.* Ed. by Dewayne E. Perry, Ross Jeffery, and David Notkin. ACM, 1995, pp. 15–24. ISBN: 0-89791-708-1.

[KC02] Sascha Konrad and Betty H. C. Cheng. “Requirements Patterns for Embedded Systems”. In: *10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002), 9-13 September 2002, Essen, Germany*. IEEE Computer Society, 2002, pp. 127–136. ISBN: 0-7695-1465-0.

[KC05] Sascha Konrad and Betty H. C. Cheng. “Real-time specification patterns”. In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 372–381.

[Knu84] Donald E. Knuth. “Literate Programming”. In: *Comput. J.* 27.2 (1984), pp. 97–111.

[KW91] Uwe Kastens and William M Waite. “An Abstract Data Type for Name Analysis”. In: *Acta Inf.* 28.6 (1991), pp. 539–558.

[Lad+17] Lukas Ladenberger et al. “Validation of the ABZ landing gear system using ProB”. In: *STTT* 19.2 (2017), pp. 187–203.

[Lak10] Matt Lake. “Epic failures: 11 infamous software bugs”. In: *Computer-World, Sept* (2010).

[Lam01] Axel van Lamsweerde. “Goal-Oriented Requirements Engineering: A Guided Tour”. In: *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*. IEEE Computer Society, 2001, p. 249. ISBN: 0-7695-1125-2.

[Lam09] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009. ISBN: 978-0-470-01270-3.

[Lam+15] Wing Lam et al. *Parameterized Unit Testing in the Open Source Wild*. Tech. rep. 2015.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. ISBN: 978-3-642-17510-7.

[Lei13] K Rustan M Leino. *Verification Corner*. 2013. URL: `https://www.youtube.com/channel/UCP2eLEql4tROYmIYm5mA27A`.

[Let01] Emmanuel Letier. "Reasoning about agents in goal-oriented requirements engineering". PhD thesis. PhD thesis, Université catholique de Louvain, 2001.

[LZ74] Barbara Liskov and Stephen N Zilles. "Programming with Abstract Data Types". In: *{SIGPLAN} Notices* 9.4 (1974), pp. 50–59.

[Maz05] Manuel Mazzara. "Timing Issues in Web Services Composition". In: *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*. Ed. by Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro. Vol. 3670. Lecture Notes in Computer Science. Springer, 2005, pp. 287–302. ISBN: 3-540-28701-9.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0.

[MB10] M. Mazzara and A. Bhattacharyya. "On Modelling and Analysis of Dynamic Reconfiguration of Dependable Real-Time Systems". In: *2010 Third International Conference on Dependability*. 2010, pp. 173–181.

[MDB02] Ralph Miarka, John Derrick, and Eerke A. Boiten. "Handling Inconsistencies in Z Using Quasi-Classical Logic". In: *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*. Ed. by Didier Bert et al. Vol. 2272. Lecture Notes in Computer Science. Springer, 2002, pp. 204–225. ISBN: 3-540-43166-7.

[Mer+15] Florian Merz et al. "Bridging the gap between test cases and requirements by abstract testing". In: *ISSE* 11.4 (2015), pp. 233–242.

[Mey03] Bertrand Meyer. “A Framework for Proving Contract-Equipped Classes”. In: *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Proceedings*. Ed. by Egon Börger, Angelo Gargantini, and Elvinia Riccobene. Vol. 2589. Lecture Notes in Computer Science. Springer, 2003, pp. 108–125. ISBN: 3-540-00624-9.

[Mey+07] Bertrand Meyer et al. “Systematic evaluation of test failure results”. In: *Workshop on Reliability Analysis of System Failure Data (RAF2007)*. Technische Universität. 2007.

[Mey09] Bertrand Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer, 2009. ISBN: 978-3-540-92144-8.

[Mey+09] Bertrand Meyer et al. “Programs That Test Themselves”. In: *IEEE Computer* 42.9 (2009), pp. 46–55.

[Mey13] Bertrand Meyer. “Multirequirements”. In: *Modelling and Quality in Requirements Engineering: Essays dedicated to Martin Glinz on the occasion of his 60th birthday*. Verl.-Haus Monsenstein u. Vannerdat, 2013.

[Mey18] Bertrand Meyer. *The Formal Picnic approach to requirements*. `https://cacm.acm.org/blogs/blog-cacm/232677-the-formal-picnic-approach-to-requirements/fulltext`. 2018.

[Mey85] Bertrand Meyer. “On Formalism in Specifications”. In: *IEEE Software* 2.1 (1985), pp. 6–26.

[Mey88] Bertrand Meyer. “Eiffel: A language and environment for software engineering”. In: *Journal of Systems and Software* 8.3 (1988), pp. 199–246.

[Mey92] Bertrand Meyer. “Applying "Design by Contract"”. In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)* Wiley, 2006. ISBN: 978-0-470-09355-9.

[ML14] Formal Methods and Software Engineering Laboratory. *Landing Gear System ASM specification*. http://fmse.di.unimi.it/sw/landingGearSystem.zip. 2014.

[ML17] Amel Mammar and Régine Laleau. “Modeling a landing gear system in Event-B”. In: *STTT* 19.2 (2017), pp. 167–186.

[Mod+97] Francesmary Modugno et al. “Integrated Safety Analysis of Requirements Specifications”. In: *Requir. Eng.* 2.2 (1997), pp. 65–78.

[Naua] Alexandr Naumchev. *Jackson-Zave Zoo Turnstile Implementation*. `https://github.com/anaumche/Zoo-Turnstile-Multirequirements`.

[Naub] Alexandr Naumchev. *Seamless Requirements example*. `https://github.com/anaumchev/seamless_requirements`.

[Nau17] Alexandr Naumchev. *Landing Gear System ground model specification and requirements in Eiffel*. `https://github.com/anaumchev/lgs_ground_model`. 2017.

[Nau18] Alexandr Naumchev. “Detection of Inconsistent Contracts Through Modular Verification”. In: *Proceedings of 6th International Conference in Software Engineering for Defence Applications, SEDA 2018, Rome, Italy, June 7-8, 2018*. Ed. by Paolo Ciancarini et al. Vol. 925. Advances in Intelligent Systems and Computing. Springer, 2018, pp. 206–220. ISBN: 978-3-030-14686-3.

[Nau+19] Alexandr Naumchev et al. “AutoReq: Expressing and verifying requirements for control systems”. In: *Journal of Computer Languages* 51 (2019), pp. 131 –142. ISSN: 2590-1184.

[Nau19a] Alexandr Naumchev. “Object-oriented requirements: reusable, understandable, verifiable”. In: *TOOLS50+1*. 2019.

[Nau19b] Alexandr Naumchev. *Seamless Object-Oriented Requirement Templates*. `https://github.com/anaumchev/requirements_templates`. 2019.

[Neu95] Peter G. Neumann. *Computer-related risks*. Addison-Wesley, 1995. ISBN: 978-0-201-55805-0.

[NM16] Alexandr Naumchev and Bertrand Meyer. “Complete Contracts through Specification Drivers”. In: *10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*. IEEE Computer Society, 2016, pp. 160–167. ISBN: 978-1-5090-1764-5.

[NM17] Alexandr Naumchev and Bertrand Meyer. “Seamless requirements”. In: *Computer Languages, Systems & Structures* 49 (2017), pp. 119–132.

[NMR15] Alexandr Naumchev, Bertrand Meyer, and Víctor Rivera. “Unifying Requirements and Code: An Example”. In: *Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers*. Ed. by Manuel Mazzara and Andrei Voronkov. Vol. 9609. Lecture Notes in Computer Science. Springer, 2015, pp. 233–244. ISBN: 978-3-319-41578-9.

[Nor09] Darío Martín Nordio. “Proofs and proof transformations for object-oriented programs”. PhD thesis. Citeseer, 2009.

[Pic+11] Paul Pichler et al. “Imperative versus Declarative Process Modeling Languages: An Empirical Investigation”. In: *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*. Ed. by Florian Daniel, Kamel Barkaoui, and Schahram Dustdar. Vol. 99. Lecture Notes in Business Information Processing. Springer, 2011, pp. 383–394. ISBN: 978-3-642-28107-5.

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57.

[Pol+14] Nadia Polikarpova et al. “Flexible Invariants through Semantic Collaboration”. In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 514–530. ISBN: 978-3-319-06409-3.

[PQF17] Cristina Palomares, Carme Quer, and Xavier Franch. “Requirements reuse and requirement patterns: a state of the practice survey”. In: *Empirical Software Engineering* 22.6 (2017), pp. 2719–2762.

[PTF18] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. “A fully verified container library”. In: *Formal Asp. Comput.* 30.5 (2018), pp. 495–523.

[Rum+91] James E. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991. ISBN: 0-13-630054-5.

[SA07] Ahmed Samy Sidky and James D. Arthur. “Determining the Applicability of Agile Practices to Mission and Life-Critical Systems”. In: *31st Annual IEEE / NASA Software Engineering Workshop (SEW-31 2007), 6-8 March 2007, Loyola College, Columbia, MD, USA*. IEEE Computer Society, 2007, pp. 3–12. ISBN: 0-7695-2862-7.

[SA17] Wen Su and Jean-Raymond Abrial. “Aircraft landing gear system: approaches with Event-B to the modeling of an industrial system”. In: *STTT* 19.2 (2017), pp. 141–166.

[SBE08] David Saff, Marat Boshernitsan, and Michael D Ernst. “Theories in practice: Easy-to-write specifications that catch bugs”. In: (2008).

[SFO03] Guttorm Sindre, Donald G. Firesmith, and Andreas L. Opdahl. “A Reuse-Based Approach to Determining Security Requirements”. In: *The 9th International Workshop on Requirements Engineering: Foundation for Software Quality, REFSQ 2003*. Vol. 8. 2003, pp. 127–136. ISBN: 3-922602-87-8.

[SWM04] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. “Making Specifications Complete Through Models”. In: *Architecting Systems with Trustworthy Components, International Seminar, Dagstuhl Castle, Germany, December 12-17, 2004. Revised Selected Papers*. Ed. by Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski. Vol. 3938. Lecture Notes in Computer Science. Springer, 2004, pp. 48–70. ISBN: 3-540-35800-5.

[Teu17] Sabine Teufl. “Seamless Model-based Requirements Engineering: Models, Guidelines, Tools”. PhD thesis. Technical University Munich, Germany, 2017.

[TFR14] Dan Turk, Robert B. France, and Bernhard Rumpe. “Limitations of Agile Software Processes”. In: *CoRR* abs/1409.6600 (2014).

[TH08] Nikolai Tillmann and Jonathan de Halleux. “Pex-White Box Test Generation for .NET”. In: *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*. Ed. by Bernhard Beckert and Reiner Hähnle. Vol. 4966. Lecture Notes in Computer Science. Springer, 2008, pp. 134–153. ISBN: 978-3-540-79123-2.

[Tho87] Muffy Thomas. “Implementing Algebraically Specified Abstract Data Types in an Imperative Programming Language”. In: *TAPSOFT’87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23-27, 1987, Volume 2: Advanced Seminar on Foundations of Innovative Software Development {II} and Colloquium on Functional an*. Ed. by Hartmut Ehrig et al. Vol. 250. Lecture Notes in Computer Science. Springer, 1987, pp. 197–211. ISBN: 3-540-17611-X.

[TS05] Nikolai Tillmann and Wolfram Schulte. “Parameterized unit tests”. In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 2005, pp. 253–262.

[Tsc+11] Julian Tschannen et al. “Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques”. In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 382–398. ISBN: 978-3-642-24689-0.

[Tsc+15] Julian Tschannen et al. “AutoProof: Auto-Active Functional Verification of Object-Oriented Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 566–580. ISBN: 978-3-662-46680-3.

[Var] *Why loop variants are integers*. URL: `https://archive.eiffel.com/doc/faq/variant.html`.

[WHR14] Jon Whittle, John Edward Hutchinson, and Mark Rouncefield. “The State of Practice in Model-Driven Engineering”. In: *IEEE Software* 31.3 (2014), pp. 79–85.

[WN94] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, 1994.

[Yam62] Hisao Yamada. “Real-Time Computation and Recursive Functions Not Real-Time Computable”. In: *IRE Trans. Electronic Computers* 11.6 (1962), pp. 753–760.

[Zai+15] Asimina Zaimi et al. “An Empirical Study on the Reuse of Third-Party Libraries in Open-Source Software Development”. In: *Proceedings of the 7th Balkan Conference on Informatics Conference, BCI ’15, Craiova, Romania, September 2-4, 2015*. Ed. by Costin Badica et al. ACM, 2015, 4:1–4:8. ISBN: 978-1-4503-3335-1.

[Zav82] Pamela Zave. “An Operational Approach to Requirements Specification for Embedded Systems”. In: *IEEE Trans. Software Eng.* 8.3 (1982), pp. 250–269.