

Open Research Online

The Open University's repository of research publications and other research outputs

Progressing problems from requirements to specifications in problem frames

Thesis

How to cite:

Li, Zhi Z. (2008). Progressing problems from requirements to specifications in problem frames. PhD thesis The Open University.

For guidance on citations see [FAQs](#).

© 2007 Zhi Z. Li

Version: Version of Record

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data [policy](#) on reuse of materials please consult the policies page.

oro.open.ac.uk

Progressing Problems from Requirements to Specifications in Problem Frames

Zhi Z. Li, B.Sc., M.Sc.

A thesis submitted in partial fulfilment of the requirements for the degree
of Doctor of Philosophy in Computer Science

Department of Computing

Faculty of Mathematics and Computing

The Open University

September 2007

DATE of SUBMISSION: 28 SEPTEMBER 2007
NAME of CANDIDATE: ZHI Z. LI

ProQuest Number: 13890034

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 13890034

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

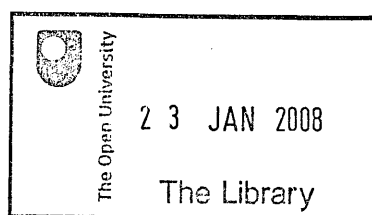
ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

ABSTRACT

One of the problems with current practice in software development is that often customer requirements are not well captured, understood and analysed, and there is no clear traceable path from customer requirements to software specifications. This often leads to a mismatch between what the customer needs and what the software developer understands the customer needs.

In addition to capturing, understanding and analysing requirements, requirements engineering (RE) aims to provide methods to allow software development practitioners to derive software specifications from requirements. Although work exists towards this aim, the systematic derivation of specifications from requirements is still an open problem.

This thesis provides practical techniques to implement the idea of *problem progression* as the basis for transforming requirements into specifications. The techniques allow us to progress a software problem towards identifying its solution by carefully investigating the problem context and re-expressing the requirement statement until a specification is reached. We develop two classes of progression techniques, one formal, based on Hoare's Communicating Sequential Processes (CSP), and one semi-formal, based on a notion of causality between events. The case studies in this thesis provide some validation for the techniques we have developed.



T005.12
C

AUTHOR'S DECLARATION

Some of the material in this thesis appears in the following papers.

- L. Rapanotti, J. G. Hall and Z. Li, Deriving specifications from requirements through problem reduction, *Journal of IEE Proceedings - Software*, Volume 153, issue 5, pages 183-198, October 2006.
- Z. Li, J. G. Hall and L. Rapanotti, From requirements to specifications: a formal approach, *Proceedings of the 2nd International Workshop on Advances and Applications of Problem Frames*, pages 65-70, ICSE'06, ACM Press, Shanghai, China, May 2006.
- L. Rapanotti, J. G. Hall and Z. Li, Problem Reduction: a systematic technique for deriving Specifications from Requirements, Technical Report No.2006/02, Centre for Research in Computing, The Open University, February 2006.
- Z. Li, J. G. Hall and L. Rapanotti, A Constructive Approach to Problem Frame Semantics, Technical Report No.2004/26, Centre for Research in Computing, The Open University, December 2004.
- Z. Li, J. G. Hall and L. Rapanotti, Reasoning about decomposing and recomposing Problem Frames developments: a case study, *Proceedings of the 1st International Workshop on Advances and Applications of Problem Frames*, pages 49-53, ICSE'04, IEEE CS Press, Edinburgh, UK, May 2004.

I declare that no part of this material has previously been submitted to a degree or any other qualification at this University or another institution.

I further declare that this thesis is my original work, except for clearly indicated sections where appropriate attributions and acknowledgements are given to work by other authors.

Zhi Li

ACKNOWLEDGEMENTS

Thanks are due to the following individuals and organisations without whose assistance this thesis would not have been possible.

Financial and facility support was provided by the Research School and the Faculty of Mathematics and Computing at the Open University.

I am grateful to an extremely dedicated team of supervisors: this thesis could not have been written without the supervision, guidance and support of Jon G. Hall and Lucia Rapanotti.

As my supervisor, Jon provided me with the inspiration and endless encouragement to help me pursue my dream of getting a PhD. He has been unfailingly supportive, patient and generous. His door is always open to me whenever I have any questions about formal methods and problem orientation.

As my supervisor, Lucia taught me many things about doing research in academia, and improved my writing greatly. She has been extremely supportive and generous to my research project, devoting a lot of time and attention to make sure that I produced a quality thesis.

Marian Petre and Trevor Collins have taught me how to make the transition from a research student to a qualified researcher. The PG-Forum (Postgraduate Forum) has become a great help to our research students.

Bashar Nuseibeh has been very generous towards all research students, making sure

we are treated the same way as members of staff.

Michael A. Jackson kindly spent some time talking to me about problem frames at the early start of my PhD. His encouragement and generosity is much appreciated.

Friendship and help from John Brier, my colleague from Yorkshire, has always been greatly appreciated. Friendship from Debra Haley and Charles Haley has been a joy. Encouragement from students and staff in the department is acknowledged.

Thanks should go to Jonathan Moffett who made me aware of the PhD project in the Open University. A special thanks should also go to Michael D. Harrison who made this project possible with his reference.

Last but not least, the support of my family has been with me beyond the PhD years: my mother Suwen Zhang, my father Kelin Li and my elder sister Ping Li have been supporting my interest and determination of pursuing a PhD in computer science since I came to the UK in 2001. My mother-in-law Xiangyang Cao, and my father-in-law Zhonghua Shen have encouraged me and helped look after my son during my stay in the UK, for which I am very grateful. I am extremely grateful to my wife Yan Shen and my son Zongze Li, for their long-lasting understanding, patience and endurance so that I can get the thesis completed.

CONTENTS

1. <i>Introduction</i>	1
1.1 Aim and Research Methodology	2
1.2 Thesis Contribution	4
1.3 Thesis Outline	4
2. <i>Literature Survey</i>	6
2.1 Why Requirements Engineering?	6
2.1.1 Software Crisis and Important Findings	6
2.1.2 The Role of Requirements Engineering in Software Development	8
2.2 Narrative Approaches	10
2.2.1 Use Cases	10
2.2.2 Scenarios	15
2.2.3 Deriving Specifications from Requirements Using Use Case and Scenario Approaches	17
2.3 Goal-Oriented Approaches	18
2.3.1 The KAOS Approach	18
2.3.2 The i* Approach	19
2.3.3 Deriving Specifications from Requirements Using Goal-Oriented Approaches	20

2.4	A Formal Approach to Relating Requirements and Specifications - The Four Variable Model	21
2.5	Problem-Based Approaches	23
2.5.1	Foundation	23
2.5.2	Problem Frames	25
2.5.3	Problem-Oriented Software Engineering	27
2.6	Transformational Approaches in Software Engineering	27
2.7	Summary	28
3.	<i>Problem Progression</i>	30
3.1	The Problem Frames Approach	30
3.1.1	The Engineering Root of PF	31
3.1.2	Representing Problems	32
3.1.3	Transforming Problems	42
3.2	Problem Progression and its Significance	45
3.3	Summary	48
4.	<i>A Formal Approach to Problem Progression</i>	49
4.1	An Example	50
4.2	Semantics of Problem Diagrams	52
4.3	Formalising a Problem Diagram Using CSP as a DRDL	53
4.3.1	The CSP language	53
4.3.2	Modelling a Domain as a CSP Process	59
4.3.3	The (Stable) Failures Model in CSP	59
4.3.4	Modelling a Requirement and \vdash_{DRDL} in the Predicative Failures Model	67

4.3.5	Modelling the Sharing of Phenomena as Parallel Composition . . .	68
4.3.6	Distinguishing “Control” and “Observe” in CSP Descriptions . .	69
4.3.7	Achieving a Complete Interpretation of Hall <i>et al.</i> ’s PF Seman- tics in CSP	70
4.4	Solving the Challenge Using Lai’s Quotient	71
4.4.1	Interpreting Problem Progression as Stepwise Applications of Lai’s Quotient	74
4.5	Case Study - Solving the POS Example Problem	75
4.5.1	Formalising the Domain and Requirement	75
4.5.2	Solving the Problem Using Lai’s Quotient	83
4.5.3	Using <i>SKIP</i> instead of <i>STOP</i>	88
4.5.4	Validating the Derived Specification Using FDR	89
4.6	Discussion on our Formal Approach to Problem Progression	92
4.6.1	Complexity	92
4.6.2	Weakening Problem Descriptions	93
4.7	Summary	94
5.	<i>A Semi-Formal Approach to Problem Progression</i>	96
5.1	Causality	97
5.1.1	Basic Notation	98
5.1.2	Types of Causality	100
5.2	Causality in Problem Description	102
5.2.1	Using Causality to Describe Domain Behaviour	102
5.2.2	Relationship between Control and Causality of Phenomena . . .	103
5.3	Progressing Problems Based on Graph Grammar	105

5.3.1	Graph Grammars	106
5.3.2	Interpreting Problem Diagrams as Directed Labelled Graphs . .	114
5.3.3	Interpreting Problem Progression as Rule-Based Graph Trans- formation	119
5.4	Causality-Based Rules for Problem Progression	121
5.4.1	The Reducing through Cause and Effect Rule Class	124
5.4.2	The Changing Viewpoint Rule Class	133
5.4.3	The Removing Domain Rule Class	139
5.5	Discussion on Heuristics for Applying the Transformation Rules	145
5.6	Summary	147
6.	Case Studies	149
6.1	The Point-of-Sale (POS) Problem	149
6.1.1	First Step of Progression	152
6.1.2	Second Step of Progression	153
6.1.3	Third Step of Progression	154
6.1.4	Fourth Step of Progression	155
6.1.5	Fifth Step of Progression	156
6.1.6	Sixth Step of Progression	157
6.1.7	Seventh Step of Progression	158
6.1.8	Eighth Step of Progression	159
6.1.9	Ninth Step of Progression	160
6.1.10	Tenth Step of Progression	161
6.1.11	Eleventh Step of Progression	162
6.1.12	Twelfth Step of Progression	163

6.1.13	Thirteenth Step of Progression	164
6.2	The Package Router Problem	167
6.2.1	First Step of Progression	173
6.2.2	Second Step of Progression	174
6.2.3	Third Step of Progression	175
6.2.4	Fourth Step of Progression	176
6.2.5	Fifth Step of Progression	177
6.2.6	Sixth Step of Progression	179
6.2.7	Seventh Step of Progression	180
6.2.8	Eighth Step of Progression	181
6.2.9	Ninth Step of Progression	182
6.2.10	Tenth Step of Progression	183
6.2.11	Eleventh Step of Progression	184
6.3	Discussions	187
6.4	Chapter Summary	188
7.	<i>Discussions, Conclusions and Future Work</i>	189
7.1	Aim of the Thesis and Contribution Evaluation	189
7.1.1	How Systematic Are They?	190
7.1.2	Scope of Their Application	190
7.1.3	Practicality of Their Application	191
7.2	Conclusion and Future Work	192
	<i>Appendix</i>	194
A.	<i>Details of Distinguishing “Control” and “Observe” in CSP Descriptions</i> . . .	195

B. Details of Problem Progression Rules 198

 B.1 The Reducing through Cause and Effect Rule Class 198

1. INTRODUCTION

One of the problems with current practice in software development is that often customer requirements are not well captured, understood and analysed, and there is no clear traceable path from customer requirements to software specifications. This often leads to a mismatch between what the customer needs and what the software developer understands the customer needs [27].

This problem has been known to the software engineering community for a long time. For example, in the *2nd International Conference on Software Engineering* in 1976, the review by Bell and Thayer [12] confirmed that “the rumoured ‘requirements problems’ are a reality”. Later in 1994, the “Chaos Report” [152] by the Standish Group indicated that this problem continued to exist in software development practice. Historically, the discipline of requirements engineering (RE) was born because of the realisation that there had not been enough focus on requirements [143].

In addition to capturing, understanding and analysing requirements, an important aim of requirements engineering is to provide methods to allow software development practitioners to derive software specifications from requirements. Although work exists towards this aim, such as the scenario approaches [3] and goal-oriented approaches [166, 159], the problem of systematically deriving specifications from requirements is still an open problem in RE. After reviewing the current state of the literature this thesis will address this open problem in a systematic way.

1.1 Aim and Research Methodology

We adopt the problem-oriented approach to requirements and specifications proposed by Jackson [82] and in particular his work on problem frames [83]. Jackson distinguishes between requirements and specifications, where a specification is a behavioural description of the computing machine in terms of its shared interface with its environment; and a requirement is a description of some desired behaviour in the environment that the computing machine must eventually bring about.

We take this approach for several reasons:

Firstly, it encompasses the basic idea that having a proper understanding of the problem (the requirement in its context) is a first essential step in providing an appropriate solution. There is evidence that many failed software projects did not get their requirements right in the first place so that mistakes were propagated through the entire development process, and became much more expensive to fix in later phases [152, 105].

Secondly, it underlines an important distinction between the problem space, where the requirements are, and the solution space, where the specifications are. By separating the description of requirements from that of specifications, we can formulate a clear argument about how the requirements can be adequately satisfied by the specifications.

Thirdly, it provides a notation (the problem diagram) to represent details of the problem space in relation to the solution space, hence the means to reason about requirements, contexts, specifications and their relationships.

The aim of this thesis is to provide practical techniques to implement the idea of problem progression sketched in [83] as the basis for transforming requirements into specifications. The techniques we will provide for problem progression will allow us to progress a problem towards identifying its solution by carefully investigating the

problem context and re-expressing the requirement statement until a specification is reached.

We develop two classes of progression techniques, one formal, based on Hoare's Communicating Sequential Processes (CSP) [68], and one semi-formal, based on a notion of causality between events [111]. We choose CSP because it has a rich set of operators we can exploit for describing and transforming problems, in particular, the parallel composition operator and Lai's quotient operator [101]. This fully-formal technique allows for the derivation of specifications from requirements by formal calculus. We develop rule-based techniques based on causality because they can be applied to a wider variety of problems where fully-formal descriptions can not be easily obtained.

We test our techniques on a range of case studies¹. We apply the formal technique to a simplified version of a point-of-sale (POS) system; we apply the semi-formal techniques to more complex case studies - a conventional point-of-sale (POS) problem and a package router control problem. We argue that although they are not real-world case studies, they are sufficiently complex and representative of real-world situations to test our hypothesis - that we have solved the problem of systematically deriving software specifications from requirements using our techniques. With these examples, we have demonstrated that our techniques can be practically applied in solving realistic software development problems that are described using causal phenomena.

Both empirical studies and well-chosen exemplars are very common ways of validating software engineering research [149]. Through these case studies we support the claim that we have developed adequate techniques for problem progression in the context of requirements engineering.

¹ In this thesis, case studies refer to examples with various complexity usually taken from the literature.

1.2 Thesis Contribution

The main contributions of the thesis are:

- A formal approach and associated techniques for the derivation of specifications from requirements based on CSP;
- A semi-formal approach and associated rule-based techniques for the practical derivation of specifications from requirements in a wide range of problems;
- An assessment of the proposed techniques on a number of examples and case studies.

1.3 Thesis Outline

This thesis is structured as follows:

Chapter 2 surveys related literature, focusing on how current RE approaches tackle the problem of deriving specifications from requirements. Their advantages and disadvantages are examined. A gap is highlighted in the literature which this thesis intends to fill.

Chapter 3 describes what problem progression is and its conceptual basis, which includes the problem frames approach (i.e., its engineering background and some basic elements).

Chapter 4 describes a formal approach to problem progression using CSP. In this chapter, problem progression is interpreted in a formal setting and constructive techniques are applied in a case study to derive specifications from requirements. Limitations of applying such a formal technique in problem progression are discussed, and the necessity of further less-formal techniques is argued.

Chapter 5 describes a semi-formal approach to problem progression based on the notion of causality. A working definition of causality and some derived notations and techniques are given. Progression rules are defined for the practical achievement of problem progression.

Chapter 6 applies the techniques defined in chapter 5 to two case studies. The first case study is a typical point-of-sale (POS) problem, and the second one is a package router problem.

Chapter 7 discusses how the aim of this thesis is fulfilled, concludes the thesis, and sets an agenda for future work.

2. LITERATURE SURVEY

This chapter reviews current requirements engineering approaches with a focus on their advantages and disadvantages. After examining how each of them allows for the derivation of specifications from requirements, we highlight a gap in the literature which our work intends to fill.

2.1 *Why Requirements Engineering?*

2.1.1 *Software Crisis and Important Findings*

The formation of *Software Engineering* (SE) was led by the so-called “software crisis” [2] in late 1960s. At that time, requirements analysis was perceived as a potentially high-leverage but neglected area in software development [55]. By the mid-1970s, the review by Bell and Thayer [12] had produced plenty of empirical data, confirming that “the rumoured ‘requirements problems’ are a reality”. The growing recognition of the critical nature of requirements in software engineering gradually established *Requirements Engineering* (RE) as an important sub-field of Software Engineering [55]. (It was not until 1993 that the 1st international conference dedicated to requirements engineering - 1st IEEE International Symposium on Requirements Engineering [143] - was held in San Diego, CA, U.S.A.)

The software crisis was also highlighted by the publication of Brooks’ famous book

The Mythical Man-Month: Essays on Software Engineering [19] and his seminal paper *No Silver Bullet: Essence and Accidents of Software Engineering* [20], which became chapter 16 of the 20th anniversary edition of the book [21]. Brooks attributed the software crisis to two distinct kinds of difficulties in software development (engineering) - essential difficulties and accidental difficulties. The paper suggested that there is no need for “a silver bullet” for solving major accidental difficulties because they have been solved by past breakthroughs in software engineering. Essential difficulties are much harder to solve because of the inherent properties of modern software systems - complexity, conformity, changeability, and invisibility, and they should be the targets for the silver bullet.

Although most of these properties seem inherent in software and hardware, in fact many of them are caused by the nature of their interaction with the outside world: for example, Brooks [21] argues that conformity is caused by the involvement of different people, and “cannot be simplified out by any redesign of the software alone”; this is more true as to changeability: “the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.”

In [20], Brooks puts “requirements refinement” as one of the promising ways to tackle such essential difficulties:

“The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more

difficult to rectify later.”

Since requirements refinement is a difficult task in SE, it deserves to be the focus of engineering efforts in modern software development. An interesting observation of this thesis on Brooks’ comments about requirements refinement is that the “detailed technical requirements” essentially refer to software specifications, and the process of “deciding precisely what to build” can be regarded as deriving specifications from requirements.

Although much progress has been made since the 1960s, requirement deficiencies in many software development projects are still a main contributing factor to project failures [43]. Sommerville and Sawyer [151] observe that a large number of project cost overruns and late deliveries still exist because of poor requirements engineering processes.

2.1.2 The Role of Requirements Engineering in Software Development

Before investigating the role that RE plays in software development, let us look at Zave’s definition of RE [168]:

“Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families.”

From the above definition, Nuseibeh and Easterbrook have argued that the role of RE is *representing* the “why” and “what” of a system, *analysing* its requirements, *validating*

that they are really what stakeholders want, *defining* what should be built, *verifying* that it has been built correctly, and *adapting* to the changing world by reusing partial specifications in RE [115].

Although there is now little dispute about the importance of requirements engineering in software development and a lot of different approaches and frameworks have been developed for RE, there is still little consensus on process support or even a common vocabulary of definitions [41, 122].

Recently, there have been some attempts to provide a common foundation and some processes for RE. For example, Zave and Jackson [169] have identified weaknesses (i.e., the “four dark corners”) in RE and they have proposed a conceptual foundation for RE: they argue all descriptions involved in RE should describe the environment, provide necessary control information, support requirement refinement, etc. They propose a minimum criteria for determining exactly what it means for RE to be considered successfully completed, based on a relationship among requirements, domain knowledge and specifications. Nuseibeh *et al.* [114, 59] have proposed the Twin-Peaks process model [114] as a way to embed RE in software development practice: the model is an adaptation of the spiral model [14] based on experiences in industrial development projects. It proposes to relate software requirements and architectures in an iterative fashion, in which the role of requirements engineering is to achieve a satisfactory structure in the problem space as early as possible to inform architectural design in the solution space. However, these proposals have yet to be widely accepted in the academic community and adopted in industrial practice.

This thesis contributes to the investigations of the above proposals, and it views the role of requirements engineering in software development in the following way: firstly it helps to start the process of moving from the problem space to the solution space

by eliciting requirements and domain knowledge, and structuring them in a suitable way to derive specifications that can influence and justify design decisions, and then drive successive iterations of the development process by fine-tuning such knowledge either informed by the problem space (e.g., mistakes, conflicts or changes in domain knowledge or requirements, etc) or the solution space (e.g., architectural styles or design choices, etc). In the following section, we will review current main approaches in requirements engineering and discuss how they support the derivation of specifications from requirements.

2.2 Narrative Approaches

There are two main types of “narrative” approaches to requirements engineering - *use cases* and *scenarios*, which often overlap with each other. We use the term narrative to indicate that these approaches describe the context and requirements in natural language. Narratives are used for eliciting and validating requirements with project stakeholders [108], and are popular in software development practice [162].

2.2.1 Use Cases

Use cases are a technique for capturing the intended requirements of a new system or software change. Each use case consists of one or more scenarios that narratively describe how the intended system should interact with the user or other systems to achieve a particular goal [164].

Use cases are thought to facilitate the elicitation and communication of requirements from the user’s point of view [139, 144]. Although use cases are not object-oriented in nature, historically, they have been closely linked to UML (Unified Modelling Language

[16]) and OOAD (Object-Oriented Analysis and Design [104]) to support a complete development process.

What Is the Definition of a Use Case?

There have been many different definitions of use case in the literature, each of which has a slightly different focus. Here are some of them:

“A use case is a narrative document that describes the sequence of events of an actor (an external agent) using a system to complete a process.” [89]

“They are stories or cases of using a system. Use cases are not exactly requirements or functional specifications, but they illustrate and imply requirements in the stories they tell.” [103]

“A use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.” [16]

In [30], Cockburn summaries 18 different definitions of use case given by different experts, teachers and consultants and gives the following definition:

“Scenario. A sequence of interactions happening under certain conditions, to achieve the primary actor’s goal, and having a particular result with respect to that goal. The interactions start from the triggering action and continue until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction.”

“Use Case. A collection of possible scenarios between the system under discussion and external actors, characterized by the goal the primary actor

has toward the system's declared responsibilities, showing how the primary actor's goal might be delivered or might fail."

Scope and Elements of a Use Case

According to the above definition, a use case consists of the following elements: firstly, the "*system under discussion*" mostly refers to the digital computer where the hardware and its intended software reside. It is typically treated as a "black box" perceived from the outside world to prevent premature assumptions about how the intended system is implemented; secondly, the "*actors*" are parties outside the system that interact with it. An actor can be a class of users or other systems (including other software systems). Actors can be classified into the primary actors and secondary actors. The primary actor is the stakeholder whose goal is the main theme of the use case and the secondary actor is an external actor who provides a service to the system under discussion; and thirdly, the "*goal*" is a single task or purpose that a use case must achieve.

The "system under discussion"

There is some ambiguity with the word "system" in the above use case definition (a detailed discussion on this can be found in [83]):

If, traditionally, the system strictly means the digital computer (including its hardware and software), then what all use cases are describing is the computer's interaction with actors outside. This view is heavily focused on the computer and its close neighbourhood, with assumptions that its relationship with the wider neighbourhood is trivial.

Figure 2.1 is a typical use case diagram illustrating this focus on the boundary between the system under discussion and the actors.

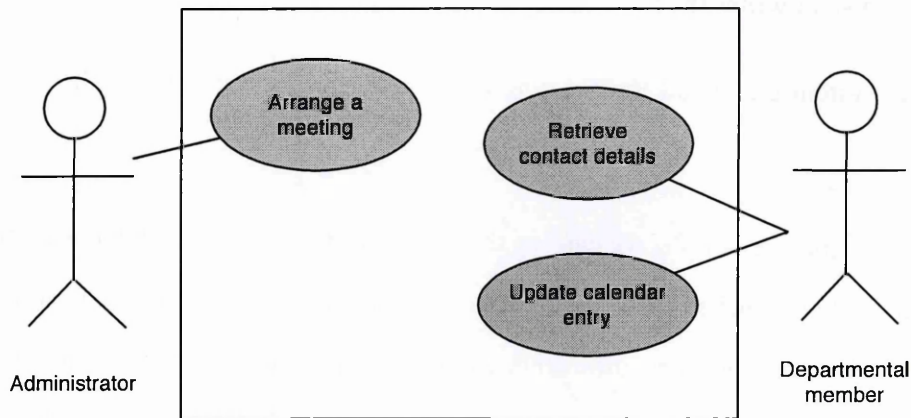


Fig. 2.1: Use case diagram for the shared calendar system taken from [124] unmodified

A brief narrative description of the use case - *arranging a meeting using the shared calendar system* - can be as follows [124]:

- The user chooses the option to arrange a meeting.
- The system prompts the user for the names of attendees.
- The user types in a list of names.
- The system checks that the list is valid.
- The system prompts the user for meeting constraints.
- The user types in meeting constraints.
- The system searches the calendars for a date that satisfies the constraints.
- The system displays a list of potential dates.
- The user chooses one of the dates.

- The system writes the meeting into the calendar.
- The system emails all the meeting participants informing them of the appointment.

From the above example, we can see that use cases (especially the textbook version by Cockburn [31]) tend to focus on details about how users interact with the computer system. However, from a requirements engineering perspective, their subject matter should be wider. For instance, Robertson and Robertson [132] suggest that “business use cases” are needed where the scope is much wider than the system-actor boundary. Instead of using the term “the system”, they use the term “the work” to cover a much wider context. In [133] they have named Cockburn’s use case “product use case” to distinguish from their “business use case”. Figure 2.2 shows the “business use case” in relation to the “product use case” in the wider context of “business event”.

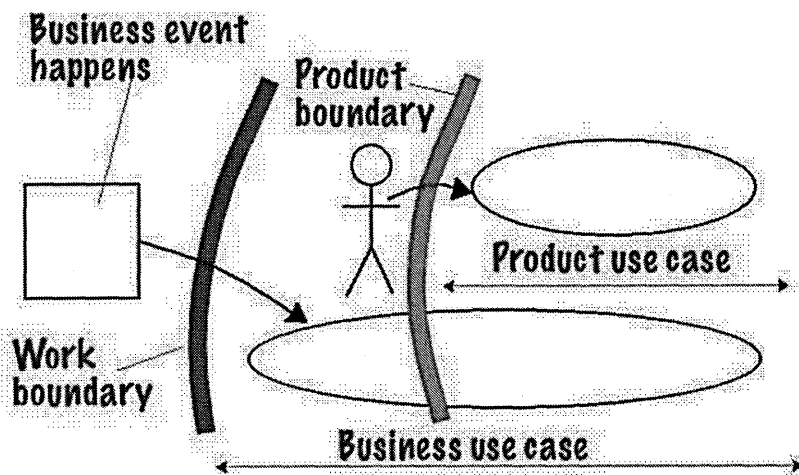


Fig. 2.2: Connections between the product use case, business use case and business events (taken from [133] unmodified)

Advantages of the Use Case Approach

In use cases, the focus is on the boundary between the digital computer and the actors, thus avoiding detailed design of the solution before the requirements are explored. The narrative nature of a use case often makes it accessible for requirements elicitation, documentation and validation from the actor's perspective.

Disadvantages of the Use Case Approach

Use cases have downsides as well: the focus of the textbook version of a use case (e.g., [31]) is limited to the boundary between the digital computer and the actor in its environment, in other words, not enough context is considered for requirements engineering. Like other natural languages, badly-written use cases suffer from ambiguity and inconsistency due to lack of sound guidelines. Use cases are not well suited to capturing non-functional requirements, hence, there is always an "other specification" section in addition to use cases (e.g., [104]). Regnell *et al.* [130] observe that we usually get "a loose collection of use cases which are separate, partial models, addressing narrow aspects of the system requirements" in this approach, which suggests use cases should be guided or complemented by more complete models.

2.2.2 Scenarios

Scenarios have been a focus in requirements engineering research and practice because they can offer narratives to bridge the communication gap among various stakeholders in a development project. In requirements engineering, they have been effective in eliciting, describing and validating requirements [5, 132, 3]. Scenarios are also used in other fields such as human-computer interaction (HCI) [25, 124] and strategic planning

[22], etc.

What Is a Scenario?

A scenario has been defined as an “informal narrative description” by Carroll [24]. Preece *et al.* [124] observe that in human-computer interaction (HCI), a scenario describes human activities or tasks in a story format which allows stakeholders to explore and express contexts, needs, and requirements. Within use cases, a scenario usually represents one path through the actor’s interaction with the machine.

Another definition of a scenario is given by Haumer [66]: a scenario presents a concrete story or instance of a specification, i.e., examples of using a system to accomplish some desired function.

Advantages of the Scenario Approach

Robertson and Robertson’s approach to requirements [132] shows how and why a scenario approach has some advantage over the textbook version of use case by looking at a wider context - responding to the real business event behind use cases in support of product innovation.

Scenarios provide an informal, narrative and concrete style of descriptions that focus on the dynamic aspects of the computer-environment interactions [160]. They help get the user involved in the RE process, increase the developer’s understanding of domain modelling, and facilitate communication between developers and customers [142].

Haumer [66] observes that scenarios help project stakeholders reach partial agreement and consistency because scenarios can ground discussions and negotiations on real examples. He also points out that scenarios are good for maintaining certain concrete levels of traceability in the whole development process, e.g., writing test cases [66].

Lamsweerde *et al.* [160] argue that scenarios may serve many purposes in the requirements engineering life-cycle, such as requirements elicitation [123] [8]; populating conceptual models [140] [138], business rules [136] or glossaries [162]; validating requirements together with prototyping [153], animation [44], or planning generation tools [7] [53]; reasoning about usability during system development [26]; generating acceptance test cases [75]; and structuring requirements through user-oriented decomposition for subsequent work assignment [162].

Disadvantages of the Scenario Approach

Scenarios share many of the disadvantages and limitations of use cases. For example, they are mainly described in a natural language, whose ambiguity may be an issue [23, 90]. Sutcliffe [154] observes that scenarios may encourage “confirmation bias”, that is, people tend to seek only positive examples that agree with their preconceptions [93]. He also points out that scenario approaches have sampling and coverage problems - scenarios can bias beliefs in frequencies of events and probabilities [155], which reflects the conflict between particular details in scenarios and the high level of abstractions required in requirements.

2.2.3 Deriving Specifications from Requirements Using Use Case and Scenario Approaches

In use case and scenario approaches, high-level use cases or scenarios usually capture business processes within organisations [31, 132, 3]. These high-level narratives are then manually re-expressed as low-level use cases or scenarios which capture the direct interaction between a software system and its actors. Once the low-level use cases or scenarios have been re-expressed as the direct interaction between the system and actors,

other techniques, such as UML, are used to generate software design or code [104]. This process is not systematic and is left to the developer's ability and experience.

The main difficulty with use case and scenario approaches is how to transform high-level descriptions into low-level ones. The fact that scenarios have sampling and coverage problems [154] reflects some difficulties for deriving specifications from requirements if scenarios are not complemented by other models.

2.3 Goal-Oriented Approaches

There are two major goal-oriented approaches to requirements engineering, namely the KAOS approach [160] and the *i** approach [166]. Goal-oriented approaches have become popular in requirements engineering because they are useful in acquiring requirements, relating requirements to organisational and business context. They also play some roles in dealing with conflicts and in driving design [167].

The definition of a goal is given by van Lamsweerde as follows [159]: “A *goal* is an objective the system under consideration should achieve. Goal formations thus refer to intended properties to be ensured; they are optative statements as opposed to indicative ones, and bounded by the subject matter [82, 169].”

2.3.1 The KAOS Approach

KAOS is a method for eliciting, specifying and analysing goals, requirements, scenarios and responsibility assignments [38]. It is aimed at providing support for the whole requirements process through elaboration from high-level goals to requirements, objects and assigning operations to various agents. It consists of a specification language, an elaboration method, and meta-level knowledge [160].

Advantages of the KAOS Approach

KAOS's starting points are goals, which can be seen as high-level requirements. They are usually far away from implementation details. They provide an appropriate language to communicate with those stakeholders whose primary concerns are the overall goals or strategies of the organisation, e.g., high-level managers and decision makers [158].

The KAOS approach uses logic to support reasoning about goal refinement with some patterns and tool support, such as GRAIL, which can be integrated with other CASE tools such as DOORS [39], and Objectiver [1].

Disadvantages of the KAOS Approach

KAOS's primary focus is on goals rather than contexts so that the way in which goals are decomposed does not always reflect the complex structures and relationships among requirements and real-world contexts; therefore sometimes a bad goal decomposition will dictate a set of sub-goals that are more difficult or even impossible to satisfy by the software or environment agents.

2.3.2 The i^ Approach*

What is i^ ?*

The i^* framework has been developed for modelling and reasoning about organisational contexts and their information systems. It has two major modelling components: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. SD describes the dependency relationships among actors in an organisational environment; SR describes stakeholder interests, concerns, and how they may be addressed by various configurations of systems and environments [166]. The framework is used in con-

texts where there are multiple parties with strategic interests that may be reinforcing or conflicting each other [165].

Advantages of the i Approach*

The starting point of the i* approach is usually far away from the computing machine. Unlike KAOS, the primary focus of i* are soft goals [29], that is, the so-called non-functional requirements. Since this approach focuses on soft goals, some global non-functional property requirements such as security, usability, performance or flexibility can be expressed as goals for refinement [163]. Since it supports an agent-oriented approach to RE, it has the potential to be linked to agent-oriented programming languages [120].

Disadvantages of the i Approach*

The i* approach shares similar disadvantage of the KAOS approach. Soft goals are difficult to quantify, thus its modelling is mostly a rough approximation to the real world.

2.3.3 Deriving Specifications from Requirements Using Goal-Oriented Approaches

In goal-oriented approaches, requirements are expressed as goals, which may range from high-level goals (e.g., strategic concerns within an organisation) down to low-level operational goals (e.g., technical constraints on the software agent or particular concerns on the environment agent), therefore goal refinement can be seen as a form of requirement transformation [129]. Software specifications are then derived from the subset of operational goals which are assigned to software agents.

In the KAOS approach, goal refinement is made systematic through the association of the goal model with a small set of related models that capture structural and

behavioural aspects of the solution software [106]. For example, scenarios and tabular event-based specifications have been exploited for the elaboration of behavioural models in [160] and [102], respectively. Generic refinement patterns were given in [40] to justify the appropriate reuse of sound goal refinement steps that have been proven formally correct. Therefore, goal decomposition in KAOS can be systematic in the sense that high-level goals (i.e., close to requirements) can be transformed into operational goals (i.e., close to specifications) by following some well-formed refinement patterns.

In the *i** approach, research towards this direction is ongoing, e.g., the Tropos project [113]. According to our literature survey, there is yet to be a systematic way of deriving specifications from requirements in this approach.

2.4 A Formal Approach to Relating Requirements and Specifications -

The Four Variable Model

The *four-variable model* proposed by Parnas and Madey provides a rigorous way of relating requirements and specifications [118]. The model was used for documenting requirements and specifications for the A7-E aircraft using the *Software Cost Reduction* (SCR) method [6], where tabular formalism was applied. The Consortium Requirements Engineering (CoRE) methodology was developed based on the model [50], which was later applied to some avionics systems in the aviation industry [51, 109].

The four variable model consists of the following four variables [110]:

- **MON** - *monitored variable* in the environment that the system¹ observes and responds to;

¹ In this context, the word “system” refers to the software and its I/O devices.

- **CON** - *controlled variable* in the environment that the system controls;
- **INPUT** - *input variable* through which the software senses the monitored variable;
- **OUTPUT** - *output variable* through which the software changes the controlled variable.

The following four mathematical relations are defined under the model [110]:

- **NAT** defines the natural constraints by the environment, such as those imposed by the physical law;
- **REQ** defines the system requirements, dictating how the controlled variable is to respond to changes in the monitored variable, which is to be imposed by the system;
- **IN** defines the relationships of the monitored variable to the input variable;
- **OUT** defines the relationship of the output variable to the controlled variable.

One of the advantages of this model is that it explicitly defines the boundary between the system and its environment and represents them as separate mathematical variables whose relationships must obey some mathematical relations. Its tabular representation and decomposition of complex logic formulas facilitates tool support such as SCR and CoRE methods.

However, as pointed out by Jackson [83], the original four-variable model is suitable for developing software for certain kinds of behaviour control problems. The range of its applicability is restricted mainly because of its underlying assumption that the requirements are always expressed in terms of the monitored and/or controlled variables.

2.5 Problem-Based Approaches

The problem-based approach was started by Jackson's first description of problem analysis in [82], which was later developed more fully in [83]. A problem is viewed as a requirement in a real-world context for which a software solution is sought. The process of software development is then regarded as a problem-solving process, eventually leading to a solution that satisfies the requirement in its context. Central to this approach is the problem frames framework [83], which delivers a whole set of concrete ideas that are usable in guiding problem analysis and associated development in requirements engineering.

In summary, the term "problem-based approach" refers to all the work that shares the same philosophy as Jackson's view on software development [83].

2.5.1 Foundation

The work by Zave and Jackson [169] provides the foundation and motivations for problem-based approaches in requirements engineering. It points out fundamental weaknesses of existing approaches in RE at that time (1997), and states that the following four aspects (the so-called "four dark corners") should be addressed (exact quotes from [169], as listed in italics below):

1. *"All the terminology used in requirements engineering should be grounded in the reality of the environment for which a machine is to be built."*
2. *"It is not necessary or desirable to describe (however abstractly) the machine to be built. Rather, the environment should be described in two ways: as it would be without or in spite of the machine and as we hope it will become because of the machine."*

3. *“Assuming that formal descriptions focus on actions, it is essential to identify which actions are controlled by the environment, which actions are controlled by the machine, and which actions of the environment are shared with the machine. All types of actions are relevant to requirements engineering, and they might need to be described or constrained formally. If formal descriptions focus on states, then the same basic principles apply in a slightly different form.”*
4. *“The primary role of domain knowledge in requirements engineering is in supporting refinement of requirements to implementable specifications. Correct specifications, in conjunction with appropriate domain knowledge, imply the satisfaction of the requirements.”*

The paper then proceeds with a proposal on how the four dark corners can be addressed through problem-oriented requirements engineering, although it falls short of indicating how a requirement engineering process can be built on such a foundation.

Following up from the Four Dark Corners paper, Gunter *et al.* [56] provide formalisation of the work by Zave and Jackson [169], with extended clarifications by Hall and Rapanotti in [60]. Their work focuses on formal models in order to be as rigorous as possible in describing and reasoning about the relationships between requirements and specifications.

Despite the importance of the work on formalisation, the problem with applying formal models to requirements engineering still remains because there is a lot of informality to deal with in requirements engineering. As argued by Jackson in [85, 86], there is always a mismatch between formal modelling and the informal world in the formalisation. Formal models are at best a simplified approximation to the real world. In many cases, the limitations of these models can not be ignored in requirements engineering.

2.5.2 Problem Frames

Problem frames were introduced in Jackson's book *Software Requirements & Specifications: a lexicon of principles, practices and prejudices* [82] in 1995 (they were first mentioned in Jackson's paper [88] a year earlier). A fuller and more systematic representation of problem frames can be found in his later book *Problem Frames: analysing and structuring software development problems* [83] in 2001.

Problem frames propose an approach to describing, analysing and giving early solution to software-intensive problems, such as control, information, business, military or medical systems [35]. Since the work in this thesis is based on this approach, we will describe problem frames more fully in chapter 3.

This approach explicitly separates the solution machine from its environment and the requirement. It provides a graphical notation for representing a problem and its parts. It requires that all descriptions be grounded in the real world, that is, be as faithful as possible to reality, and be the basis of communication with domain experts and users in a language that they can understand: problem owners usually do not have expertise in the computing machine but have experiences or expertise in the application domains.

There is a great emphasis on domain properties, which are the basis for defining the scope of a development project - getting the scope right is crucial to any development [132] [131],

In many ways, the problem frames approach remains an open framework in that it does not prescribe a particular process or description language, thus enabling links to other frameworks or integration with other approaches. It also provides patterns for recognising basic problem classes, which can help solve more complex problems.

Deriving Specifications from Requirements Using Problem Frames

The problem frames approach includes two forms of problem transformation to allow for the derivation of specifications from requirements:

Problem decomposition

Problem decomposition adopts a divide-and-conquer approach to solving a problem: from an initial complex problem, simpler and smaller subproblems are derived. Each solution to the subproblems will contribute to the solution of the original problem. Decomposition may be achieved by matching basic problem frames defined in [83], by applying generic decomposition heuristics [83], or based on specific knowledge of the problem, which requires specific skills of the analyst [35].

Currently, the process of problem analysis is based primarily on problem decomposition guided by heuristics until the problem becomes so simple that we can define the specifications. No systematic techniques have been provided to support the process.

Problem progression

Problem progression is an idea given in [83] which is a form of transforming problem contexts and requirements so that the analysis of the problem can be progressed towards the computing machine. However, not many details are given, and it remains an under-explored area of the problem frames approach [35].

This thesis gives a definition of problem progression, develops associated techniques, and applies them to several case studies. Details on problem progression will be given in Chapter 3.

2.5.3 Problem-Oriented Software Engineering

Recently, Hall *et al.* [64] have proposed Problem-Oriented Software Engineering (POSE) which extends and generalises Jackson's problem frames in the following way: it permits various forms of solution descriptions that stretch to different levels of abstraction, such as from high-level specifications, design down to low-level code; it supports architectural structuring of the solution space; the process of problem solving is transformational, providing traceability between problem and solution domains and is accompanied by adequacy justification of the transformation. Hence, POSE stretches from requirements engineering through to program code. Development in POSE is stepwise with transformations by which problems are moved towards software solutions. The framework takes the form of a *sequent calculus* in the Gentzen style [95], in which both formal and informal steps of software development are accommodated.

2.6 Transformational Approaches in Software Engineering

As observed by Rapanotti *et al.* in [127], since the late 1970s, many approaches to software development have been focusing on the transformation of software specifications into code using techniques and processes that work within the solution domain. For example, many formal approaches to software development have been focusing on logic and calculi. Representatives of such approaches are Feather's approach to formal specification of closed systems using the language *Gist* [52], the refinement calculi of Morgan [112] and Back *et al.* [9], and the categorical refinement of Smith [150]. Some recent developments in automatic tool support have given hopes of achieving large scale program verifications [70, 71, 94]. However, to what extent these formal techniques are suitable for the systematic derivation of specifications from requirements remains, by

and large, an open question. Chapter 4 of this thesis explores this issue and gives some observations and arguments on one particular formal technique.

Many researchers have also explored transformational approaches in requirements engineering. For example, Johnson's work on deriving specifications from requirements [92] proposes automated support for transforming requirements into specifications. He has defined a language [91] for the description of requirements and environmental properties, from which simulations of the behaviour of the system and environment can be derived. Jackson and Zave [87] give some elements of a method for transforming requirements to specifications, and illustrate them with an example. We share in this thesis much of the principled basis of their approach. The work in this thesis (in particular chapter 5) embodies such principles as practical techniques for transforming requirements into specifications in the context of problem frames.

2.7 Summary

This chapter has reviewed major current approaches in requirements engineering and examined how each of them contributes to the systematic transformation of requirements into specifications. Results of the review suggest the following points. Use case and scenario approaches need to be complemented by other models to derive low-level scenarios from high-level ones due to their sampling and coverage problems [155]. One variant of goal-oriented approaches, KAOS, has some well-formed patterns to help the systematic derivation of operational goals from high-level goals, but transformation of contexts is not explicit in goal refinement. The problem frames approach explicitly allows for the transformation of both requirements and contexts, but systematic transformational techniques are currently missing from this approach. To fill this gap in

the problem frames approach, this thesis provides some transformational operators and rules (which will be defined in later chapters) for one class of problem transformation - problem progression.

3. PROBLEM PROGRESSION

In this chapter we describe problem progression and its conceptual basis in problem frames. We describe only those aspects of problem frames that are relevant to our work. A more complete presentation can be found in [83].

3.1 *The Problem Frames Approach*

The idea of *problem frames* was published in Jackson's book *Software Requirements & Specifications* [82], in which it was outlined as one of a small number of topics related to software development. He gave a more systematic account of problem frames later in *Problem Frames: Analyzing and Structuring Software Development Problems* [83].

For more than a decade, researchers in the requirements community have explored and extended problem frames into a conceptual framework for requirements engineering (see [35, 36, 62] for collections of recent work). This framework suggests a principled approach to software development. As Jackson puts it [84], "The problem frames approach is not a development method. It is, rather, a perspective and a conceptual framework, embodying a certain way of looking at an important group of problem classes and of structuring the intellectual processes of developing good solutions."

In this thesis, "the problem frames approach" is used interchangeably with "the problem frames framework"; we will simply use "problem frames" or "PF" to represent both in most situations. Wherever we need to make the meaning explicit, we often

prefer the phrase “the problem frames approach”.

3.1.1 *The Engineering Root of PF*

The problem frames approach takes an engineering view of software development. For example, in [82] Jackson gives an account of various possible aspects of software development, such as the concerns and expertise voiced by the mathematician, the financier, the management, the sociologist, the lawyer, or the stockbroker, etc. He argues that although each of them may play a crucial role in certain development projects, yet the central point of all software development should be the task of the software engineer. He points out that as software engineers, “our business is engineering - making machines to serve useful purposes in the world. And our technology is the technology of description”.

Of course, this does not mean that the knowledge and expertise of mathematicians, financiers, project managers, sociologists, lawyers or stockbrokers are ignored by engineering. In fact, they can be elicited from these domain experts as domain knowledge, which is an important part of requirements engineering. In PF, they are encoded as domain properties. PF does not prescribe any particular language for describing them so as to accommodate a variety of languages used by these experts.

This engineering perspective is emphasised and elaborated again by Jackson in [84] based on Rogers’ definition of engineering [134], which is quoted and expanded by Vincenti [161]:

“Engineering refers to the practice of organizing the design and construction of any artifice which transforms the physical world around us to meet some recognized need.”

In the PF view of software development, Jackson [84] interprets the artifice to be designed and constructed as the *machine*, on which software is built and executed to serve a particular purpose. The purpose is to satisfy a recognised need, which is called *requirement*. In order to satisfy the requirement, we need the machine to transform part of the physical world around us, which is called the *problem world*. The satisfaction of the requirement can be observed only in the problem world, therefore PF views the requirement as existing only in the problem world.

In PF, problem descriptions are captured and expressed by diagrams (notations will be introduced later), which model the machine, the problem world, the requirement, and their relationships. Moreover, in order to serve engineering purposes, PF also provide tools to help analyse problems and derive solutions, such as problem decomposition, subproblem recomposition, and dealing with some standard concerns that arise in the analysis process [83].

3.1.2 Representing Problems

Phenomena - The Most Basic Elements of Problem Descriptions

In order to describe the problem world in a way that facilitates understanding and communication, Jackson proposes the notion of *phenomenon* as the basis of descriptions. He defines a phenomenon to be “an element of what we can observe in the world” [83].

The word “element” implies that phenomena provide the fundamental vocabulary or alphabet for describing the world, in other words, identifying all the relevant phenomena in the context provides enough basic elements for describing the problem at hand.

Of course, as Jackson argues in [84], abstractions are unavoidable in any treatment of physical phenomena. We can write abstract phenomena as long as they can be un-

ambiguously explained in terms of phenomena that we can observe. For example, if we regard the pressing of a button by a lift user to be a phenomenon in a problem addressing the specification of a lift controller, then we are making abstractions from a chain of causal events which start from depression of the button all the way to, let us assume, assigning the corresponding encoded value to a machine register. We can consider this complex chain of causal events as a single event at certain higher level of abstraction, provided that we can unambiguously interpret it using observable phenomena: in our example, when a lift user presses a button, the physical movement of the button connects the associated circuit, which sends an electronic signal through the cable connected to the controller machine, which then matches one of the predefined key codes, for which an encoded value is correspondingly assigned to a register of the machine. This abstraction of phenomena is not only convenient for communication but also powerful in controlling complexity in analysis and design [45].

According to Jackson [83], phenomena consist of *individuals* (something that can be named and distinguished from others) and *relations* (a set of associations among individuals):

- An *individual* can be an *event* - an occurrence at some point in time, regarded as atomic and instantaneous, e.g., a keystroke; or an *entity* - something that can persist or change over time, e.g., a motor car; or a *value* - something that can not change over time, e.g., the character "X" or the number 23.
- A *relation* can be a *state* - a relationship among individual entities and values that can be true at one time and false at another, e.g., *Temperature(Room, 29.5)*; or a *truth* - a relationship among values that is either true at all times or false at all times, e.g., *LengthOf("ABCD", 6)*; or a *role* - a relationship between an event and

its participating individuals.

Jackson [83] introduces two categories of phenomena: *causal phenomena* are directly caused or controlled by some domain, and they can cause other phenomena in turn, e.g., a pulse event in a traffic light unit can cause a state change in the Stop and Go lights; *symbolic phenomena* are used to symbolise other phenomena and relationships among them because they can neither change themselves nor cause changes elsewhere, though they can be changed by external causation, e.g., the data content of a floppy disk record. As we will see later, a large part of this thesis focuses on causal phenomena and associated cause-and-effect relationships.

Domains are an abstraction of phenomena grounded in the real world: a *domain* is defined to be “a set of related *phenomena* that are usefully treated as a unit in problem *analysis*” by Jackson [83]. Another characteristic of a domain is that it is usually a concrete and self-contained artefact that maps to domain experts’ intuition and knowledge on how they partition the problem world into well-understood parts whose phenomena are potentially relevant to the problem. PF makes an explicit distinction between the *internal phenomena* and the *external phenomena* of a domain. The internal phenomena of a domain are private to the domain un-shared with other domains; while the external phenomena are shared with other domains.

Jackson’s classification of domains is also based on phenomena [83]: a *causal domain* is one whose properties include predictable causal relationships among its phenomena; a *biddable domain* usually consists of people, and it lacks positive predictable internal causality among its phenomena; a *lexical domain* is a physical representation of data of symbolic phenomena (a lexical domain can be regarded as a structure of symbolic phenomena, or as a causal domain).

Similarly, a requirement is grounded in the real world since it is “a condition on one or more *domains* of the *problem context* that the *machine* must bring about” [83]. In other words, the only way that we (including the customer and the developer) can judge if the requirement is satisfied is by observing the desired phenomena in the real world.

From above, we can see that every artefact in PF is an abstracted form of phenomena with certain characterisation; therefore phenomena are the building blocks for PF. Any reasoning or analysis in PF is based on phenomena descriptions.

Adopting the notion of phenomenon to describe the problem world has at least two advantages:

- Descriptions that are based on phenomena are firmly grounded in the real world. In PF, the soundness of complex phenomena descriptions can be validated with domain experts by elaborate structures of “designations” and “refutable descriptions” [81, 82]: a *designation* refers to the relationship between a phenomena description and what it describes in the real world, thus allowing for informal explanation of how the phenomena can be recognised; a *refutable description* says something about the problem world that can, in principle, be refuted by finding a counter example of the description.
- It is a way of allowing certain important stakeholders (e.g., domain experts) to be involved early in establishing the problem scope for analysis [107].

Domain Properties - Indicative Relationships among Phenomena

A domain is defined to be an encapsulated set of related phenomena and its properties are the inherent (or indicative) relationships among its internal and external phenomena. In Jackson’s own words, *domain properties* are “the expected and assumed relationships

among the *phenomena* of a *domain*” [83].

In many realistic software development problems, several phenomena are not shared directly with the solution machine, but they still need to be affected indirectly by the machine for the requirements to be satisfied. As we will see later, domain properties are important for bridging the gap between phenomena that are directly shared with the solution machine and those phenomena that are not.

Problem Diagrams - Schematic Organisation and Scalable Abstraction of Phenomena Descriptions

In any problem descriptions, the scope of observable phenomena needs to be established. PF provides a graphical notation to express the scope of a problem and its parts. A *context diagram* shows the structure of the problem context in terms of domains and connections between them [83]. A *Problem diagram* augments the context diagram with a representation of the requirement. An example of problem diagram is given in Figure 3.1.

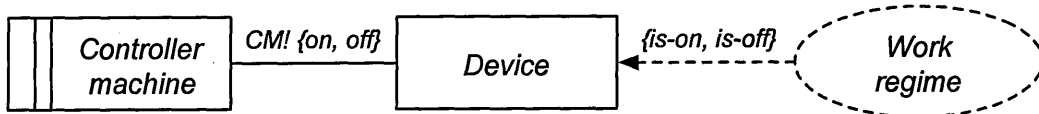


Fig. 3.1: A simple problem diagram taken from [35] modified

Figure 3.1 shows a control problem in which the machine is to control a device for a work regime. There are the following basic elements of a problem diagram:

- The machine domain named *Controller machine* is represented by a box with a double stripe; the device domain named *Device* is represented by a box with no stripe; the requirement named *Work regime* is represented by a dashed oval.

- The shared phenomena between the *Controller machine* domain and the *Device* domain are represented by a solid line connecting them, identified by *CM!* {*on*, *off*}, where *CM!* represents that these shared phenomena are controlled by the machine (in other words, they are observed by the device). There is a convention to follow about shared phenomena in a problem diagram containing many domains - if there is no line linking two domains in a the diagram, then it is assumed that they do not directly share any phenomena - this implicit convention is important for any problem analysis.
- The fact that the requirement *Work regime* constrains certain internal phenomena of the device is represented by a dashed line with an arrowhead pointing towards the *Device* domain, identified by {*is-on*, *is-off*}.
- Phenomena {*on*, *off*} are known as *specification phenomena* because they are shared with the machine; while phenomena {*is-on*, *is-off*} are known as *requirement phenomena* because they are the subject of the requirement references [83].

Problem diagrams provide a schematic organisation of the phenomena that are within the scope of the problem to be solved. Their roles in describing problems are twofold:

- on the one hand, they help visualise the topological complexity of the software development problem (depending on the complexity of the problem, an arbitrary number of application domains with varied connections could be drawn in the same problem diagram, see Chapter 6 for more complex examples);
- on the other hand, for clarity of the model, they omit details of the domains' internal phenomena unless they are referred to or constrained by the requirement.

Problem diagrams are complemented by problem descriptions with details about the domain's internal phenomena.

Basic Problem Classes and Frames

Central to the PF approach is the idea of providing a catalogue of recurrent software problems for reuse. Essentially, a *problem frame* is a recurrent problem template representing a problem class. Jackson [83] introduces five basic frames as an initial catalogue of identified problem classes for structuring and decomposing complex problems and their solution.

Practitioners can follow the same principle and build their own repertoire of problem patterns as their ability to solve problems grows over time. For example, the problem frames community have found new problem frames such as the *user interaction frame* [61], the *simulator frame* [17] and the *pipe-and-filter* or *model-view-controller* AFrames [126].

Next, for the purpose of illustration, we will look closely at the *required behaviour frame*, which is one of the five basic problem frames.

The required behaviour frame - an example

The problem described by the required behaviour frame is: "There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control." [83]. The graphical representation of a problem frame is called a *frame diagram*. The frame diagram associated with the required behaviour frame is given in Figure 3.2.

In the diagram, *C1*, *C2*, *C3* are causal phenomena; the *C* annotation in the bottom right corner of the *Controlled domain* represents the fact that the *domain type* of this

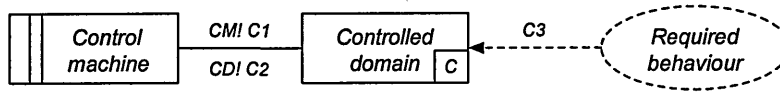


Fig. 3.2: Required behaviour: frame diagram (taken from [83] unmodified)

domain is causal; the annotation *CM!* represents the fact that the shared phenomena *C1* are controlled by the *Control machine* domain (e.g., this is where the machine can exert control); the annotation *CD!* represents the fact that shared phenomena *C2* are controlled by the *Controlled domain* (e.g., this is where the machine gets the feedback about the controlled domain); and the dashed arrow line labelled *C3* constrains certain internal causal phenomena *C3* of the *Controlled domain*.

The above diagram is a template for recurrent problems. In order to match a problem diagram to this template, the domain types, the phenomena types, and the control and observation characteristics of the phenomena have to be the same.

A *frame concern* [83] is an argument that we must make, by fitting descriptions of the requirement, the machine and the problem domains together, to convince our customers that the requirement is adequately satisfied. The frame concern is expressed diagrammatically, as shown in Figure 3.3 for the required behaviour frame.

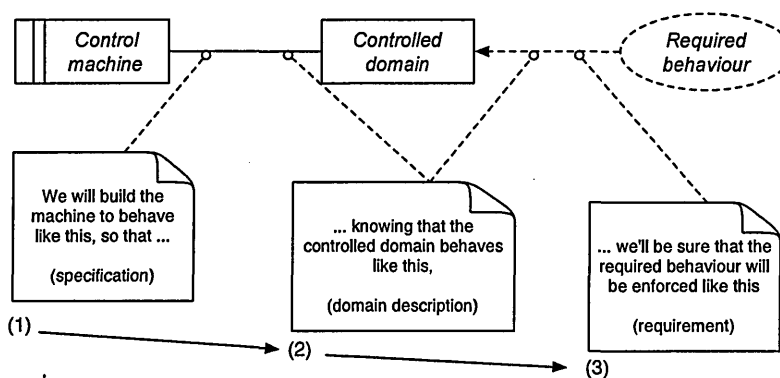


Fig. 3.3: The frame concern for the required behaviour frame

Here is an example of a required behaviour problem - a simple automatic temperature control problem.

A modern office building needs an automatic heating control system during the cold winter months in a year. The building has a fixed pattern of usage - the building needs heating on every working day from 9:00 am till 5:00 pm, which are the regular working hours in the offices. The problem is to build a simple controller machine that will switch on the heating devices (we assume the heating devices have a mechanism to maintain the temperature) at 8:45 am and switch them off at 4:45 pm every day.

Figure 3.4 shows the problem diagram for this problem:

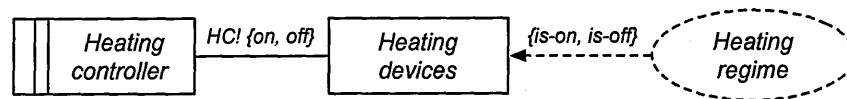


Fig. 3.4: A simple automatic heating control problem diagram

Heating devices: devices used to generate heat. They can be in either the *is-on* state or the *is-off* state. Pulse events *on* and *off* can affect state changes, thus this domain is a causal domain.

Heating regime: the requirement is that the heating devices should be on between 8:45 am and 4:45 pm every day.

{on, off}: these are shared phenomena between the *Heating controller* domain and the *Heating devices* domain; *HC!* means that the phenomena are controlled by the *Heating controller* domain; they are the specification phenomena.

{is-on, is-off}: these are the requirement phenomena, which happen to be internal to the *Heating devices* domain; *{is-on, is-off}* represent the two states of the *Heating devices*: *is-on* represents the devices being on; *is-off* represents the devices being off.

The task of problem analysis is to find a machine behaviour that will make the heat-

ing devices do what is required. Once a machine behaviour is found, the frame concern captures the form of argument we need to have in terms of all problem descriptions. Therefore, addressing the frame concern adequately means making sure that requirement, domain and machine specification descriptions match properly, and the problem is solved.

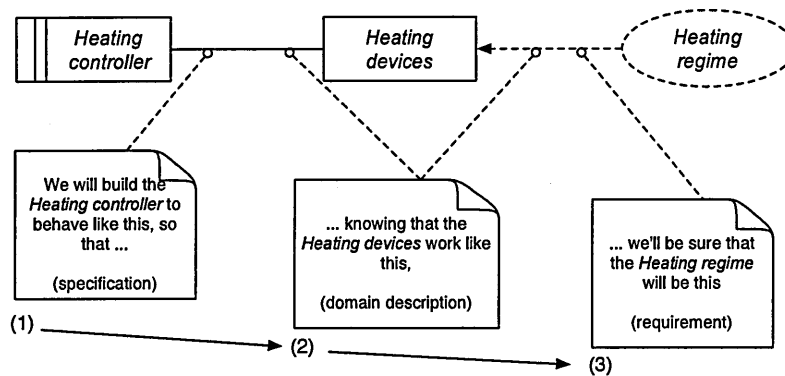


Fig. 3.5: Frame concern in the heating control problem

For this heating control problem our descriptions must support the argument shown in Figure 3.5, that is, we must establish that the specified behaviour of the *Heating controller* (1), combined with the domain properties of the *Heating devices* (2), will adequately achieve the required behaviour - the *Heating regime* (3). A controller specification which would allow us to make such an argument is:

Heating controller: the heating controller machine should send an *on* pulse at 8:45 am and send an *off* pulse at 4:45 pm every day.

And the argument is:

(1) We will build the *Heating controller* to behave like this: “the heating controller machine should send an *on* pulse at 8:45 am and send an *off* pulse at 4:45 pm every day”, so that ...

(2) ... knowing that *Heating devices* work like this: “*devices used to generate heat. They can be in either the is-on state or the is-off state. Pulse events on and off can affect state changes*”,

(3) ... we’ll be sure that the requirement *Heating regime* will be this: “*the heating devices should be on between 8:45 am and 4:45 pm every day*”.

The advantage of using the basic frames is that we can utilise the expertise of others and the structured analysis that has been proven useful in software development. In other words, the basic frames give a template of the problem and an associated argument template for us to use. However, for realistic problems that do not necessarily fit any of the basic frames, we need to find other ways of solving them.

One approach proposed in [83] is to decompose the problem into a combination of simpler subproblems that match basic frames. Then the solutions to these subproblems are eventually recomposed into a machine specification.

Another approach, which is the subject of this thesis, is to transform the complex problem into something that is more amenable to solution (something that we are more familiar with or have previous experiences in solving, but which does not necessarily fit a basic frame), while preserving the requirement traceability [54] expressed in the problem diagram by following some systematic rules (as suggested in [129]).

3.1.3 Transforming Problems

According to Jackson [84], although the PF does not prescribe particular steps of development, we can imagine a development process where we begin by capturing the customer’s requirement, and proceed with the given domain properties to devise a machine behaviour specification. Part of this process is what Jackson calls *problem progression (or reduction)* - starting from an overall problem involving all the observable

phenomena in the problem world, we need to derive a *reduced problem* where only the specification phenomena are left. At this point, Jackson suggests: “a problem of engineering in the world has been reduced to the problem of building a machine with a specified external behaviour” [84].

In this thesis, we consider problem progression as starting from a situation where the problem world consists of a complex structure of interacting problem domains. We propose ways of progressing the problem in a stepwise manner by successively removing domains which are farthest from the machine and re-interpreting the requirement appropriately. In other words, we propose ways of deriving specifications from requirements in a systematic fashion. This is reminiscent of the work on deriving code from specification [70, 72]. The similarity between the two can be summarised as follows: the main purpose of the former is to provide a systematic way of deriving a specification that satisfies the customer’s requirements; the main purpose of the latter is to systematically derive code to satisfy a specification. The two notions complement each other within an overall development process.

Tools for Problem Analysis

In problem frames, a number of tools have been given for problem analysis:

Problem decomposition through projection [83]: also known as the “divide-and-conquer” principle in solving complex problems; the idea is to apply some heuristics or previous knowledge in order to divide the overall problem into a finite number of projected subproblems that are easier to solve than the overall problem. Moreover, very often the subproblems are fitted to basic frames. This projection is different from partitioning the overall problem, as illustrated in Figure 3.6: in problem decomposition through projection, the relationships among subproblems are like those among A, B, C

and D on the left-hand side of the figure, e.g., subproblems A and B may include the same domains or shared phenomena in their overlap area; in contrast, in partition, as in the right-hand side of the figure, the overall problem is partitioned into non-overlapping subproblems.

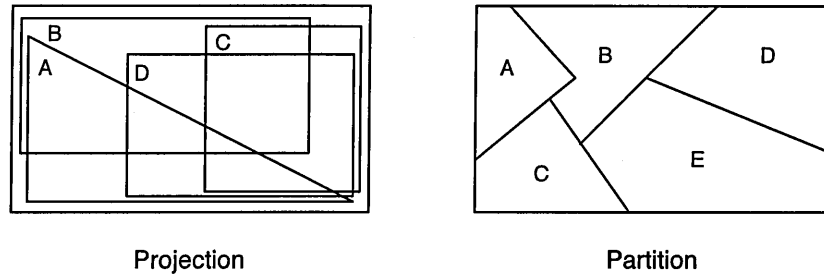


Fig. 3.6: A comparison between projection and partition (taken from [83] unmodified)

Problem variant [83]: this includes Jackson's treatment of variant problems. A *variant frame* is a variant of a basic problem frame in which an additional problem domain is added, or the control characteristics of a shared phenomenon are changed. Four variants are introduced to deal with problems that do not fit the basic frames, namely, by adding connection variants, description variants, operator variants to the problem diagram or elaborating control variants in the diagram [83].

AFrames: Hall *et al.* [61, 126] have introduced architectural frames (known as AFrames) as the means to apply architectural patterns to identify subproblems based on standard solution architectures. One of the merits of this approach is that both problem decomposition and subsequent recomposition are addressed at the same time.

Problem progression: this will be discussed in detail in the next section as it is the subject of this thesis.

3.2 Problem Progression and its Significance

The idea of problem progression was briefly explained in [83], reflected in Figure 3.7.

In the words of Jackson:

“You can think of any problem [expressed in PF] as being somewhere on a progression towards the machine, like this:

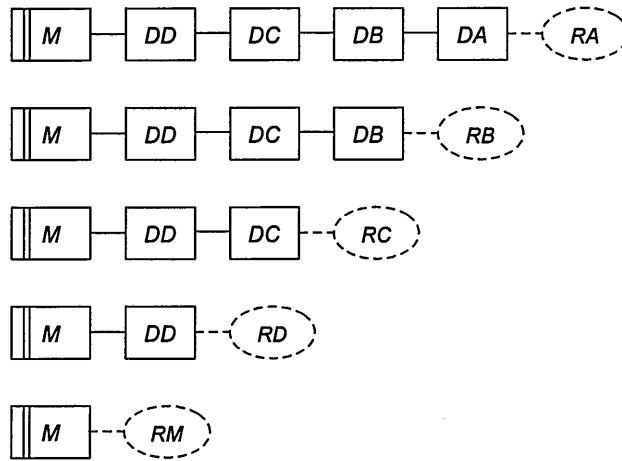


Fig. 3.7: A progression of problems (taken from [83] unmodified)

The top problem is deepest into the world. Its requirement RA refers to domain DA. By analysis of the requirement RA and the domain DA, a requirement RB can be found that refers only to domain DB, and guarantees satisfaction of RA. This is the requirement of the next problem down. Eventually, at the bottom, is a pure programming problem whose requirement refers just to the machine and completely ignores all problem domains.”

This discussion emphasises that we cannot just look at software development problems very close to the machine. We should look at the problem in its wider context. When we are solving problems that are very close to the machine, we have to make sure

that the solution satisfies the wider problem.

The above general principle of moving closer to the machine by analysing assumptions about deeper contexts in relation to the requirements is valuable because it enables the possibility for a problem analyst to move systematically from an unfamiliar problem to a familiar problem: the closer you get to the machine, the easier it becomes for the software developer to apply his or her expertise, thus the more familiar the problem becomes. Diagrammatically in Figure 3.7, the solution to each of the problems is represented by the same machine *M*. This indicates that from the initial problem at the top, we transform each problem in a solution-preserving way: that is, the solution to the progressed problem satisfies the original problem. Note that in each step of the transformation, we change the requirement to compensate for the reduced context by making appropriate assumptions. This is required to guarantee that the solution to the progressed problem will satisfy the initial problem when embedded in the wider context.

In order to explain our interpretation of problem progression in [83], let us take the heating control problem as an illustrative example of problem progression:

Recall that the requirement *Heating regime* is “*the requirement is that the heating devices should be on between 8:45 am and 4:45 pm every day*”, and the heating devices’ domain properties are: “*Heating devices: devices used to generate heat. They can be in either the is-on state or the is-off state. Pulse events on and off can affect state changes, thus this domain is a causal domain*”. It follows that in order for the heating devices to be *is-on* between 8:45 am and 4:45 pm every day, they have to be switched on at 8:45 am (caused by the pulse event *on*) and switched off at 4:45 pm (caused by the pulse event *off*). Then we can re-express the requirement as the specification *Controller commands*: “*the heating controller machine should send an on pulse at 8:45 am and send an off pulse at 4:45 pm every day*”. The transformation is carried out in such a

way that it takes into consideration domain properties of the heating devices so that the solution *Controller commands* will work in the initial problem (expressed by the top diagram in Figure 3.8).

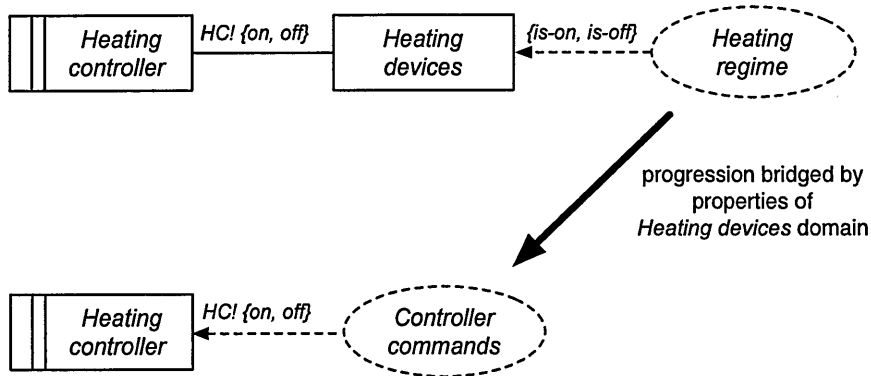


Fig. 3.8: Problem progression for the simple automatic heating control problem

Problem progression is not well-developed in Jackson’s original book [83], but only mentioned as an idea in one of the question-and-answer sections of the book. In this thesis, we take this idea forward by working out the details of transforming both the requirement and the problem context. Therefore, we claim that the work in this thesis contributes to this idea in a practical and constructive way.

Very recently, Seater and Jackson [148] have done some related work on deriving specifications from requirements in the context of problem frames, in which the requirement is transformed into a specification, and, as a by-product of the transformation, a record of domain assumptions, which they call “breadcrumbs”, are produced as justification for the progression. The focus of the transformation is on rephrasing the requirement progressively until it is expressed as a machine specification, while developing domain assumptions which make the requirement transformation sound. They call such transformation “requirement progression” as their focus is rewriting the re-

quirement rather than transforming the whole problem as we do in this thesis. Also Seater and Jackson's work is focused on Alloy [79], a first-order logic modelling language, while we apply a wider range of techniques from fully formal, based on Hoare's CSP, to semi-formal, based on causal reasoning.

3.3 *Summary*

In this chapter, we introduce Jackson's idea of problem progression based on the problem frames framework. We take the idea of problem progression forward by exemplifying how progression can be carried out in practice on an example. In the next two chapters, we will develop two classes of techniques to systematically support problem progression.

4. A FORMAL APPROACH TO PROBLEM PROGRESSION

As introduced in the previous chapter, problem progression is a type of problem transformation that is carried out in a solution-preserving way. It is captured and represented by a series of related transformed problem diagrams. Given this conceptual basis, our aim is to find practical ways to interpret it so that constructive techniques can be applied to its implementation. The next two chapters will give two complementary approaches to problem progression and show how constructive techniques can systematically help solve problems.

Our first formal approach adopts CSP descriptions and operators. We show how CSP can be used as a description language for problem diagrams, and then derive a CSP-based semantics for them. This allows certain constructive CSP operators from the literature to be used to progress problems. We then apply the technique we develop to an example problem to show how our formal approach to progression works.

We begin the chapter by formulating the example problem which will be used for illustration throughout, and conclude the chapter with a discussion of the limitations of the use of formally based techniques in problem progression, arguing the need for further and less formal approaches.

4.1 An Example

The example is that of a supermarket point-of-sale (POS) system which allows customers to scan and pay for their shopping without any intervention from supermarket staff¹. The problem is described as follows:

A Self-Checkout POS System

A new point-of-sale (POS) system is needed to process sales for a supermarket shop in the UK. The POS includes both the desired software and some hardware purchased from a third party, including a barcode reader, a cash acceptor and dispenser handler, a touch-screen display, and a receipt printer, etc. The problem is that customers should pay for and receive a receipt for the correct amount on presentation of items to the POS system.

Table 4.1 shows the identified domains and their informal descriptions for this problem.

Name	Description
<i>CUST</i>	A person who wants to buy an item from the shop.
<i>POS</i>	The system which includes the desired software and the hardware purchased from a third party, such as a barcode reader, a cash acceptor and dispenser handler, a touch-screen display, and a receipt printer, etc

Tab. 4.1: Domains and their descriptions

¹ This type of POS has recently appeared in many UK supermarkets.

A problem diagram for the self-service POS system is given in Figure 4.1.

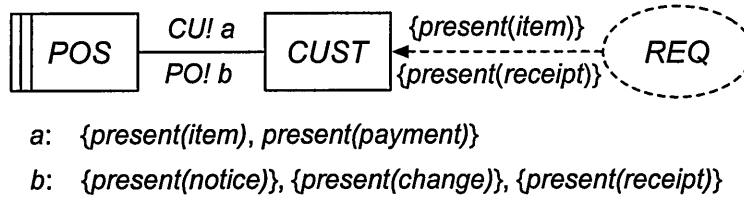


Fig. 4.1: Point-of-sale: problem diagram

Table 4.2 shows the shared phenomena between domains in Figure 4.1 and explains their designations in natural language.

Name	Designation
<i>present(item)</i>	The event in which the customer presents an item of product s(he) wants to buy to the <i>POS</i> system. This event is initiated and controlled by the customer <i>CUST</i> domain, thus represented by <i>CU!</i> that proceeds it.
<i>present(payment)</i>	The event in which the customer presents the payment for the purchased item to the <i>POS</i> system. This event is initiated and controlled by the customer <i>CUST</i> domain, thus represented by <i>CU!</i> that proceeds it.
<i>present(notice)</i>	The event in which the <i>POS</i> system presents a notice to the customer. This event is initiated and controlled by the <i>POS</i> domain, thus represented by <i>PO!</i> that proceeds it.
<i>present(change)</i>	The event in which the <i>POS</i> system presents the change due to the customer. This event is initiated and controlled by the <i>POS</i> domain, thus represented by <i>PO!</i> that proceeds it.
<i>present(receipt)</i>	The event in which the <i>POS</i> system presents a receipt to the customer. This event is initiated and controlled by the <i>POS</i> domain, thus represented by <i>PO!</i> that proceeds it.

Tab. 4.2: Shared phenomena and their designations

The requirement statement represented by *REQ* is: “customers should pay for and receive a receipt for the correct amount on presentation of items to the *POS* system.”

4.2 Semantics of Problem Diagrams

Hall *et al.* [63] provide a denotational semantics of problem diagrams defined as follows. Let us consider the problem diagram in Figure 4.2.

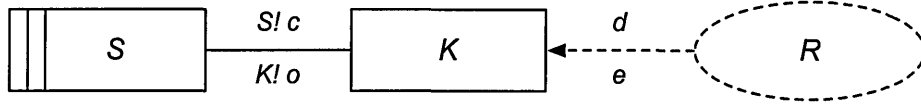


Fig. 4.2: A generic problem diagram, K may be arbitrarily complex

The semantics assumes descriptions of the diagram are expressed in a language, called the *Domain and Requirement Description Language* (DRDL). The only requirement which is made of this language by the semantics is that it has a notion of satisfaction. The meaning of a problem diagram is that of a “challenge” to find a specification S that satisfies R in the context of K , and is denoted by the set:

$$c, o : [K, R] = \{S : \text{Specification} \mid S \text{ controls } c \wedge S \text{ observes } o \wedge K, S \vdash_{\text{DRDL}} R\}$$

In the set definition, “observes” and “controls” have the usual PF meaning, and \vdash_{DRDL} indicates satisfaction as defined in the chosen DRDL.

Formally, the above formula denotes the set of all possible solutions to a generic problem diagram. A limitation of the above semantics is that the formula is not constructive: we do not know how to calculate an element of the set. For example, the semantics does not tell us how to solve our example problem in section 4.1.

To solve the problem formally, we need to find techniques within a formal framework that allow us to calculate and construct a precise solution specification based on the semantics. The techniques should give more technical insights and guidance to pop-

ulate the solution set. Also we need to be able to address more complex problems than that of Figure 4.2, with problem diagrams containing an arbitrary number of interacting domains. To be able to progress these problems formally, we need a technique that captures this complexity and supports a process of reducing it by formal transformation.

To summarise, in this section we have chosen a formal interpretation of a generic problem diagram and its solution specification as a set, based on Hall *et al.*'s semantics. In the remainder of this chapter we will define a constructive approach to calculate an element of the set semantics, that is, a formal solution specification for a problem like that in section 4.1. In the next section, we will choose a restricted form of CSP as a DRDL. We will formalise various artefacts in a problem diagram into various CSP descriptions, and then find constructive operators for progressing problems based on such descriptions.

4.3 Formalising a Problem Diagram Using CSP as a DRDL

In the following, we will give a brief introduction to the relevant CSP concepts we are going to use to consider CSP as a DRDL, so that we can use it as the basis for problem progression. Note that CSP is a very rich language and we will only use a subset of it for our purpose.

4.3.1 The CSP language

Hoare's *Communicating Sequential Processes* (CSP) [68] is a formal description language used in software engineering. Although its original purpose was to describe concurrency in programming [67], it has evolved and been applied to other areas of software engineering: for example, modelling and analysis of security protocols [141],

specifying software architecture connections [4], describing system level interactions between software and hardware [135], and software verification [72, 70]. It also influenced the development of the Occam programming language [78]. Recently, since its event-based notations can map to real-world events, a small subset of CSP-like notations have been used to model the interactions between the computer system and its environment to satisfy human-computer interaction requirements [65]. In software engineering practice, the CSP tools FDR (Failures-Divergence Refinement) and ProBE developed by *Formal Systems (Europe) Ltd.* [77] have been applied to industrial-scale projects, such as security systems [58], hybrid systems [119] and model-checking [37].

The theory of CSP has undergone many revisions and extensions, whose milestones are represented by several classical books: the early work is outlined in Hoare's book *Communicating Sequential Processes* [68], which introduces the basic concepts of the CSP language. Later Roscoe extended Hoare's work on CSP foundations, semantics, and tool applications in his book *The Theory and Practice of Concurrency* [137]². A more recent book, *Concurrent and Real-time Systems: The CSP Approach* [146], by Schneider introduces the main aspects of modern CSP, adding more CSP models and introducing timed CSP. It uses an operational semantics to explain CSP operators and adopts real-world examples and exercises to make it more suitable and accessible for education to a wider audience.

In this chapter we choose CSP as a DRDL in the formalisation of problem diagrams and their semantics, based on which some CSP operators are chosen for the formal construction of the solution guided by problem progression.

² The CSP used in Hoare's book [68] is considered as the first version, and the one used in Roscoe's book is regarded as the second version [137].

Basic Concepts, Definitions and Notations

CSP provide notations suitable for describing and analysing real-world systems which consist of interacting components. As summarised in [146], the view taken by CSP for analysing the world is that of regarding each of these interacting components as a *process*, that is, an independent and self-contained entity with particular interfaces through which it interacts with its environment. If two processes are combined to form a bigger system, then their combination becomes a self-contained entity with a particular interface, i.e., a bigger process. This highlights the fundamental view of this framework that processes are compositional in nature, for example, Kramer observes that CSP supports compositional analysis [97].

The following definitions and conventions are adopted for the meaning and basic syntax of events, processes and alphabets [68, 146]:

- An *event* is an *atomic action* that can be performed or suffered by an entity (or object) in the world. An event is denoted by a single lower-case letter, e.g., *a*, *b*, *c* or a lower-case word, e.g., *coin* - a coin is inserted in the slot of the vending machine, *choc* - a chocolate is extracted from the dispenser of the vending machine;
- A *process* is an independent and self-contained entity (Hoare called such entities “objects in the world around us” [68]) with a particular set of events, through which it interacts with its environment. A process is denoted by an upper-case word or acronym, e.g., *VMS* - simple vending machine, *USR* - user, or a single upper-case letter *P*, *Q*, *R*;
- An *alphabet* is the set of events that are relevant for a particular description of

an entity. An alphabet is denoted by adding α before a process name, e.g., $\alpha VMS = \{coin, choc\}$ - the simple vending machine has in its alphabet two different classes of events³, *coin* and *choc*. Note this choice of alphabet ignores some other possible classes of events, e.g., the maintenance of the vending machine could require that *loadchoc* and *emptycoin* events. Choosing what should be included in the alphabet of a process depends on the assumptions made about its context and may have a significant impact on the analysis.

Basic CSP Syntax

The following describes some basic CSP syntax that we will use (adapted selectively from [68, 146, 137]):

- $STOP_A$ is a special process which does nothing and never engages in an event in its alphabet A (A can often be omitted if it's clear from context what events A contains).
- $CHAOS_A$ is a process which can always choose to engage in or reject any events in A . It is regarded as the least predictable and the least controllable process.
- *Event Prefix*: If P is a process and an event a is in P 's alphabet, then the new process $a \rightarrow P$ can be constructed. It is a process that is initially able to perform only a , then afterwards it behaves as P . For example, a partial behaviour of a simple vending machine that consumes one coin and serves one chocolate can be described as $coin \rightarrow choc \rightarrow STOP$.
- *Communication*: When a is an event between the process P and its environment, it is usually denoted in the $c.v$ format, where c represents a *communi-*

³ There may be many *occurrences* of events belonging to these two classes.

cation channel and v represents the *value* being sent or received by P via c . For a process that engages in a communication, the process either accepts an input variable x on channel c , denoted $c?x$, or outputs the value e on channel c , denoted $c!e$. For example, in the above simple vending machine, *coin* is regarded as an “input” event and *choc* is an “output” event, and the “values” can be a *1pound* coin and a *200g* chocolate bar, respectively, so we can write $coin? 1pound \rightarrow choc! 200g \rightarrow STOP$.

- **Event Prefix Choice** is a process that is initially prepared to perform any of the prefix events of more than one possible process choices prefixed by different events. The actual behaviour of this process depends on which prefix event actually occurs, then it behaves as the corresponding process after the chosen prefix event. The prefix choice is denoted in a format like $a \rightarrow P \mid b \rightarrow Q$ which separates all the candidate choices. For example, a vending machine that serves either one chocolate or one toffee before it breaks can be described as $choc \rightarrow STOP \mid toffee \rightarrow STOP$.
- **Process internal choice:** $P \sqcap Q$ denotes a process that behaves either like P or Q , where the selection between them is arbitrary, uninfluenced by the external environment. It is also named the *nondeterministic choice*. For example, in a money-changing machine (MCM) which always gives the right change in one of two combinations $MCM = in? 1pound \rightarrow ((out! 50p \rightarrow out! 50p \rightarrow MCM) \sqcap (out! 20p \rightarrow out! 20p \rightarrow out! 20p \rightarrow out! 20p \rightarrow out! 20p \rightarrow MCM))$, its external user has no influence over which combination she or he gets.
- **Indexed internal choice** $\bigsqcap_{i \in J} P_i$ is a process which can behave as any one of the P_i , where J is a non-empty set of indices and process P_i is defined for each $i \in J$.

Examples will be given in the case study.

- *Process external choice*: $P \sqcap Q$ denotes a process that behaves either like P or Q , where the selection between them is chosen by the environment. The choice is resolved by the performance of the very first event of either P or Q , in favour of the process that performs it. For example, if the initial event of P is a , and the initial event of Q is b , and a is different from b , that is, if $P = a \rightarrow P'$ and $Q = b \rightarrow Q'$ and $a \neq b$, then the external choice operator \sqcap is the same as the event prefix choice operator: $P \sqcap Q = (a \rightarrow P') \sqcap (b \rightarrow Q') = (a \rightarrow P') \mid (b \rightarrow Q')$.
- *Parallel Composition*: when two processes P and Q are executed concurrently, each process may execute independently according to its prescribed patterns of behaviour. If P and Q share a *synchronised* event, then the range of possible behaviour of P or Q will be influenced by the synchronisation. We describe the combined behaviour of P and Q as parallel composition, denoted $P \parallel Q$.
- *Event Hiding*: the event hiding operator \backslash applied to P denoted $P \backslash c$ is a process which behaves like P but with all communications on channel c concealed; its alphabet is $\alpha P \setminus \{c\}$.
- *Process Recursion*: if F is a continuous function from processes to processes, then $\mu X : A.F(X)$ is the process X with alphabet A satisfying $X = F(X)$. For example, a simple vending machine which serves as many chocolates as required $VMS = (coin \rightarrow (choc \rightarrow VMS))$ can be equivalently described by a recursive equation $VMS = \mu X : \{coin, choc\}.(coin \rightarrow (choc \rightarrow X))$.

4.3.2 Modelling a Domain as a CSP Process

Having introduced the basic elements of the CSP, we can now find similarities (in fact, a close match) between Jackson's notion of a domain in PF and the notion of a process in CSP: they are both self-contained entities that interact with other domains (processes) through shared phenomena (alphabet). At this point, the formalisation is quite straightforward: a domain D in PF is a process D , with its set of shared phenomena as the alphabet αD . Individually, a single shared phenomenon (including an instance of shared event, state or role) of domain D is formalised as a single external event ev of process D . Note this does not prevent D having "internal" phenomena, only that such phenomena should be hidden from its environment through event hiding.

4.3.3 The (Stable) Failures Model in CSP

CSP is a very rich language, for which many theories and models have been developed, such as the *traces* model, the *failures* model, the *failures/divergences/infinite traces* model, etc [137, 146]. For the purpose of formalising problem diagrams and interpreting problem progression, we need to choose a suitable CSP model that has the closest match.

Justification for Choosing the (Stable) Failures Model

Our motivation behind formalising a problem diagram is to reason about transforming requirements and domain descriptions in a rigorous manner. In PF, Jackson gives two important aspects of a domain property and requirement that must be captured and addressed in the reasoning: *safety* is "a domain property or requirement that some specified event or state change will definitely not happen"; *liveness* is "a domain property or requirement that some specified event or state change will definitely happen" [83].

Therefore, a formal description of a problem (diagram) should include both safety properties and liveness properties.

The stable failures model in CSP is widely regarded as being able to model both safety properties and liveness properties, while the traces model only captures the safety properties [146].

Although the failures/divergences/infinite traces model takes into account a more complex situation where a process may have a divergent behaviour (the process performs internal transitions forever, never reaching a stable state nor performing any event), nothing can be guaranteed of the behaviour of such a process [146]. After comparing and reviewing many CSP models, Schneider [146] concludes that “the stable failures model for CSP [137] is a relatively recent development [...], the insight behind the stable failures model is that divergence can often usefully be ignored” (on page 259). We do not choose a model that contains divergent behaviours in our formal approach to problem progression because in PF divergent behaviours of a domain raise standard problem concerns that are analysed and addressed informally. For our purpose of progression, formal reasoning has to make the assumption that these divergent behaviours have been addressed. We claim that our formal approach to problem progression addresses the main part of the problem rather than formalises every aspect of the informal world. Based on the above reasons, we do not choose the failures/divergences/infinite traces model.

Traces - Basic Concepts, Definitions and Notations

Since the (stable) failures model involves both traces and refusals, let us have a brief look at traces first:

A basic way of describing a process is through the description of its traces. A *trace*

of a process is a finite sequence of *symbols* recording the events in which the process has engaged up to some moment in time. The relative order of the occurrences of these events is also recorded. For example, a trace is denoted as a sequence of symbols, separated by commas and enclosed in angle brackets: $\langle a_1, \dots, a_n \rangle$ is the trace consisting of an ordered sequence of event symbols a_1, \dots, a_n . The trace that has no event involved is called an *empty trace*, denoted $\langle \rangle$. The empty trace is the shortest possible trace of every process.

The complete set of all possible traces of a process P is a function of P denoted as $traces(P)$ [68].

The following are some basic operations on finite traces that we will use later in this chapter (adapted from [68] and [146]):

- *Catenation* is an operation that constructs a trace by putting two traces s and t together by writing s first and then connecting the beginning of t to the end of s . It is denoted as $s \frown t$, e.g., $\langle coin, choc \rangle \frown \langle coin, toffee \rangle = \langle coin, choc, coin, toffee \rangle$;
- *Restriction* is an operation that constructs a trace from a given trace t by omitting all symbols outside a given set A . It is denoted as $t \upharpoonright A$, for example, $\langle coin, choc, coin, toffee, coin, choc \rangle \upharpoonright \{choc, toffee\} = \langle choc, toffee, choc \rangle$;
- *Head* is an operation that allows to get the first symbol of a non-empty trace, denoted as $head(tr)$. *Tail* is an operation that allows us to construct a trace by getting the result of removing the head of a non-empty trace, denoted as $tail(tr)$. For example, $head(\langle coin, choc, coin \rangle) = coin$, $tail(\langle coin, choc, coin \rangle) = \langle choc, coin \rangle$; these operations on an empty trace are undefined;
- *Length* is the number of symbols in a trace. It is denoted $|tr|$ for a trace tr , for

example, $|\langle a, b, a \rangle| = 3$;

- *Prefix*: $ur \leq tr$ means ur is a prefix of tr , for example, $\langle a, b \rangle \leq \langle a, b, c \rangle$; a more general form of trace prefix can be written as $ur \leq^n tr$ which means ur is a prefix of tr at most n symbols shorter, that is, $ur \leq tr \wedge |tr| - |ur| \leq n$, for example, $\langle a, b \rangle \leq^2 \langle a, b, c, d \rangle$, and $\langle a, b \rangle \leq^2 \langle a, b, c \rangle$.

Stable Failures - Basic Concepts and Definitions

A process P is guaranteed to respond to an offer of an event ev if that event can be performed from P , provided that there are no internal transitions from P that keep P fully occupied, thus preventing P from engaging in event ev . In other words, a process P which can make no internal progress is said to be *stable*, denoted as $P \downarrow$. Guarantees are concerned with stable states. A stable process P can always respond in some way to the offer of a set of events X by its environment if there is at least one event $a \in X$ that P can engage in. If there is no such event $a \in X$, then P *refuses* the entire offer set X [146].

The CSP approach to the semantics of a *refusal* is to associate a process with its traces, and then to use this information to understand the behaviour of the process as a whole. Suppose that we carry out an experiment on the process P in an environment that offers the set X of events, and we wait as long as necessary to see if any events in X are performed. If no events are performed, then set X is considered a *stable refusal* of process P [146].

According to [146], at some point during an execution of process P , an offer set X of events will be refused by P . This refusal will be recorded with the finite traces of events tr which were performed during the execution leading up to the refusal of X .

The pair (tr, X) (usually written as (tr, ref)) is said to be a *stable failure* of P .

A Predicative Semantics of the (Stable) Failures Model

In this thesis, we adopt Lai and Sanders' "predicative" semantics [101] of CSP syntax. Their work, which originates in [100] (it has become part of the unifying theory of programming [70]), gives a predicative version of CSP's failures model, which defines some basic concepts and their components in the model using predicates on traces tr and refusals ref :

In the predicative failures model, a *specification* is a predicate with free variables tr (traces) and ref (refusals).

In the predicative failures models, a *process* is a specification that satisfies the following four conditions:

$$P1. P(<>, \{\})$$

$$P2. P(tr \frown ur, \{\}) \Rightarrow P(tr, \{\})$$

$$P3. Y \subseteq X \wedge P(tr, X) \Rightarrow P(tr, Y)$$

$$P4. P(tr, X) \wedge \neg \exists v : val(c) \bullet P(tr \frown \langle c.v \rangle, \{\}) \Rightarrow P(tr, X \cup \{c\})$$

Recall that a CSP process has been informally defined as an "independent and self-contained" entity or object with a particular set of events, through which it interacts with its environment [67, 146]. We observe that the above four conditions give a formal meaning to the "independent and self-contained" properties that a valid process must have.

P1 defines that a process can refuse nothing before it starts to execute;

P2 defines the trace integrity of a process: if a sequence of events has happened or has been recorded, e.g., $tr \frown ur$, then some early part of the sequence of events, e.g., tr must have happened. P2 is called *prefix closure*;

P3 defines the failure integrity of a process: if a process P can refuse a set of events X after engaging a sequence of events tr , then it can certainly refuse a subset $Y \subseteq X$ events after the same trace. P3 is called *subset closure*;

P4 defines the relationship between refusals and events that are not possible: if no event from the value set of channel c can follow the trace tr , then the value set can be added to the refusal set. Events are either possible or can be refused [146].

The following defines a predicative failures semantics of various components of an arbitrary process P in terms of trace tr and refusal ref (adapted selectively from [101]):

- Process $STOP_A$ with alphabet A refuses to engage in any communication in A , that is, the simplest process

$$STOP_A(tr, ref) \stackrel{\wedge}{=} (tr = \langle \rangle) \wedge (ref \subseteq A).$$

- Process $CHAOS_A$ is modelled by arbitrary behaviour, that is, the weakest process

$$CHAOS_A(tr, ref) \stackrel{\wedge}{=} true.$$

- $c!e \rightarrow P$ is a process whose alphabet equals that of P , which contains c ; it outputs a value e on channel c and then behaves like process P

$$\begin{aligned} (c!e \rightarrow P)(tr, ref) &\stackrel{\wedge}{=} (c \notin ref) \triangleleft tr = \langle \rangle \triangleright (head(tr) = \langle c.e \rangle) \\ &\wedge P[tail(tr)/tr]. \end{aligned}$$

The above defines that the very first event that process $(c!e \rightarrow P)$ engages in has to be its output event $c!e$ (when it starts, i.e., $tr = \langle \rangle$, it cannot refuse communi-

cation on channel c), then afterwards, i.e., when $tr \neq \langle \rangle$, the head of its trace is $\langle c.e \rangle$ and the tail of its trace is exactly like that of process P .

- $c?x \rightarrow P$ is a process whose alphabet equals that of P , which contains c ; it inputs a value on channel c , stores it as variable x , and then behaves like process P

$$(c?x \rightarrow P)(tr, ref) \stackrel{\Delta}{=} (c \notin ref) \triangleleft tr = \langle \rangle \triangleright \exists v : val(c) \bullet \\ (head(tr) = \langle c.v \rangle \wedge P[tail(tr)/tr, v/x]).$$

The above defines that the very first event that process $(c?x \rightarrow P)$ engages in has to be its input event $c?x$ (when it starts, i.e., $tr = \langle \rangle$, it cannot refuse communication on channel c), then afterwards, i.e., when $tr \neq \langle \rangle$, there exists a value v on channel c such that the head of its trace is $\langle c.v \rangle$ and the tail of its trace is exactly like that of process P by replacing x with v .

- The nondeterministic choice $P \sqcap Q$ between P and Q is a process that behaves like either P or Q , but the choice is internal, uninfluenced by the environment

$$(P \sqcap Q)(tr, ref) \stackrel{\Delta}{=} P(tr, ref) \vee Q(tr, ref).$$

In our work, a process that is composed using the internal choice operator is usually implemented/programmed using conditional instructions (e.g., “if ... then ... else”) in a programming language, see the FDR script in our case study.

- The deterministic choice $P \sqsubseteq Q$ between P and Q is a process that behaves like either P or Q , but the choice is determined by the environment on the first interaction

$$(P \sqsubseteq Q)(tr, ref) \stackrel{\Delta}{=} (P(tr, ref) \wedge Q(tr, ref)) \triangleleft tr = \langle \rangle \triangleright (P(tr, ref) \vee \\ Q(tr, ref)).$$

- For processes P and Q , their communication interface is defined to be $\alpha(P) \cap \alpha(Q)$. The parallel composition $P \parallel Q$ of P and Q is a process whose alphabet is the union of those of P and Q ; it behaves like P and Q evolving in parallel, with all communications on their communication interface synchronised

$$(P \parallel Q)(tr, ref) \stackrel{\Delta}{=} \exists X \subseteq \alpha P, Y \subseteq \alpha Q \bullet [(ref = X \cup Y) \wedge P(tr \upharpoonright \alpha P, X) \wedge Q(tr \upharpoonright \alpha Q, Y)]^4.$$

The above defines that if P is able to refuse some events X in its interface αP , then so is the combination; if Q is able to refuse some events Y in its interface αQ , then so is the combination; if synchronisation is required for the performance of events, then either component is independently capable of blocking them [146].

- $P \setminus c$ is a process that behaves like P but with all communications on channel c concealed; its alphabet equals $\alpha P \setminus \{c\}$. The failure semantics of $P \setminus c$ has a more complex definition [101], which is not used in this thesis, thus omitted.
- recursion: if F is a continuous function from processes to processes, then $\mu X : A.F(X)$ is the process X with alphabet A satisfying $X = F(X)$. The failure semantics of $\mu X : A.F(X)$ given by Lai and Sanders [101] is not used in this thesis, thus omitted.

There are other process combinators, some of which can be found in [67, 18]. Since they are not used in this thesis, we omit them for reasons of conciseness. The behaviour of an arbitrary process P is one of the combinations of the above components [101]:

⁴ Note that in order to avoid confusion with other brackets like “(” and “)”, we use “[” and “]” to indicate the scope of the existential quantifiers.

$$\begin{aligned}
P(tr, ref) ::= & \\
& (STOP_A \mid CHAOS_A \mid (c!e \rightarrow P) \mid (c?x \rightarrow P) \mid (P \sqcap Q) \mid (P \sqbox Q) \mid (P \parallel Q) \\
& \mid (P \setminus c) \mid \mu X : A.F(X))(tr, ref)
\end{aligned}$$

4.3.4 Modelling a Requirement and \vdash_{DRDL} in the Predicative Failures Model

In PF, a *requirement* is defined to be some constraint on or reference to some physical phenomena in the problem context. Unlike a *domain* which is defined to be an independent and self-contained entity modelled by a process, a requirement is generally described by a predicate that can be either satisfied when it evaluates *true*, or not satisfied when it evaluates *false*. By modelling a requirement in PF as a *specification* in the predicative failures model, we can find a close match between the truth value of a predicate and the satisfaction or dissatisfaction of a requirement.

Recall that in the predicative failures model, a CSP *specification* is defined to be a predicate with free variables *tr* (traces) and *ref* (refusals). The set of all specifications is denoted *Spec*. The set *Spec* is defined to be an ordered set (an ordered set is a set that contains a binary relation for expressing the order that is reflexive, anti-symmetric and transitive, for details and examples refer to [147]). Within the ordered set *Spec* of specifications, there is the following equivalence relationship between the meaning of *satisfaction* (usually denoted **sat**) and logical implication between predicates [101]:

Under Lai and Sanders' predicative failures model, a specification *Sp* is said to *satisfy* specification *Sq*, i.e., *Sp sat Sq* if and only if $Sp \Rightarrow Sq$ [101]. If we regard the solution set in Hall *et al.*'s semantics as a subset of the ordered set *Spec*, then the entailment \vdash_{DRDL} relation can be interpreted as satisfaction **sat** in the predicative failures model.

There is a single complication; more details will be given in a later section.

4.3.5 Modelling the Sharing of Phenomena as Parallel Composition

The notion of parallel composition in CSP was introduced to investigate the behaviour of a complete system composed of subsystems that act and interact with each other as they evolve concurrently. For example, when we analyse the combined behaviour of two processes put together, their interactions (if they exist) can be regarded as events that require simultaneous participation of both processes involved. Hoare [68] argues that we can assume that the alphabets of the two processes are the same when analysing their overall behaviour. He uses the notation $P \parallel Q$ to denote the process that behaves like the composition of processes P and Q interacting in lock-step synchronisation. He gives an example where a chocolate can be extracted from a vending machine VM only when its customer $CUST$ wants it and only when the vending machine is ready to serve. When thinking about this particular interaction, we can describe the combined process as $VM \parallel CUST$.

Although some other styles of parallel composition operators have been introduced since Hoare's work, such as alphabetised parallel, interleaving, generalised parallel, Roscoe [137] points out that the main difference between Hoare's text on parallel composition and others is the treatment of alphabet. Hoare's treatment makes the operator more elegant while other versions have explicit alphabets thus more complex. He concludes that the choice of one version over the other is a matter of taste, and this difference is not regarded as an important issue since everything done in one version can be done in the other with trivial changes.

In PF, the interactions between two connecting domains have similar characterisations: the phenomenon they share is considered instantaneous, and both domains are simultaneously engaged in the same phenomenon [83]. From the CSP point of view,

parallel composition is equivalent to conjunction (i.e., logical “and”) when we use CSP as a specification language (rather than an implementation language) [137]. The view of this thesis that parallel composition is essentially conjunction with channel phenomena shared is in agreement with Zave and Jackson’s observation - “Conjunction as Composition” [170].

More details of the modelling will be given in a later section.

4.3.6 Distinguishing “Control” and “Observe” in CSP Descriptions

In PF, the notion of “control” and “observe” plays an important part in problem descriptions. From a domain’s description, we should be able to distinguish those visible phenomena that are controlled by the domain from those that are observed by it; this amounts to the property [169] that only a domain that controls a phenomenon should be able to change it. As Zave and Jackson [169] point out, full CSP [68] does not have the syntax to explicitly distinguish between control of a shared communication and observation of it, so we must impose it. We need to restrict domain models to those CSP processes for which “control” and “observe” make sense:

For any CSP process P with alphabet αP , we define

- (a). $P! \triangleq \{d \mid (d!v \in \alpha P) \vee (P = \text{CHAOS}_{\alpha P} \wedge d!v \in \alpha P)\}$, i.e., those channels controlled by P ;
- (b). $P? \triangleq \{d \mid (d?x \in \alpha P) \vee (P = \text{CHAOS}_{\alpha P} \wedge d?x \in \alpha P)\}$, i.e., those channels observed by P .

To be able to distinguish “control” from “observe”, we must consider only processes such that $P! \cap P? = \{\}$ holds. Appendix A contains a characterisation of processes for which this condition holds.

4.3.7 Achieving a Complete Interpretation of Hall et al.'s PF Semantics in CSP

Let us revisit Figure 4.2 (recalled here as Figure 4.3):

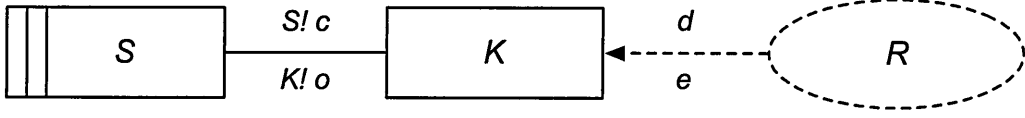


Fig. 4.3: Interpreting problem frame semantics using CSP

Recall that a domain's behaviour in PF can be formalised as a CSP process: the machine domain S in Figure 4.3 can be formalised as a process S , and the context K can be formalised as a process K (we can model n number of application domains D_1, D_2, \dots, D_n as a single combined process $K = D_1 \parallel D_2 \parallel \dots \parallel D_n$) with their sharing of phenomena as parallel composition ($S \parallel K$). Since the requirement R is a constraint on or reference to domain K 's property or phenomenon, we can formalise it as a predicate on the context K , i.e., a CSP specification R . Also recall that the entailment relation \vdash_{DRDL} in Hall *et al.*'s semantics can be interpreted as **sat**. We note that in the *POS* example, the requirement does not mention $present(notice)$, $present(payment)$ or $present(change)$. This presents us with a problem in our CSP modelling (the complication referred to earlier) as these events must be mapped to the silent action or else be captured by the *REQ* statement. To this end we must alter the semantics slightly so that

$$(K \parallel S) \setminus [(o \cup c) \setminus (d \cup e)] \text{ sat } R.$$

The control-and-observe relationships about domain S can be formalised as the following two equations based on the definitions given in the previous section:

- $S! = c$, meaning “domain S controls its shared phenomena c ”;

- $S? = o$, meaning “domain S observes its shared phenomena o ”.

Now we can interpret Hall *et al.*’s semantic challenge

$$c, o : [K, R] = \{S : \textit{Specification} \mid S \text{ controls } c \wedge S \text{ observes } o \wedge K, S \vdash_{\textit{DRDL}} R\}$$

as a challenge in CSP

$$c, o : [K, R] = \{S : \textit{Specification} \mid S! = c \wedge S? = o \wedge (K \parallel S) \setminus [(c \cup o) \setminus (d \cup e)] \textbf{sat} R\}.$$

When $K = D_1 \parallel D_2 \parallel D_3, \dots \parallel D_n$ for CSP processes D_1, D_2, \dots, D_n ,

$$\begin{aligned} c, o : [D_1 \parallel D_2 \parallel \dots \parallel D_n, R] \\ = \{S : \textit{Specification} \mid S! = c \wedge S? = o \wedge (D_1 \parallel D_2 \parallel \dots \parallel D_n \parallel S) \\ \setminus [(c \cup o) \setminus (d \cup e)] \textbf{sat} R\}. \end{aligned}$$

Note the parallel composition operator in the above formula is valid for all complex topologies/structures of connecting domains, though details of the operator and the calculated result may be more complex.

4.4 Solving the Challenge Using Lai’s Quotient

We consider the case where $d \cup e = c \cup o$ first, so that $(K \parallel S) \setminus [(c \cup o) \setminus (d \cup e)] = K \parallel S$. In order to meet the challenge of finding an S such that $K \parallel S \textbf{sat} R$, we need a new operator that can perform the opposite calculation of parallel composition. Let us look at what is available in CSP literature:

According to Chen and Sanders [28], the concept of “weakest calculation” in com-

puting owes its origins to Dijkstra's *weakest preconditions* [42]. Later, Hoare and He [69] define the weakest prespecification and postspecification to provide a “weak inverse” for sequential composition. The meaning of a weak inverse operator can be explained in the following simple example:

In algebra the operator “ $-$ ” is called the *inverse* of the “ $+$ ” operator, because if $X + A = B$, then we can calculate unknown value of integer X from given values of integers A and B , that is $X = B - A$; we can apply operator “ $-$ ” to any known integers, and the result is always an integer.

However, as Chen and Sanders [28] point out, not every operation of a given type has an inverse. For example, integer multiplication does not have an inverse: for an unknown integer X and given integers A and B , if $X \times A = B$, then X can be calculated by $X = B \div A$; however, we cannot always get an integer if we apply operator “ \div ” to any two integers (sometimes we get decimal fractions). Therefore, for the given type of integer calculation, operator “ \div ” is called a *weak inverse* of the operator “ \times ” rather than an exact inverse of “ \times ” [28].

Lai and Sanders [101] extend Hoare and He's notion of “weak inverse” of sequential composition to parallel composition and they have given the *weakest environment* calculus to provide the weakest process X that placed in parallel with an established subcomponent P satisfies their overall specification R :

$$X \parallel P \text{ sat } R \Leftrightarrow X \text{ sat } P \parallel R$$

$P \parallel R$ is called the weakest environment of a process. Lai and Sanders [101] provide a closed predicate definition for the weakest environment: given specifications P , R and a chosen set $A \subseteq \alpha P$, the weakest environment of P in R , denoted $P \parallel R$ with alphabet

$\alpha R \setminus \alpha P \cup A$ as the specification:

$$\begin{aligned}
 P \parallel R(tr, ref) &\triangleq \forall ur : traces(R) \forall rep \subseteq \alpha P \\
 &\bullet [tr = ur \upharpoonright \alpha(P \parallel R) \\
 &\quad \wedge P(ur \upharpoonright \alpha P, rep) \\
 &\quad \Rightarrow R(ur, rep \cup ref)]
 \end{aligned}$$

Figure 4.4 illustrates the role of Lai's quotient in problem progression.

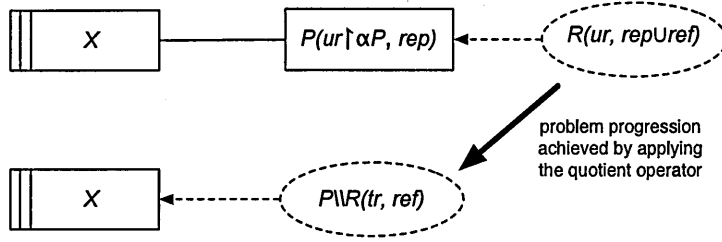


Fig. 4.4: A generic problem diagram, (adapted from [84]) illustrating Lai's quotient)

An informal explanation of the above formula is: in a CSP failures model, given that a composed system must satisfy R (a process expressed by a predicate on variables ur and $rep \cup ref$), if one of the subsystem can be expressed as a given process P , then the weakest environment of P - the remaining subsystem to be specified $P \parallel R$ can be calculated constructively by the following two predicates: $P \parallel R$'s trace is tr and its refusal is ref ; for all the traces of R - ur and for all the refusals of P - rep , such that $P \parallel R$'s trace is the overall system's trace restricted to the remaining subsystem $P \parallel R$'s alphabet, and if the predicate $P(ur \upharpoonright \alpha P, rep)$ on process P 's trace and refusals holds, then the predicate $R(ur, rep \cup ref)$ on the overall system's traces and refusals must hold.

For us, the importance of Lai's quotient is that it provides a (in some sense) canonical solution to a challenge, at least when domains are described in the CSP family

of notations. Now Hall *et al.*'s semantic challenge (at least in the simple case when $c \cup o = d \cup e$) becomes:

$$c, o : [K, R] = \{S : \text{Specification} \mid S! = c \wedge S? = o \wedge S \text{ sat } K \parallel R\}.$$

4.4.1 Interpreting Problem Progression as Stepwise Applications of Lai's Quotient

From Figure 4.4 we can see that by applying the quotient operator we achieve the effect of removing domain P and re-expressing requirements R into a new statement $P \parallel R$ which specifies domain X 's behaviour. Therefore, if we can formalise a problem diagram using CSP, then one problem progression step can be interpreted as one step of applying the quotient operator.

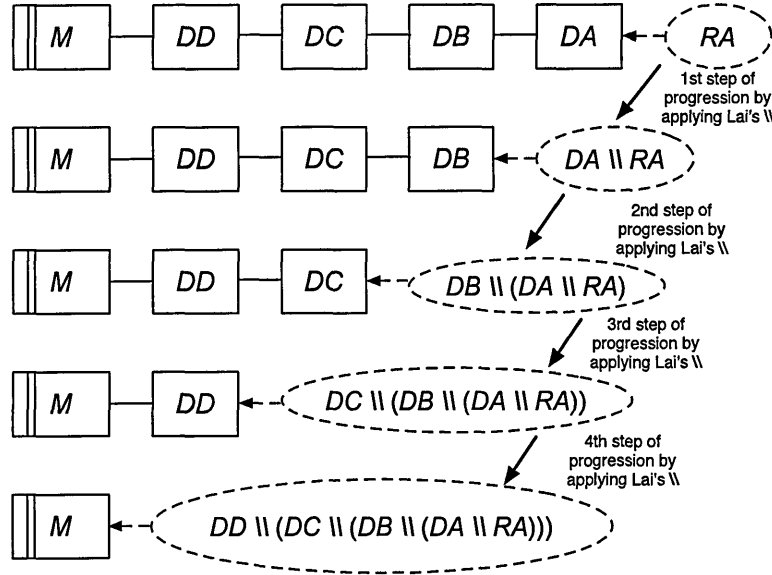


Fig. 4.5: A progression of problems (adapted from [84]), interpreting problem progression as stepwise applications of Lai's quotient.

For a complex problem diagram which may have many domains, problem progression can be regarded as stepwise applications of the quotient operator until the

re-expressed requirement constrains or refers to only the machine's behaviour, as illustrated in Figure 4.5.

Note that when we apply the first step of progression using Lai's quotient, we regard the combined process $M \parallel DD \parallel DC \parallel DB$ as the unknown process to be found (like X in Figure 4.4). This is in agreement with the view in PF that the solution domain is treated in the same way as an application domain [83]. We can apply similar techniques until only the machine domain M is left, which indicates that the problem progression is completed.

4.5 Case Study - Solving the POS Example Problem

Based on our techniques in providing the general solution, we are now ready to solve the example POS problem that we have introduced in the beginning of this chapter.

4.5.1 Formalising the Domain and Requirement

Note that when applying our formal techniques to the example problem, we need to describe it using both predicate expressions and process expressions in CSP. We need predicates to be able to apply the definition of Lai's quotient operator to construct the solution specification; we need process expressions to communicate intuitions about relative orderings of occurrences of events and associated values communicated, and for validating the derived specification against requirements using FDR, which has direct support for process expressions in CSP.

The following are the informal domain and requirement descriptions and their formalisation (with justifications):

The Customer Domain CUST:

Informally, a customer is a person who wants to buy an item from the shop. First of all, he presents the item he wants (whose price is i pence, with i a number between 1 and 100) to the self-checkout POS system (e.g., through the bar code scanner). Then, after receiving a notice n from the system (e.g., via a screen display showing the payment needed), he presents, perhaps, part payment in cash p pence, a coin of value 1p, 2p, 5p or 10p to the system (e.g., through a cash acceptor). If the presented payment is sufficient, i.e., $i \leq p$, then the customer will be given the change c (e.g., via the dispenser handler), followed by a receipt for $r = i$ as a proof of purchase (e.g., a printout from the receipt printer); if the presented payment is insufficient, i.e., $p < i$, then further notices displaying the remaining amount of payment are issued to the customer until sufficient payment is presented, after which the customer will be given the change and a receipt. Note that i, n, p, c, r are assumed to be in natural numbers, i.e., $i, n, p, c, r \in \mathbb{N}$. In this example, we assume that the above payment method is in cash for a single item, and that i, n, p, c, r are expressed in pence in British money.

In this example, for brevity of presentation, we use *item*, *notice*, *pay*, *change*, and *receipt* as a short form of events $\text{present}(\text{item})$, $\text{present}(\text{notice})$, $\text{present}(\text{payment})$, $\text{present}(\text{change})$, and $\text{present}(\text{receipt})$ in Figure 4.1, respectively.

From the descriptions above, we model the behaviour of a customer using the following formula:

$$\begin{aligned} CUST &= \prod_{i \in \{1, \dots, 100\}} \text{item!}i \rightarrow \text{notice?}i \rightarrow PAY, \text{ where} \\ PAY &= \prod_{p \in \{1, 2, 5, 10\}} \text{pay!}p \rightarrow (\text{change?}c \rightarrow \text{receipt?}i \rightarrow STOP_{\alpha CUST} \\ &\quad \square \text{notice?}n \rightarrow PAY). \end{aligned}$$

In the above formula, *item*, *notice*, *pay*, *change*, *receipt* denote the names of communication channels of process *CUST*, all of which are synchronised with its envi-

ronment process *POS*. Within this context, i, n, p, c, r denote the values being passed through these channels. The symbol $!$ means the value is output by process *CUST* onto its communication channel, and $?$ means a certain value is received by process *CUST* from its communication channel. For brevity, we sometimes refer to an event by its channel name only, when unambiguous.

Eventually process *CUST* ends with $STOP_{\alpha CUST}$, where

$$\alpha CUST = \{item, notice, pay, change, receipt\},$$

which indicates that his engagement in the above events is terminated.

The justifications for the above formalisation are:

- The customer is a biddable domain in PF, whose behaviour is modelled through the indexed internal choice operator⁵ $\prod_{i \in \{1, \dots, 100\}}$, where the value of the item i is assumed to range from 1 to 100 *pence*. The value of the item is determined by the customer's choice. Similarly, *PAY* is also modelled through the indexed choice operator $\prod_{p \in \{1, 2, 5, 10\}}$, where the amount of payment p is assumed to be any of 1, 2, 5 or 10 *pence*, whose choice is determined by the customer.
- Only sensible behaviours of the customer shared with *POS* should be formalised. This is consistent with PF that non-sensible commands or events are often ignored [83]. For instance, some random behaviours of the *CUST*, such as presenting a payment without any item, should be ignored/refused by *POS*. Therefore, *CUST* should start with event $item!i$, which means any other events such as $pay!p$, $notice?n$, $change?c$ or $receipt?r$ should be in *CUST*'s refusal set;
- In this particular example, the value communicated in the first *notice* event is i ; while the values communicated in other *notice* events keep changing, thus repre-

⁵ In this thesis, biddable behaviour is modelled by internal choice.

sented by variable n ; similarly, the values communicated in the *pay* event and the *change* event keep changing, thus represented by variables p and c , respectively; the value communicated in the *receipt* event is always i - a constant;

- After presenting the item *item!* i , and receiving a notice about it *notice?* i , the customer engages in the *PAY* process;
- Whether to pay more or leave the shop with the change and receipt is not the decision of the customer *CUST*, but of the *POS*. Therefore, after presenting the payment *pay!* p , *CUST*'s behaviour could be either:
 - receiving the due change *change?* c , followed by the receipt *receipt?* i . Then the customer's involvement with *POS* stops, resulting in the customer leaving the shop with the purchased items and receipt (this is the situation when $i \leq p$); or
 - receiving a notice *notice?* n about further payment is needed, which prompts the customer back to the beginning of the *PAY* process (this is the situation when $p < i$).

In process *PAY*, external choice operator \square is used between the two processes after event *pay!* p because the above choice is determined externally by *POS*.

The Requirement REQ:

The requirement could be informally described as: “customers should pay for and receive a receipt for the correct amount on presentation of items to the *POS* system”.

From the above statement, the requirement *REQ* only constrains two events: whenever event *item.i* happens, eventually event *receipt.r* should happen, and the value of r should be equal to that of i , i.e., $r = i$. Therefore,

$$REQ = \prod_{i \in \{1, \dots, 100\}} item.i \rightarrow receipt.i \rightarrow STOP_{\{item, receipt\}}.$$

Note we use $item.i$ and $receipt.i$ to represent that both $CUST$ and POS participate in this event. In other words, from $CUST$'s perspective, the event should be denoted as $item!i$, and from POS 's perspective, the same event should be denoted as $item?i$, therefore expression $receipt.i$ includes both perspectives of $CUST$ and POS . This gives us the intuition that if an $item.i$ and $item?i$ are given/exist, we are sure that $item!i$ can be derived/must exist.

The above process expression is not detailed enough for us to construct POS because it does not prescribe all of the interaction behaviours between $CUST$ and POS , i. e., events $notice$, pay and $change$ do not appear in REQ 's alphabet. For instance, according to the CSP semantics of a problem diagram introduced previously, for problem diagram in Figure 4.1 we need to find a process POS such that

$$(POS \parallel CUST) \setminus [\{item, notice, pay, change, receipt\} \setminus \{item, receipt\}] \text{ sat } REQ,$$

and the solution set for the problem diagram is:

$$\begin{aligned} & \{notice, change, receipt\}, \{item, pay\} : [CUST, REQ] \\ = & \{POS : Specification \mid POS! = \{notice, change, receipt\} \wedge POS? = \{item, pay\} \\ & \wedge (POS \parallel CUST) \setminus \{notice, pay, change\} \text{ sat } REQ\}. \end{aligned}$$

Notice that the problem is to find a POS to satisfy the above formula. However, Lai's quotient can not directly allow us to calculate POS . As do Lai and Sanders [101], we therefore introduce the above missing events into a more detailed requirement statement which we call $REQC$.

We construct $REQC$ in a way that relates to $CUST$'s behaviour, meanwhile still satisfying REQ after hiding events $notice$, pay and $change$, as follows:

REQC relates to *CUST* in the following way:

- *REQC* corresponds to *CUST*: *REQC*'s *item.i* maps to *CUST*'s *item!i*; *REQC*'s *notice.i* maps to *CUST*'s *notice?i*; *REQC*'s component *REQCPAY* maps to *CUST*'s component *PAY*; both of them share the same event sequence and binding on value *i*;
- *REQCPAY* corresponds to *PAY*: *REQCPAY*'s *pay.p* maps to *PAY*'s *pay!p*; *REQCPAY*'s internal choice operator \sqcap corresponds to *CUST*'s external choice operator \square - the difference is because the choice on whether to perform *change* or *notice* is made by the *POS*, which is external to *PAY* but internal to *REQCPAY*; *REQCPAY*'s *change.c* maps to *PAY*'s *change?c*; *REQCPAY*'s *receipt.i* maps to *PAY*'s *receipt?i*; *REQCPAY*'s $STOP_{\alpha REQC}$ maps to *PAY*'s $STOP_{\alpha CUST}$; *REQCPAY*'s *notice.n* maps to *PAY*'s *notice?n*; both of them share the same event sequence and binding on value *p*.

Based on the above correspondence, we begin by constructing an abstract $REQC_A$, from which *REQC* will be derived, as follows:

$$REQC_A = \prod_{i \in \{1, \dots, 100\}} item.i \rightarrow notice.i \rightarrow REQCPAY_A, \text{ where}$$

$$REQCPAY_A = \prod_{p \in \{1, 2, 5, 10\}} pay!p \rightarrow (change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST} \sqcap notice? remain \rightarrow REQCPAY_A).$$

To determine the value of *remain*, and to resolve the internal choice, we will introduce conditional expression “if ... then ... else”, to give the concrete *REQC*. This means we must define a concrete *REQCPAY* as a function with two parameters $REQCPAY(i, i)$ in the following way:

Assume that the *pay* events lead to *n* coins of values $p_1, p_2, p_3, \dots, p_n$ being ex-

changed.

$$\sum_{x=1}^n p_x$$

is then the total amount exchanged after n payment events.

- The first parameter i is a constant used for passing the item cost i to the *receipt* event;
- The second parameter i is a variable whose initial value is the same as the item cost i , after which its value is substituted by

$$remain = i - \sum_{x=1}^{n-1} p_x$$

which will change as x increases from 1 to $n - 1$ (n is the subscript/index for the last payment, after which no further payment is needed), which is used for passing values to the *notice* event, which keeps displaying updated information on the remaining payment needed;

- Once the payment is sufficient, a value of

$$\sum_{x=1}^n p_x - i$$

will be passed to the *change* event, after which *receipt.i* will be issued to the customer.

Notice that, by combining the above, we get

$$\sum_{x=1}^{n-1} p_x < i \leq \sum_{x=1}^n p_x.$$

From abstract process $REQC_A$, we may now construct a concrete version:

$$\begin{aligned}
 REQC &= \prod_{i \in \{1, \dots, 100\}} item.i \rightarrow notice.i \rightarrow REQCPAY(i, i), \text{ where} \\
 REQCPAY(i, remain) &= \\
 \prod_{p \in \{1, 2, 5, 10\}} pay.p \rightarrow \\
 &\quad \text{if } p < remain \\
 &\quad \text{then } (notice.(remain - p) \rightarrow REQCPAY(i, remain - p)) \\
 &\quad \text{else } (change.(p - remain) \\
 &\quad \rightarrow receipt.i \rightarrow STOP_{\alpha REQC})
 \end{aligned}$$

Applying the hiding operator \setminus to $REQC$, we get

$$\begin{aligned}
 &REQC \setminus \{notice, pay, change\} \\
 &= (\prod_{i \in \{1, \dots, 100\}} item.i \rightarrow notice.i \rightarrow REQCPAY(i, i)) \setminus \{notice, pay, change\} \\
 &= \prod_{i \in \{1, \dots, 100\}} item.i \rightarrow (REQCPAY(i, i) \setminus \{notice, pay, change\}) \\
 &= \prod_{i \in \{1, \dots, 100\}} item.i \rightarrow receipt.i \rightarrow STOP_{\{item, receipt\}} \\
 &\text{sat } REQ.
 \end{aligned}$$

(The validity of the above formula can also be checked by the FDR tool, which we do in a later section.)

Thus, if POS is such that

$$(POS \parallel CUST) \text{ sat } REQ,$$

then

$$(POS \parallel CUST) \setminus \{notice, pay, change\} \text{ sat } REQ \setminus \{notice, pay, change\} \text{ sat } REQ.$$

From the properties of Lai's quotient, any $POS \text{ sat } CUST \parallel REQ$ will solve the problem, though in general Lai's quotient may not always lead to a process [101].

4.5.2 Solving the Problem Using Lai's Quotient

In this particular problem, *CUST* and *POS* synchronise on all their communication channels, namely, *item*, *notice*, *pay*, *change*, *receipt*. Recall that in Lai's definition of the quotient, set A is the alphabet of chosen communication channels between the two sub-processes X and P . In a general case, X and P may have other communication channels that are not shared (i.e., in parallel composition $X \parallel P$, X only needs to synchronise with P via their shared communications, while X 's other communications can be performed independently), thus in this particular example, $CUST \parallel REQC$'s alphabet should be calculated as $(\alpha REQC \setminus \alpha CUST) \cup A$.

We choose the entire alphabet of *CUST* as the set A because it is assumed that all of *CUST*'s alphabet are synchronised communications with *POS*, and is constrained or referred to by *REQC*. In our model, we ignore any other irrelevant behaviours of *CUST* in this formal analysis. Therefore, in this example,

$$\begin{aligned}
 A &= \{item, notice, pay, change, receipt\} \\
 \alpha REQC &= \{item, notice, pay, change, receipt\}, \\
 \alpha CUST &= \{item, notice, pay, change, receipt\}, \\
 \alpha(CUST \parallel REQC) &= (\alpha REQC \setminus \alpha CUST) \cup A \\
 &= \{item, notice, pay, change, receipt\}.
 \end{aligned}$$

We will solve the problem by constructing:

$$POS = (CUST \parallel REQC).$$

The predicate expressions for *CUST* and *REQC*, as needed in Lai's quotient, are derived according to the predicative semantics introduced earlier. For ease of presentation, we express their predicate expressions in the tabular form, as shown below.

Predicates on *CUST*'s *tr* and *accept* (its meaning is given below) expressed in a tabular form:

trace length l	0	1	2	3	4	...	$2n+1$	$2n+2$	$2n+3$
l^{th} element of <i>tr</i>	$\langle \rangle$	$i.i$	$n.i$	$p.p_1$	$n.(i-p_1)$...	$p.p_n$	$c.(\sum_{x=1}^n p_x - i)$	$r.i$
<i>accept</i>	$\{i\}$	$\{n\}$	$\{p\}$	$\{c, n\}$...	$\{p\}$	$\{c, n\}$	$\{r\}$	$\{i\}$

Predicates on *REQC*'s *tr* and *accept* expressed in a tabular form:

trace length l	0	1	2	3	4	...	$2n+1$	$2n+2$	$2n+3$
l^{th} element of <i>tr</i>	$\langle \rangle$	$i.i$	$n.i$	$p.p_1$	$n.(i-p_1)$...	$p.p_n$	$c.(\sum_{x=1}^n p_x - i)$	$r.i$
<i>accept</i>	$\{i\}$	$\{n\}$	$\{p\}$	$\{n\}, \{c\}$...	$\{p\}$	$\{n\}, \{c\}$	$\{r\}$	$\{i\}$

In the above tables, in which, for brevity, we have abbreviated events to their first letters, we show all possible behaviours of *CUST* and *REQC* that are associated with an item that costs i . An item of cost i will lead to a trace of no longer than $2i+3$ events: each time the customer pays, it must be with a coin of value greater than 1 *pence*, so that the amount remaining is at most one less. As i is finite, this ensures all traces of the system are finite.

The first row of the table shows a trace of length l ($0 \leq l \leq 2n+3$). In the second row of the table we give the events of the trace; in the third row, we indicate the refusal set after that trace. We name this set *accept* to represent those entries that the process cannot refuse. For example, in the first table, the entry for $l=3$ is $p.p_1, \{c, n\}$, indicating that the failure is $(\langle i.i, n.i, p.p_1 \rangle, \alpha CUST \setminus \{c, n\})$ (We use *accept* to stand for the intuitive meaning of acceptance, rather than a strictly formal meaning of acceptance, as in [137].).

We can check that the representation of the table interpreted in this way provide the predicative semantics for the represented terms. For example, in *CUST*'s table, from

$$CUST = \prod_{i \in \{1, \dots, 100\}} item!i \rightarrow notice?i \rightarrow PAY, \text{ where}$$

$$PAY = \prod_{p \in \{1, 2, 5, 10\}} pay!p \rightarrow (change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST}$$

$$\square notice?n \rightarrow PAY).$$

we give the following explanations of two representative entries in the table:

- When the trace length is 0, which means $tr = \langle \rangle$, then according to the semantics of event prefix in section 4.3.3, $item.i$ can not be refused, $item.i \notin ref \Leftrightarrow ref \subseteq \alpha CUST \setminus \{item.i\}$, that is, $accept = \{i\}$; also according to the semantics, the next event in tr must be the head of $CUST$ which is $item.i$ whose shorthand is $i.i$ in the table;
- $CUST$'s refusal set after the trace $\langle i.i, n.i, p.p_1 \rangle$ is derived according to the semantics of external choice in section 4.3.3, as follows:

before $(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST} \sqcap notice?n \rightarrow PAY)$ is executed, that is, its trace is empty, its behaviour is defined to be

$$(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha CUST})(tr, ref) \wedge (notice?n \rightarrow PAY)(tr, ref),$$

again, according to the semantics of event prefix, $change.c \notin ref \wedge notice.n \notin ref$ holds, which means $ref \subseteq \alpha CUST \setminus \{change, notice\}$, which explains the entry $accept = \{c, n\}$ (notice the shorthand) in $CUST$'s table.

The rest of the entry can be similarly derived according to the predicative semantics in section 4.3.3.

We also give an explanation for a representative entry in $REQC$'s table:

Different from $CUST$, the choice is internal after the trace $\langle i.i, n.i, p.p_1 \rangle$, i.e.,

$$(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha REQC} \sqcap notice?n \rightarrow REQCPAY)$$

$REQC$'s refusal set after the trace $\langle i.i, n.i, p.p_1 \rangle$ is derived according to the semantics of internal choice in section 4.3.3, as follows:

the above internal choice's behaviour is defined to be

$$(change?c \rightarrow receipt?i \rightarrow STOP_{\alpha REQC})(tr, ref) \vee (notice?n \rightarrow REQCPAY)(tr, ref)$$

according to the semantics of event prefix, $change.c \notin ref \vee notice.n \notin ref$ holds, which means $ref \subseteq \alpha CUST \setminus \{change\}, \{notice\}$, which explains the entry $accept = \{c\}, \{n\}$ in $REQC$'s table. Note that we use “,” to represent “exclusive or”, which means that $REQC$ can refuse either c or n , but not both.

Deriving/Constructing POS's Table Entries Using Lai's Quotient

Lai's quotient is defined as:

$$\begin{aligned}
 CUST \parallel REQC(tr, ref) = \\
 \forall ur : traces(REQC) \forall rep \subseteq \alpha CUST \bullet [tr = ur \upharpoonright \alpha(CUST \parallel REQC) \\
 \wedge CUST(ur \upharpoonright \alpha CUST, rep) \Rightarrow REQC(ur, rep \cup ref)] \\
 \quad (since \alpha REQC = \alpha CUST = \alpha(CUST \parallel REQC), thus tr = ur) \\
 \Leftrightarrow \forall rep \subseteq \alpha CUST \bullet [CUST(tr, rep) \Rightarrow REQC(tr, rep \cup ref)]
 \end{aligned}$$

From the above step of derivation based on Lai's quotient definition, we know that $tr = ur$, which means $POS = CUST \parallel REQC$'s trace tr is always equal to that of $REQC$, due to the fact that $\alpha REQC = \alpha CUST = \alpha(CUST \parallel REQC)$ holds. Therefore, all the entries of trace events in POS 's table is exactly the same as those in $CUST$'s table.

Next, let us look at the *accept* entries in POS 's tables. We derive some representative *accept* entries in POS 's table from the given entries in $CUST$ and $REQC$'s tables.

In the first trace event, given that $CUST(\langle \rangle, \{n, p, c, r\})$ and $REQC(\langle \rangle, \{n, p, c, r\})$ are true (it is a fact, as shown in the tables),

$$\begin{aligned}
 CUST \parallel REQC(\langle \rangle, ref) \\
 = \forall rep \subseteq \{i, n, p, c, r\} \bullet [CUST(\langle \rangle, rep) \Rightarrow REQC(\langle \rangle, rep \cup ref)]
 \end{aligned}$$

That $rep = \{i\}$ contradicts with the fact $CUST(\langle \rangle, \{n, p, c, r\})$ holds. When $rep \subseteq \{n, p, c, r\}$, we know for a fact that the antecedent is always *true*, and in order to make the consequent *true* so that the entire predicate holds, $\{n, p, c, r\} \cup ref = \{n, p, c, r\}$ must hold, therefore we can derive that $ref \subseteq \{n, p, c, r\}$, which means $ref \subseteq \alpha POS \setminus \{i\}$, which allows us to derive the *accept* entry in *POS*'s table as $\{i\}$.

As another example, in the fourth trace event, given that $CUST(\langle i.i, n.i, p.p_1 \rangle, \{i, p, r\})$ is true, and that $REQC(\langle i.i, n.i, p.p_1 \rangle, \{i, p, c, r\} \vee \{i, n, p, r\})$ is true (it is a fact, as shown in the tables),

$$\begin{aligned} & CUST \parallel REQC(\langle i.i, n.i, p.p_1 \rangle, ref) = \\ & \Leftrightarrow \forall rep \subseteq \{i, n, p, c, r\} \bullet [CUST(\langle i.i, n.i, p.p_1 \rangle, rep) \\ & \Rightarrow REQC(\langle i.i, n.i, p.p_1 \rangle, rep \cup ref)] \end{aligned}$$

That $rep = \{c, n\}$ contradicts with the fact $CUST(\langle i.i, n.i, p.p_1 \rangle, \{i, p, r\})$ is true. When $rep \subseteq \{i, p, r\}$, we know for a fact that the antecedent is always *true*, and in order to make the consequent *true* so that the entire predicate holds, either $\{i, p, r\} \cup ref = \{i, n, p, r\}$ or $\{i, p, r\} \cup ref = \{i, c, p, r\}$ must hold (but not both), therefore we can derive that $ref \subseteq \{i, n, p, r\}$ or $ref \subseteq \{i, c, p, r\}$ (but not both), which means $ref \subseteq \alpha POS \setminus \{c\}$ or $ref \subseteq \alpha POS \setminus \{n\}$ (but not both), which allows us to derive the *accept* entry in *POS*'s table as $\{c\}, \{n\}$.

The derivations of the other entries in *POS*'s table are similar.

The constructed table shows *POS*'s behaviour in terms of *tr* and *accept*:

trace length l	0	1	2	3	4	...	$2n+1$	$2n+2$	$2n+3$
l^{th} element of <i>tr</i>	$\langle \rangle$	$i.i$	$n.i$	$p.p_1$	$n.(i-p_1)$...	$p.p_n$	$c.(\sum_{x=1}^n p_x - i)$	$r.i$
<i>accept</i>	$\{i\}$	$\{n\}$	$\{p\}$	$\{n\}, \{c\}$...	$\{p\}$	$\{n\}, \{c\}$	$\{r\}$	$\{ \}$

Note that entries in *POS*'s table correspond to *REQC*'s entries, which leads us to

derive POS 's expression in a process form based on the correspondence, as follows:

$$\begin{aligned}
 POS &= \prod_{i \in \{1, \dots, 100\}} item?i \rightarrow notice!i \rightarrow POSPAY(i, i), \text{ where} \\
 POSPAY(i, remain) &= \\
 &\prod_{p \in \{1, 2, 5, 10\}} pay?p \rightarrow \text{if } p < remain \text{ then } (notice!(remain - p) \\
 &\quad \rightarrow POSPAY(i, remain - p)) \text{ else } (change!(p - remain) \\
 &\quad \rightarrow receipt!i \rightarrow STOP_{\alpha POS})
 \end{aligned}$$

Note that $POSPAY$ involves the communication of at least two values, value i for the first *receipt* event, and a variable value *remain* for later *notice* event representing the remaining amount of payment needed; the choice is chosen by a conditional: if the payment $remain \leq p$, then a change and a receipt will be given out by POS ; if $p < remain$ then a notice for the need of further payment will be given by POS . These elaborated details can be implemented quite easily in a programming language as a function with two parameters, which will be shown in our FDR script later.

With this derivation of POS , we have solved the problem constructively.

4.5.3 Using *SKIP* instead of *STOP*

In the original theory of CSP [67], Hoare points out that “the process $STOP$ is defined as one that never engages in any action. It is not a useful process, and probably results from a deadlock or other design error, rather than a deliberate choice of the designer”. He suggests that in order to describe a process that terminates successfully, i.e., a process that accomplishes everything that it was designed to do and it should do nothing more, a different notation $SKIP$ should be used. He proposes to represent a successful termination as a special event, denoted by the symbol \surd .

According to [67], the first and only action of the process $SKIP$ is successful termination, so it has only two traces $traces(SKIP) = \{\langle \rangle, \langle \surd \rangle\}$. Lai and Sanders [101]

have not given the predicative semantics to *SKIP*. However, we can give the following predicative semantics to *SKIP*:

$$SKIP_A(tr, ref) \triangleq (tr = \langle \rangle) \wedge (ref \subseteq A \setminus \{\sqrt{\}\}).$$

In computer programming, the explicit distinction between *STOP* and *SKIP* is fully justified when they are used in describing the behaviour of computer programs, and proving freedom from deadlock is usually an important task and good practice in program design. Therefore, if we want to construct a machine that is deadlock-free, then we could have made the specification stronger by replacing $STOP_{\alpha POS}$ with $SKIP_{\alpha POS}$, like the following:

$$\begin{aligned} POS_{stronger} &= \prod_{i \in \{1, \dots, 100\}} item?i \rightarrow notice!i \rightarrow POSPAY(i, i), \text{ where} \\ POSPAY(i, remain) &= \\ &\prod_{p \in \{1, 2, 5, 10\}} pay?p \rightarrow \text{if } p < remain \text{ then } (notice!(remain - p) \\ &\quad \rightarrow POSPAY(i, remain - p)) \text{ else } (change!(p - remain) \\ &\quad \rightarrow receipt!i \rightarrow SKIP_{\alpha POS_{stronger}}) \end{aligned}$$

However, in this thesis, we regard the above replacement as a decision of the programmer, rather than an obligation of our derivation. Indeed, our derivation based on Lai's quotient only leads to the weakest specification, i.e., $POS_{stronger}$ is stronger than POS .

4.5.4 Validating the Derived Specification Using FDR

We have adapted the process expressions of *CUST*, *REQ*, *REQC* and *POS* to FDR scripts, as shown in Figure 4.6 (next page).

In the FDR script, we have allowed the value of the items i to range from 1 to 100, and the allowable payment to be any one of 1, 2, 5, and 10. FDR check confirms the calculated machine specification POS in parallel with $CUST$ does refine the orig-

```

-- The self-help POS problem
-- Using FDR to check that the solution machine POS is correct
-- (to satisfy the requirement REQ) when placed in parallel with the
-- customer domain CUST

-- First, the set of values to be communicated, of Money type
-- i, p, c and r are in Sterling (pence)

-- Channel declarations, specifying that the values communicated over them are of Money or Display type
channel item, pay, receipt, change, notice, leave : {0..100}

-- Describing the customer domain as a process CUST
CUST = l~l i : {1..100} @ item!i -> notice?i -> PAY

-- Describing the payment process as PAY so that CUST can be defined easily
PAY = l~l p : {1, 2, 5, 10} @ pay!p -> (change?c -> receipt?i -> STOP) [notice?n -> PAY]

-- Describing the requirement REQ
-- REQ only specifies what is required, that is, a success scenario "r=i", so "pay", "change" and "notice"
-- are hidden; other scenarios should be ignored, or a warning display should be issued.
REQ = l~l i : {1..100} @ item.i -> receipt.i -> STOP

-- The derived solution POS (should be the same as that calculated using Lai's quotient)
-- Note first i in POSPAY(i,i) does not change; while second i keeps changing to reflect the remaining
-- payment needed
POSPAY = l~l i : {1..100} @ item?i -> noticeli -> POSPAY(i,i)

-- Since it's a card payment, "r==p" is the condition under which the machine issues a receipt; otherwise a
-- warning display should be issued to the customer
POSPAY(i,remain) = l~l p : {1, 2, 5, 10} @ pay?p -> if p<remain then (notice!(remain-p)
-> POSPAY(i,remain-p)) else (change!(p-remain) -> receipt!i -> STOP)

-- REQ by concealing {notice, pay, change}
REQC = l~l i : {1..100} @ item.i -> notice.i -> REQCPAY(i,i)
REQCPAY(i,remain) = l~l p : {1, 2, 5, 10} @ pay.p -> if p<remain then (notice.(remain-p)
-> REQCPAY(i,remain-p)) else (change.(p-remain) -> receipt.i -> STOP)

-- checking if CUST || POS refines/satisfies REQ
IMPL1 = CUST [K{item, pay, notice, change, receipt}] || POS

-- checking if CUST || POS refines/satisfies REQ
IMPL2 = (CUST [K{item, pay, notice, change, receipt}] || POS) \{pay, change, notice\}

```

Fig. 4.6: Model-checking the derived machine specification for the POS problem, using FDR developed by Formal Systems Europe Ltd.

inal requirement $REQC$, that is $IMPL1 = CUST || POS$ refines $REQC$, as shown in Figure 4.7. Likewise, POS in parallel with $CUST$ (by hiding events $present(pay)$, $present(change)$ and $present(notice)$) does refine the original requirement REQ , as shown in Figure 4.8.

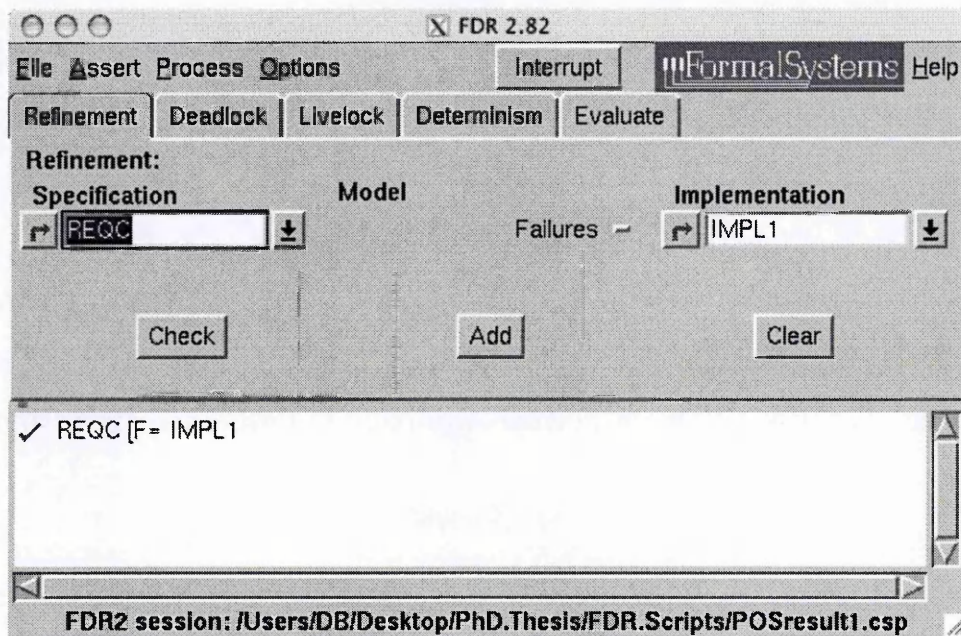


Fig. 4.7: Model-checking the derived machine specification for POS problem, checking if $IMPL1$ refines/satisfies $REQC$

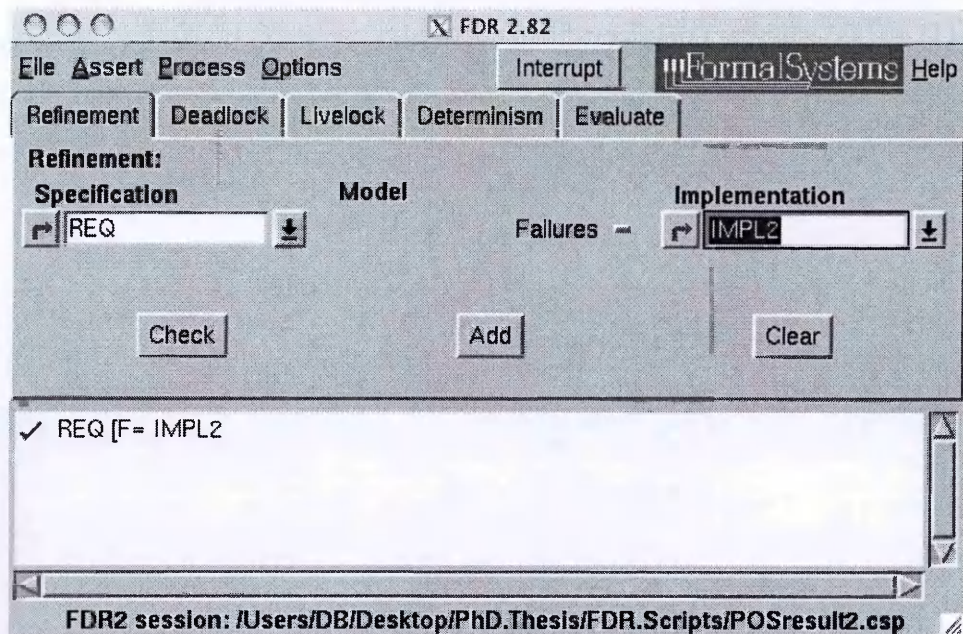


Fig. 4.8: Model-checking the derived machine specification for POS problem, checking if *IMPL2* refines/satisfies *REQ*

4.6 Discussion on our Formal Approach to Problem Progression

4.6.1 Complexity

We have shown the derivation of a solution to a problem, using a formal approach to problem progression. Even though the problem was simple, its formal solution required a complex process of formalisation and associated manipulations. For any problem of realistic complexity, it is unlikely that the approach will be tractable, even with tool support. Moreover, requirements engineering involves activities and communication amongst many non-technical stakeholders, and we can not assume that practitioners have knowledge of CSP and the predicate calculus. Therefore, other ways of making our techniques transparent to a general audience are needed. Although slightly disappointing, it is by no means unexpected. Many sources relate the difficulties of applying

formalism in the real world [157].

4.6.2 Weakening Problem Descriptions

As shown previously, in the same specification space *Spec*, logical implication \Rightarrow (informally interpreted as “stronger than”) is equivalent to satisfaction **sat** in terms of a set of traces *tr* and refusals *ref* (in other words, under the stable failures model in CSP). This is the context where the notion of *weaker* or *stronger* is defined. It is concerned with the *implication or satisfaction ordering on predicates* [101]. The terms *stronger* and *weaker* provide a way to express the relative relationships between specifications in the *Spec* space or between processes in the *Proc* space. The formal semantics of “A is stronger than B” can be interpreted formally as $A \text{ sat } B$, or $A \Rightarrow B$ if *A* and *B* are specifications.

Based on the notion of implication ordering, deriving the weakest sub-component process from the whole process and the other sub-component process using Lai’s quotient operator may provide a useful theoretical tool to reason in the *Spec* space about CSP descriptions: for example, in the PF semantics formula, let *S* stands for the machine specification, *K* for the whole domain description, and *R* for the overall requirement. Since $S \parallel K \text{ sat } R$ holds, we know that the solution $K \backslash\!\!\backslash R$ is the weakest solution - in the *Spec* space, anything stronger than it is a solution to the problem; anything weaker is not, in other words, if the actual designed machine specification S_{designed} is stronger than $K \backslash\!\!\backslash R$, then we have the grounds to argue that it is a solution; otherwise, it is not a solution.

As shown in the previous section, for many non-trivial software development problems, a fully formal description of domains and requirements can not be easily obtained. This concerns the difference between modelling and reality - most of the time, the in-

formal domain and requirement descriptions are not strong enough for making useful formal argument. Therefore, in order to address a wider variety of problems, we need less formal approaches to deal with informal descriptions.

4.7 Summary

In this chapter we have proposed a formal technique for problem progression based on CSP and in the context of the denotational semantics of problem diagrams defined by Hall *et al.* [63]. The technique was applied to an example, and formal descriptions were verified through the FDR tool.

There are many technical issues we have not discussed: for instance, we have not modelled divergent behaviours. If a process P performs internal transitions forever, never reaching a stable state nor performing any external event, then it is said to be *divergent*, denoted as $P \uparrow$ [146].

From an outside observer of a process P , we can only reason about its guaranteed external behaviour when it is stable. As Schneider [146] points out, the stable failures model completely ignores any divergent behaviour that a process might have (page 221). This is the assumption of the failures model in CSP - its primary focus is on guaranteed behaviour rather than divergent behaviour, and from the PF perspective, this is a limitation that might be treated informally, e.g., the standard concerns (e.g., reliability concerns) in problem frames [83] take into account possible divergent behaviours of a domain (process), and the state-machine diagram in PF can semi-formally express an unknown/broken state of a domain by using a box that contains a question mark [83].

Whereas we expect technical solutions to these issues to exist, it is unlikely that addressing them will move us any closer to a practical approach to problem progression.

Thus they remain unexplored. In the next chapter we look to define more practical approaches.

5. A SEMI-FORMAL APPROACH TO PROBLEM PROGRESSION

This chapter introduces a less-formal approach to problem progression than the previous one. It takes causality as its foundation. By relaxing some of the restrictions imposed by the CSP language, we demonstrate that causality can give us more widely-applicable techniques for problem progression without resorting to a fully formal description language.

The chapter gives a working definition of causality and demonstrates how some derived notations and techniques can help underpin problem progression in a systematic way. The main contribution of this chapter is a set of rules for the practical achievement of problem progression. They will be applied in a number of case studies in the next chapter.

In order to illustrate causality and associated concepts and techniques for problem progression, we will use the following two examples. The first example is the heating control problem of Chapter 3 whose problem diagram we recall here for convenience (Figure 3.1 in Chapter 3 recalled here as Figure 5.1).

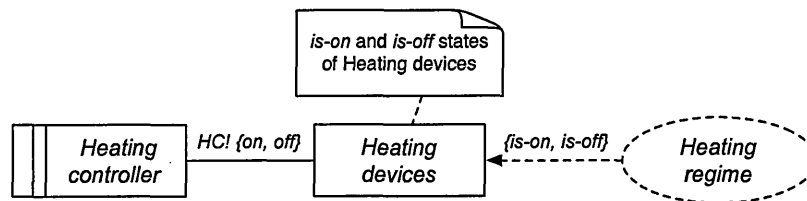


Fig. 5.1: A simple heating control problem, with added annotations for internal phenomena

The second example is a variant of the POS problem of Chapter 4, which represents a more traditional POS system. The problem diagram is given in Figure 5.2. We will return to this problem in Chapter 6, where we will provide further details.

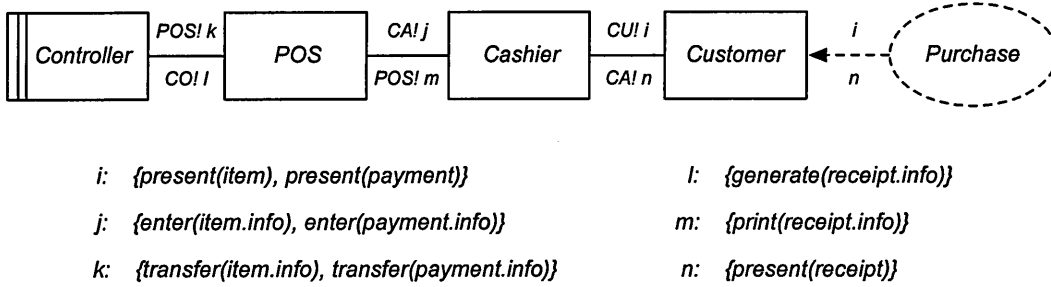


Fig. 5.2: The POS problem diagram

5.1 Causality

According to the Oxford English Dictionary, the word “causality” generally refers to “the relationship between cause and effect”. This general meaning of causality is ubiquitous in our everyday life, and it is shared among various branches of social and natural sciences, such as philosophy, logic, physics, and psychology, etc. For example, Hopkins [74] points out that the notion of causality was first studied and researched in philosophy by Aristotle. Then in the 17th century, Francis Bacon (1561-1626) introduced causality into science by establishing that causality could be open to empirical investigation. In the 18th century, David Hume (1711-1776) shifted the study of causality from logic to psychology and established his defining characteristics of causality. However, Hume’s theoretical characterisation was challenged by John Stuart Mill (1806-1873) with his notion of multiple causation in contrast to the simple, linear causality adopted by Hume. As summarised by Hopkins, Mill’s notion of causation “was something that

occurred through the grace of multiple intersections of interweaving causal lines and none of which on their own brought about an effect” [74].

Moffett *et al.* [111] observe that causal reasoning is a useful tool for describing mechanisms, problems and solutions. They propose a formal causal language for requirements specification to fill the gap between natural language and formal reasoning in RE.

For the purpose of this thesis (e.g., underpinning problem progression) and in line with the work of Moffett *et al.* [111], we define *causality (or causation)* as the relationship between cause and effect, which we formalise as a relation between pairs of events. By focusing on events, we have a working definition capable of describing the behaviour of problem domains.

5.1.1 Basic Notation

From Moffett *et al.*’s work [111], we adopt the following basic concepts and notations to be used for problem progression:

- we distinguish between an *event* and an *occurrence* of an event; for instance, the single event occurrence “bell rings” can typically occur many times in the lifetime of the bell.
- we regard *cause* as a relationship between events which induces a relationship between occurrences of those events. Notationally, we use:
 - \rightsquigarrow to indicate direct cause: given two events ev_1 and ev_2 , $ev_1 \rightsquigarrow ev_2$ indicates that an occurrence of ev_1 is the immediate cause of an occurrence of ev_2 ; and

- \rightsquigarrow^+ to indicate the transitive closure of \rightsquigarrow : given two events ev_1 and ev_2 , $ev_1 \rightsquigarrow^+ ev_2$ indicates that an occurrence of ev_1 eventually leads to an occurrence of ev_2 , possibly through a chain of other event occurrences.

The causal relationship “button pressed” \rightsquigarrow “signal sent” is an example of the first form; “button pressed” \rightsquigarrow^+ “bell rings” is an example of the second form, where “button pressed” \rightsquigarrow “signal sent” and “signal sent” \rightsquigarrow “bell rings”. In this way, causality is an irreflexive transitive relation between events.

Of course, the distinction between the two forms of causality depends on the level of granularity in the analysis: if we abstract away the “signal sent” event, then “button pressed” \rightsquigarrow “bell rings”.

- Like Moffet *et al.* [111], we make a clear distinction between *sufficient* and *necessary* cause. The difference between sufficient cause and necessary cause is that the former is expressed in a positive statement while the latter is expressed in a double negative statement: if ev_1 is a sufficient cause for ev_2 , then the occurrence of ev_1 is inevitably followed by the occurrence of ev_2 (this is a positive statement); if ev_1 is a necessary cause for ev_2 , then given the presence of its enabling conditions, if ev_1 *does not* occur then ev_2 *will not* occur (this is a double negative statement). Moffet *et al.* observe that it is easiest to think in terms of sufficient cause when working with practical examples, instead of double negations of necessity. Throughout this thesis, the word “cause” refers to sufficient cause.

To represent state changing events, the following notation is used: given a state st of a domain D , the event corresponding to D entering the state st is denoted by $\uparrow st$.

In this thesis, behaviours that involve sequences of event occurrences are represented as traces. For convenience, we label an event occurrence with the name of the

event. For example, we use the following notations to describe a simple heating device's behaviours:

$$\langle \text{switch_on}, \uparrow \text{working} \rangle$$

$$\langle \text{switch_off}, \uparrow \text{stopped} \rangle.$$

5.1.2 Types of Causality

In order to adapt causality to a variety of problems, we distinguish between the following types of causality (see how they are used in later examples):

Simple causality: An occurrence of one event is the cause of an occurrence of another event. For example, “*pressing button i in the lift will make the lift cabin arrive on floor i* ” expresses a simple causality. Formally, we can express this simple causality as $\text{pressButton}(i) \rightsquigarrow \text{cabinArrivesOnFloor}(i)$. In RE practice, simple causality is useful for communicating intuitive knowledge about causal aspects of events among stakeholders.

Conditional causality: An occurrence of one event ev_1 is the cause of another event ev_2 , guarded by some condition g . In other words, the causal relation holds only when some condition is true. We use the following notation to express conditional causality: $(g) : ev_1 \rightsquigarrow ev_2$, where g is a Boolean condition (g will be omitted when it is always true). For example, at a lower level of abstraction, the event “*pressing button i in the lift*” causes the event “*the lift cabin arrives on floor i* ” only when some Boolean condition such as “*proper mechanical operations of cables, motors and correct electrical signal transmission*” is true. The causal relation can be described precisely as follows (\wedge is logical conjunction):

$$(\text{button}(ok) \wedge \text{electricalSignal}(ok) \wedge \text{controller}(ok) \wedge \text{motor}(ok) \wedge \text{cable}(ok)) : \text{pressButton}(i) \rightsquigarrow \text{cabinArrivesOnFloor}(i).$$

Please note that when the guard is trivially true, conditional causality is reduced to simple causality, in which case the guard g is omitted.

Timed causality: An occurrence of one event ev_1 is the cause of another event ev_2 , separated by a time duration ΔT . We use the following notation to express timed causality: $ev_1 \xrightarrow{\Delta T} ev_2$. Timed causality is useful in the analysis of real-time systems where timing issues are critical. For instance, whenever a lift user presses the emergency button, the lift cabin must be stopped within a very short time (for example 0.5 second). This causal relation can be described precisely as:

$$pressButton(emergency) \xrightarrow{0.5s} liftCabin(stopped).$$

Please note that when time can be ignored, timed causality is reduced to simple causality, in which case the time duration ΔT is omitted.

Biddable causality: Biddable causality is a relationship we introduce to describe and reason about the behaviour of people. Biddable causality is not true causality in that there is no physical law that allows us to establish the relationship between cause and effect, however it is a relationship between cause and effect that can be expected, e.g., by training: although a human being may have free will or exhibit random behaviour, we can still manage to constrain their behaviour to a certain extent by training. We use the following notation to express biddable causality: $ev_1 \xrightarrow{bidden} ev_2$. In the POS example (see Figure 5.2), we have good reasons to expect the *Cashier* domain to behave like a causal domain because he or she has received training in processing customer's items and payment. Therefore, we can expect that whenever a cashier is presented an item of product, he or she should faithfully enter the item's information into the *POS* domain. In other words, we can expect the following causal relation:

$$present(item) \xrightarrow{bidden} enter(item.info).$$

5.2 Causality in Problem Description

In this section, we apply the notion of causality to two important aspects in problem description - describing causal behaviours of different types of domains and associating causality with control of phenomena. Furthermore, we introduce the notion of a causal chain that allows us to reason through chains of behaviour in a problem description. In the next section we will argue that enhancing problem descriptions with causality provides a basis for problem progression.

5.2.1 Using Causality to Describe Domain Behaviour

Describing the causal aspects of a domain plays an important part in reasoning about its properties and behaviours, which is one of the crucial activities in problem progression. For this purpose, we need to consider the nature of a domain. Jackson [83] distinguishes the following two types:

A *causal* domain is one whose properties include *predictable* causal relationships among its phenomena. For instance, in the POS example, the *POS* domain is considered as a causal domain: whenever the item's information is entered, the *POS* domain will transfer the item's information to the *Controller* domain, that is, $enter(item.info) \rightsquigarrow transfer(item.info)$. (Of course, it is assumed that the *POS* domain operates reliably.)

A *biddable* domain usually consists of people, who lack predictable internal causality. As argued in the previous section, a biddable domain can be *bidden*, but *not forced* to do something. So generally speaking, causality cannot be claimed for a biddable domain; still there is a possibility that some causal relationship between its events can be assumed with reasonable justifications (e.g., through training).

For completeness, Jackson [83] also introduces lexical domains which are physical

representations of data. Since causal properties allow the data to be read and written, we can always treat lexical domains as causal in problem progression.

5.2.2 Relationship between Control and Causality of Phenomena

In the PF framework, domains interact with each other through shared phenomena. The sharing of phenomena is something that all sharing participants are part of simultaneously, as Jackson states in [83]:

“The participation in a shared event is like a hammer hitting a nail: there’s only one event, and the hammer and the nail both take part in it simultaneously”.

Sharing is not always limited to two participant domains (an example can be found on page 52 in [83]). For a shared phenomenon, all sharing participants have access to it. We can see in Figure 5.2 how shared phenomena are represented in a problem diagram as annotated arcs linking domains.

Phenomena which are not shared are private (thus hidden inside boxes). For instance, in Figure 5.2 we can imagine there are *scan(item.info)* (the item’s barcode is scanned into an optical signal information) and *convert(item.info)* (the item’s optical information is transferred to electrical information) events private to the *POS* domain. All private phenomena of a domain are, by default, controlled by that domain.

For a shared phenomenon, only one sharing domain has control. The notion of control has slightly different interpretations depending on the type of phenomenon. For example, if the phenomenon is an event *ev*, “domain *D* controls *ev*” means that *D* initiates an occurrence of event *ev* and that if *ev* is shared between domains *D* and *D'*, only *D* can initiate its occurrence; if the phenomenon is a state *st* of domain *D* shared

with domain D' , then “ D controls st ” means that only D can change the state, although the change is visible to D' .

Although the notions of control and causality are related, their focus is different: identifying shared phenomena and which domain controls them allows us to reason about interactions among domains; while identifying causal relations within a domain allows us to reason through the behaviour of the domain. We argue that by exploiting the two notions together, we can reason about chains of behaviour in a problem description so that a systematic way of problem progression can be achieved.

Let us look at Figure 5.2. As we will see later on (next chapter), to achieve problem progression, in addition to relying on control annotations in the diagram we also need causal notations to elaborate and reason about domain properties.

Here is an example of the chain of causal events in domain *POS* along which the item’s information is read by a barcode reader and the optical signal is converted into electrical signal, and finally the electrical signal is transferred into the *Controller* domain:

- firstly whenever the cashier enters the item’s information, the barcode reader scans the information on the item, so $enter(item.info) \rightsquigarrow scan(item.info)$;
- then whenever the item’s information is correctly scanned, the optical-to-electrical unit converts the optical signal into electrical signal, so $scan(item.info) \rightsquigarrow convert(item.info)$;
- finally whenever the signal is converted, it is then allowed to be transferred onto the computer, so $convert(item.info) \rightsquigarrow transfer(item.info)$.

By combining the above three causal relations, we obtain:

$enter(item.info) \rightsquigarrow scan(item.info) \rightsquigarrow convert(item.info) \rightsquigarrow transfer(item.info)$.

At a higher level of abstraction, we can simplify and represent the above set of causal relations as:

$$\text{enter}(\text{item.info}) \rightsquigarrow^+ \text{transfer}(\text{item.info})$$

From above, we can see that the same causal relation can be folded into a single \rightsquigarrow^+ notation or unfolded into a long chain of causal relations connected by the \rightsquigarrow notation. We name this unfolded expression of causality as a *causal chain*.

5.3 Progressing Problems Based on Graph Grammar

This section introduces a semi-formal technique for achieving problem progression. It is based on a set of rules adapted from a general framework of problem orientation by Hall *et al.* [64]. Hall *et al.* [64] have given a formal conceptual framework for problem-oriented software engineering, where problem progression is one of many problem transformation classes. In that framework, problem progression consists of two steps - removing shared phenomena and removing domains from a problem context. The notion of a problem in that framework is represented as a *sequent* in a Gentzen-style calculus [95]. The sequent is cast in the general form of $W, S \vdash R$, where W represents the problem world (given domain description), S represents the solution (specification statement), and R represents the requirement (statement).

Departing from Hall *et al.*'s work in [64], we provide an interpretation of those rules in the context of problem frames. The results are three classes of constructive rules for problem progression based on the notion of causality. To this end this thesis makes use of an algebraic approach to graph representation and transformation using graph grammars [11, 48]. This is motivated by our observation that the manipulation of problem diagrams in problem progression can be regarded as graph transformation

following some constraining graph grammar rules.

This section gives an introduction to the relevant concepts and definitions of graph grammars, which are used in our rule-based approach to problem progression. We then define three classes of rules in the next section and show that they are applicable to progressing a variety of problems.

5.3.1 Graph Grammars

Graph grammars provide a formal foundation for the manipulation of graph structures. They have been used widely in computing [33], for example, the algebraic approach of graph grammars [46, 48] has lead to useful results in parallelism analysis [98, 99], evaluation of functional expressions and logic programs [117, 34], synchronisation mechanisms [15], distributed systems [47, 145], and object-oriented systems [96].

Basic Concept and Definition

A *graph* consists of a set of *vertices* V (sometimes called *nodes* or *points*) and a set of *edges* E (sometimes called *arcs* or *lines*), and each edge e in E has a *source vertex* $s(e)$ in V and a *target vertex* $t(e)$ in V [11]. A *directed graph* is a graph in which every edge has a distinguished start vertex (its source) and end vertex (its target) [48].

In an algebraic style [48], a graph can be represented as $G = (V, E, s, t)$, where V is a (finite) set of vertices and E is a (finite) set of edges such that $V \cap E = \emptyset$; $s, t : E \rightarrow V$ are the source and target functions, respectively (see below). A *subgraph* of a graph G is a graph whose vertex and edge sets are subsets of those of G .

$$E \begin{array}{c} \xrightarrow{s} \\ \xrightarrow{t} \end{array} V$$

For example, an algebraic representation of Figure 5.3 is $G = (V, E, s, t)$, where $V = \{u, v, x, y\}$ is the vertex set, $E = \{a, b\}$ is the edge set, with source function $s : E \rightarrow V : s(a) = u, s(b) = u$, and target function $t : E \rightarrow V : t(a) = v, t(b) = v$.

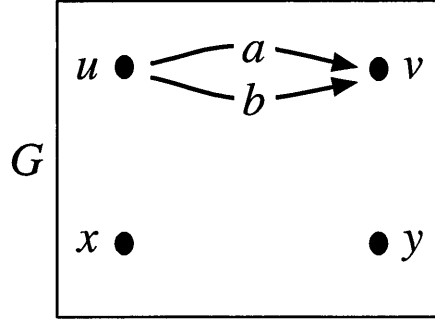
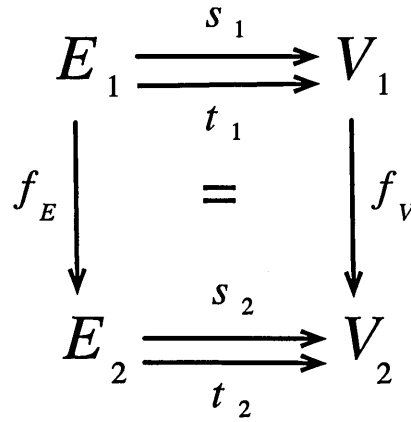


Fig. 5.3: A graph example, adapted from [48]

Graph morphism: Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$, a graph morphism $f : G_1 \rightarrow G_2, f = (f_V, f_E)$ consists of two functions $f_V : V_1 \rightarrow V_2$ and $f_E : E_1 \rightarrow E_2$ which preserve the source and target functions, that is, $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$ [48] (the symbol \circ is the function composition operator), as shown in the commutative diagram below (adapted from [48]):



For example, Figure 5.4 shows a graph morphism between two graphs G_1 and G_2 , where:

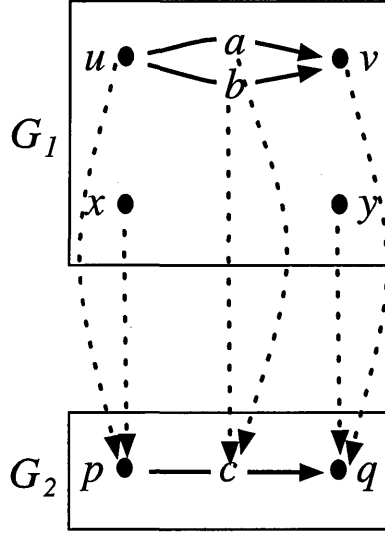


Fig. 5.4: A graph morphism example, adapted from [48]

$$G_1 = (\{u, v, x, y\}, \{a, b\}, s_1, t_1),$$

with $s_1(a) = u$, $s_1(b) = u$; $t_1(a) = v$, $t_1(b) = v$; and

$$G_2 = (\{p, q\}, \{c\}, s_2, t_2),$$

with $s_2(c) = p$, $t_2(c) = q$.

The graph morphism is $f : G_1 \rightarrow G_2 = (f_V, f_E)$,

with $f_V : V_1 \rightarrow V_2 : f_V(u) = p, f_V(x) = p, f_V(v) = q, f_V(y) = q$, and

$f_E : E_1 \rightarrow E_2 : f_E(a) = c, f_E(b) = c$.

A graph morphism f is *injective* if both f_V and f_E functions are injective - in discrete mathematics [121], the function (mapping) $f : A \rightarrow B$ is injective, if $f(x_1) = f(x_2)$ only when $x_1 = x_2$, where $x_1, x_2 \in A$ and $f(x_1), f(x_2) \in B$. The sets A and B are known as the *domain* of f and the *codomain* of f , respectively.

For example, Figure 5.5 shows an injective graph morphism from G_1 to G_2 , where every edge in G_1 maps to one distinct edge in G_2 . Also every vertex in G_1 maps to one distinct vertex in G_2 .

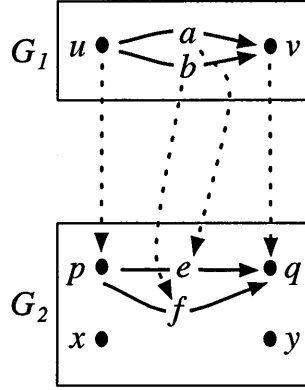


Fig. 5.5: An injective graph morphism example adapted from [48]

Formally:

$$G_1 = (\{u, v\}, \{a, b\}, s_1, t_1), \text{ and}$$

with $s_1(a) = u, s_1(b) = u; t_1(a) = v, t_1(b) = v$, and

$$G_2 = (\{p, q, x, y\}, \{e, f\}, s_2, t_2),$$

with $s_2(e) = p, s_2(f) = p; t_2(e) = q, t_2(f) = q$.

The graph morphism $f : G_1 \rightarrow G_2 = (f_V, f_E)$ is injective, because

$f_V : V_1 \rightarrow V_2 : f_V(u) = p, f_V(v) = q$ is injective (the fact that vertices x and y in V_2 have no pre-image in V_1 does not prevent f_V from being injective), and $f_E : E_1 \rightarrow E_2 : f_E(a) = e, f_E(b) = f$ is injective.

A *labelled graph* (also known as a *coloured graph* [32]) $G = (V, E, s, t, l_V, l_E)$ consists of an underlying graph $G^0 = (V, E, s, t)$ together with two label functions $l_V : V \rightarrow L_V$ and $l_E : E \rightarrow L_E$, where L_V and L_E are alphabet sets of vertex labels and edge labels, respectively [48].

$$L_E \xleftarrow{l_E} E \xrightleftharpoons[t]{s} V \xrightarrow{l_V} L_V$$

For example, Figure 5.6 shows a labelled graph G , which is obtained by adding labels to all the vertices and edges in Figure 5.3.

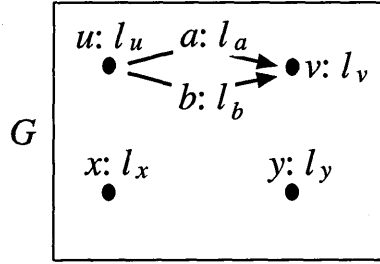


Fig. 5.6: A labelled graph example

We follow the conventions in [32] to use “:” to separate the vertex/edge name from its label. Although sometimes we omit these labels to avoid making the diagram overcrowded, we always express these labels in an algebraic style, that is:

$$\begin{aligned} G &= (\{u, v, x, y\}, \{a, b\}, s, t, l_V, l_E), \text{ where} \\ s(a) &= u, s(b) = u, t(a) = v, t(b) = v; \\ l_V(u) &= l_u, l_V(v) = l_v, l_V(x) = l_x, l_V(y) = l_y; \\ l_E(a) &= l_a, l_E(b) = l_b. \end{aligned}$$

A labelled graph morphism $f : G_1 \rightarrow G_2$ is a graph morphism $f : G_1^0 \rightarrow G_2^0$ between the underlying graphs which is compatible with the label functions, that is, $l_{2,V} \circ f_V = l_{1,V}$ and $l_{2,E} \circ f_E = l_{1,E}$ [48], as shown in Figure 5.7 (next page).

In graph theory, a labelled graph morphism is defined to preserve the following three kinds of mapping relationships between two labelled graphs:

1. mapping relationships between vertices are preserved, by $f_V : V_1 \rightarrow V_2$;

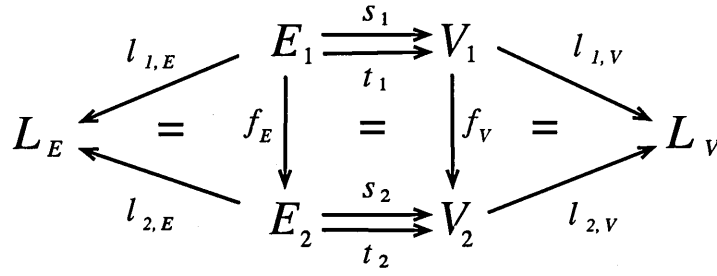


Fig. 5.7: A labelled graph morphism, taken from [48]

2. mapping relationships between edges are preserved, by $f_E : E_1 \rightarrow E_2$;
3. mapping relationships between labels are preserved, by $l_{2,V} \circ f_V = l_{1,V}$
and $l_{2,E} \circ f_E = l_{1,E}$.

A *graph production* (also known as a *rule*) $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of graphs L , K , and R , called the *left-hand side*, the *gluing graph or interface*, and the *right-hand side*, respectively, and two injective graph morphisms $l : K \rightarrow L$ and $r : K \rightarrow R$ [48].

Because of the injective morphisms, the interface graph K remains the common structure shared between L and R . In other words, graph K represents the subgraph which is common to both L and R under the graph production rule, while other graph structures (those represented by the sets $L \setminus K$ and $R \setminus K$ - “left-over” structures due to the injective functions - the codomain of an injective function may have extra elements that are not mapped by the function) represent those structures which are different.

Production rules are the basis for the definition of *graph transformation*. Suppose that we have a graph G and a production rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$; transforming G by using p means the following:

- identifying a subgraph in G which matches the structure of L . Formally we do this through a graph morphism $m : L \rightarrow G$ called “match”; and

- transforming $m(L)$ according to p . This has the effect of replacing the subgraph $m(L)$ in G with a subgraph whose structure is defined through p , which leads to a new graph H .

Such a transformation is represented by $G \xrightarrow{p,m} H^1$.

Here is an example (for brevity in this thesis we adopt the following convention in representing a graph transformation: we use the same names for elements which remain invariant through the transformation, for instance, v_1, v_2, v_3 in Figure 5.8).

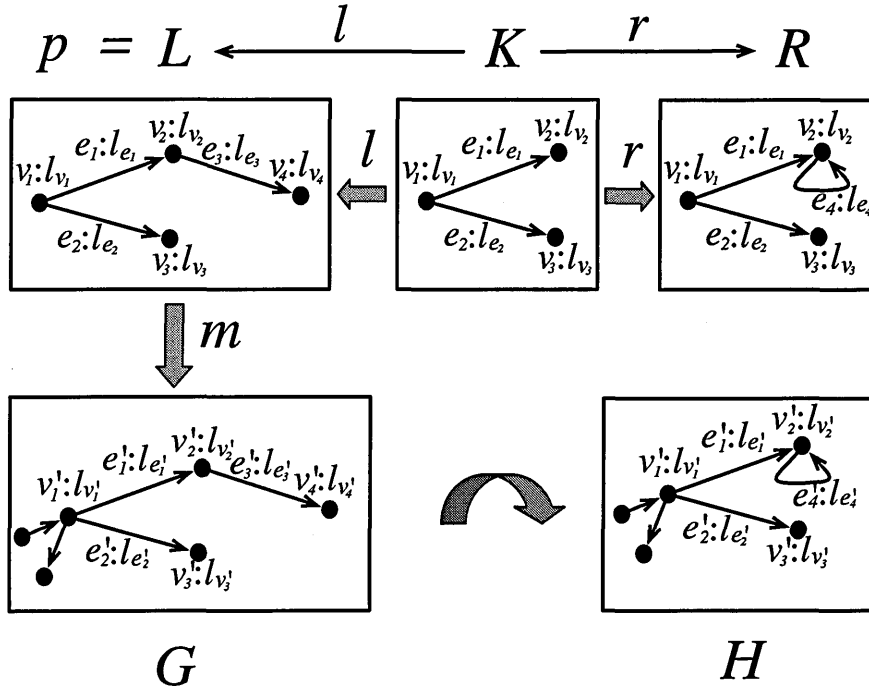


Fig. 5.8: An example of graph transformation $G \xrightarrow{p,m} H$ based on a production rule p and an injective match m

Figure 5.8 illustrates a graph transformation from G to H based on a production rule

¹ For our purposes, this is all we need to know about graph transformation. For a more detailed and fully formal treatment, please see [48].

$p = (L \xleftarrow{l} K \xrightarrow{r} R)$ (the darkened straight arrow represents injective graph morphism; while the bent darkened arrow represents graph transformation), where:

$$\begin{aligned} l : K \rightarrow L : l(v_1) = v_1, l(v_2) = v_2, l(v_3) = v_3, l(l_{v_1}) = l_{v_1}, l(l_{v_2}) = l_{v_2}, \\ l(l_{v_3}) = l_{v_3}, l(e_1) = e_1, l(e_2) = e_2, l(l_{e_1}) = l_{e_1}, l(l_{e_2}) = l_{e_2}; \end{aligned}$$

and

$$\begin{aligned} r : K \rightarrow R : r(v_1) = v_1, r(v_2) = v_2, r(v_3) = v_3, r(l_{v_1}) = l_{v_1}, r(l_{v_2}) = l_{v_2}, \\ r(l_{v_3}) = l_{v_3}, r(e_1) = e_1, r(e_2) = e_2, r(l_{e_1}) = l_{e_1}, r(l_{e_2}) = l_{e_2}; \end{aligned}$$

and

$$\begin{aligned} m : L \rightarrow G : m(v_1) = v'_1, m(v_2) = v'_2, m(v_3) = v'_3, m(v_4) = v'_4, \\ m(l_{v_1}) = l'_{v'_1}, m(l_{v_2}) = l'_{v'_2}, m(l_{v_3}) = l'_{v'_3}, m(l_{v_4}) = l'_{v'_4}, m(e_1) = e'_1, \\ m(e_2) = e'_2, m(e_3) = e'_3, m(l_{e_1}) = l'_{e'_1}, m(l_{e_2}) = l'_{e'_2}, m(l_{e_3}) = l'_{e'_3}. \end{aligned}$$

The production rule p specifies that the “left-overs” elements in L , i.e., e_3 , v_4 , l_{e_3} , and l_{v_4} , are deleted, and that the “left-overs” elements in R , i.e., e_4 and l_{e_4} , are added.

According to rule p , we need to delete e'_3 , v'_4 , $l'_{e'_3}$, and $l'_{v'_4}$ from G , and add e'_4 and $l'_{e'_4}$, to derive H , which is shown in the bottom right corner of Figure 5.8.

In this thesis, we want to restrict the way we can manipulate a problem diagram in problem progression - some elements of a graph are allowed to be changed under some conditions and some remain unchanged. A graph transformation through graph production rules matches this purpose nicely.

A *graph grammar* is defined to be a pair $GG = (G_0, P)$, where G_0 represents the starting graph, and P represents a set of graph productions rules [48]. In our work, we essentially define graph transformation by using production rules. A graph grammar

gives us a system in which we have an initial graph and a set of production rules that allow us to implement graph transformations.

5.3.2 Interpreting Problem Diagrams as Directed Labelled Graphs

Given the characteristics of a problem diagram and the definition of a graph, we propose to relate them in the following way.

We can regard problem diagrams as labelled graphs: boxes representing domains are vertices; dashed ovals representing requirements are also vertices; arcs linking domains are edges; dashed arcs linking domains and requirements are edges as well. The descriptions of domains, requirements and phenomena are labels. Since PF do not prescribe which language to use to describe the problem, these labels can be in either natural language or some formal language.

More precisely, we represent a problem diagram as a labelled graph (examples will follow immediately afterwards) as follows:

1. Domains and requirements as *vertices*: if $n > 1$ is the total number of domains plus the requirement, then we represent them as vertices $\{v_1, \dots, v_n\}$;
2. Phenomena sets as *directed edges* (or *directed arcs*): if $m > 1$ is the total number of phenomena sets (including shared phenomena between domains, requirement phenomena, or relevant internal phenomena of domains - part of the domain properties), then we represent them as edges $\{e_1, \dots, e_m\}$. The direction of the edges is represented through the source and target functions, based on the following:
 - (a) *Controlling* domain or *constraining* requirement as *source of an edge*: if a domain or a requirement represented by vertex v_i controls or constrains

phenomena set e_j , then v_i is the source of the directed arc (the end without the arrowhead);

- (b) *Observing domain or referencing requirement as target of an edge*: if a domain or requirement represented by vertex v_i observes or refers to phenomenon e_j , then v_i is the target of the directed arc (the end with the arrowhead);

Note that with this convention, problem diagrams can be represented as directed graphs so that each individual phenomenon and associated control-and-observe information is captured as a directed arc in the graph.

3. Domain and requirement descriptions through the *vertex label function* l_V as the mapping $l_V : V \rightarrow Types \times Names \times Descriptions$, where: $Types = \{Machine, Designed, Given, Requirement\}$; $Names$ is a set of names for domains and requirement; $Descriptions$ is a set of descriptions, for domains and requirement, in any formal, semi-formal or informal description language;
4. Phenomena descriptions through the *edge label function* l_E as the mapping $l_E : E \rightarrow \mathbb{P}(phenomena)$, where $\mathbb{P}(phenomena)$ is the power set of all phenomena in the problem diagram.

Let us revisit the simple heating control example of Chapter 3 whose problem diagram is re-drawn as Figure 5.1 (the dog-eared box indicates the internal phenomena of *Heating devices*). The problem diagram can be interpreted as the graph in Figure 5.9, encoding our formalisation (internal phenomena are represented as a reflexive arc e_3),

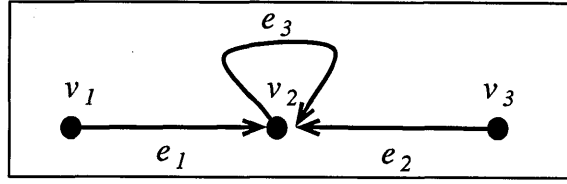


Fig. 5.9: Heat control problem diagram in Figure 3.4 represented as a directed graph - labels are omitted

where:

$l_V(v_1) = (\text{Machine, Heating controller, "the solution to be found"})$

$l_V(v_2) = (\text{Given, Heating devices, "devices that can be in either the is-on state or the is-off state. Pulse events on and off can effect state changing events in the devices, thus this domain is a causal domain. The heating devices have a mechanism to maintain the temperature"})$

$l_V(v_3) = (\text{Requirement, Heating regime, "the heating devices should be switched on at 8 : 45 am and switched off at 4 : 45 pm every day"})$

$l_E(e_1) = \{\text{on, off}\}$

$l_E(e_2) = \{\text{is - on, is - off}\}$

$l_E(e_3) = \{\text{is - on, is - off}\}.$

As further illustration, let us look at another example of the mapping between problem diagrams and labelled graphs. The following *occasional sluice gate control* problem is taken from [83].

"A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to raise and lower the sluice gate in response to the commands of an operator.

The gate is opened and closed by rotating vertical screws. The screws are

driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut. The connection to the computer consists of four pulse lines for motor control, two status lines for the gate sensors, and a status line for each class of operator command."

Figure 5.10 shows the problem diagram based on that given by Jackson in [83].

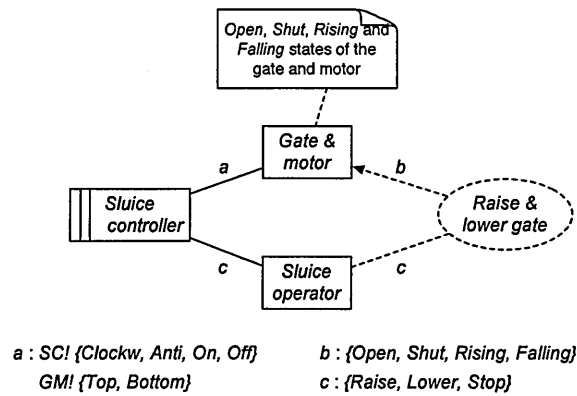


Fig. 5.10: Occasional sluice gate: problem diagram, adapted from [83]

The diagram can be interpreted as the graph in Figure 5.11, where:

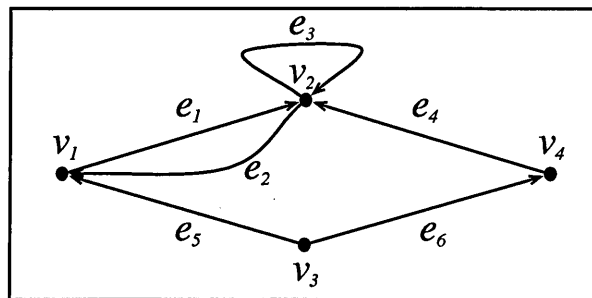


Fig. 5.11: Occasional sluice gate problem diagram in Figure 5.10 represented as a directed graph

$l_V(v_1) = (\text{Machine, Sluice controller, "the solution to be found"})$

$l_V(v_2) = (\text{Given, Gate \& motor, "The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut. The connection to the computer consists of four pulse lines for motor control, two status lines for the gate sensors"})$

$l_V(v_3) = (\text{Given, Sluice operator, "A person who sends out operating commands. Its connection to the computer consists of a status line for each class of operator command."})$

$l_V(v_4) = (\text{Requirement, Raise \& lower gate, "A computer system is needed to raise and lower the sluice gate in response to the commands of an operator."})$

$l_E(e_1) = \{\text{Clockw, Anti, On, Off}\}$

$l_E(e_2) = \{\text{Top, Bottom}\}$

$l_E(e_3) = \{\text{Open, Shut, Rising, Falling}\}$

$l_E(e_4) = \{\text{Open, Shut, Rising, Falling}\}$

$l_E(e_5) = \{\text{Raise, Lower, Stop}\}$

$l_E(e_6) = \{\text{Raise, Lower, Stop}\}.$

From the above two examples, we can see that representing a problem diagram as a labelled graph allows us to describe systematically the problem diagram as a mathematical object, which includes all relevant elements of the problem, that is, the topology of domain and shared phenomena plus all their descriptions.

To summarise, the motivation for regarding problem diagrams as directed labelled graphs is that we can apply production rules for their manipulations. Problem progression can then be regarded as a form of graph transformation, in which some graph arte-

facts such as vertices, edges or labels are removed or added under some defined graph production rules. For the purpose of representing progression rules and the application of such rules to a particular problem diagram, the pictorial style of graph representation gives us the intuition for matching the rules to a part of the problem diagram; while the algebraic style of graph representation provides the vehicle for rigorous manipulation during problem progression.

5.3.3 Interpreting Problem Progression as Rule-Based Graph Transformation

Having represented problem diagrams as directed labelled graphs, we aim at capturing problem progression through graph transformations.

Let us look at an example - the automatic heating control problem of Chapter 3 and its problem progression (Figure 3.8 is re-drawn here as Figure 5.12).

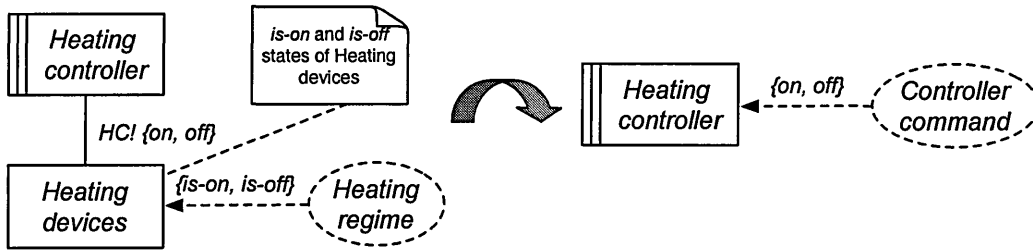


Fig. 5.12: Heating control problem progression diagram

The problem diagram progression of Figure 5.12 can be expressed as the graph transformation of Figure 5.13 under our interpretation of problem diagrams as directed labelled graphs.

In the figure, for graph G_1 :

$l_{V_1}(v_1) = (\text{Machine}, \text{Heating controller}, \text{"the solution to be found"})$

$l_{V_1}(v_2) = (\text{Given}, \text{Heating devices}, \text{"devices that can be in either the is-on state"})$

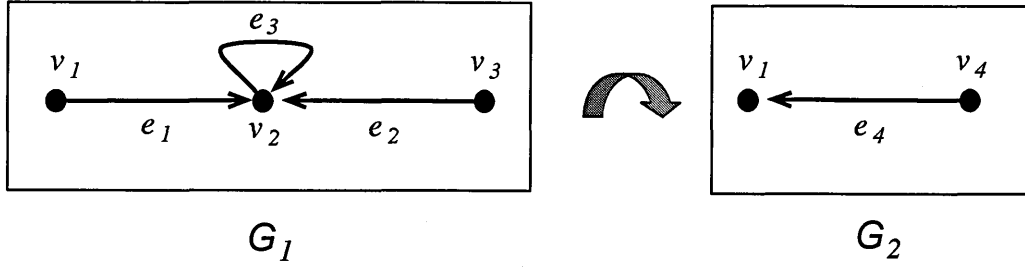


Fig. 5.13: Heating control problem progression diagram in Figure 5.12 as graph transformation
- labels are omitted

or the is-off state. Pulse events on and off can effect state changing events in the devices, thus this domain is a causal domain. The heating devices have a mechanism to maintain the temperature")

$l_{V_1}(v_3) = (\text{Requirement, Heating regime, "the heating devices should be switched on at 8 : 45 am and switched off at 4 : 45 pm every day"})$

$$l_{E_1}(e_1) = \{\text{on, off}\}$$

$$l_{E_1}(e_2) = \{\text{is - on, is - off}\}$$

$$l_{E_1}(e_3) = \{\text{is - on, is - off}\};$$

For graph G_2 :

$$l_{V_2}(v_1) = (\text{Machine, Heating controller, "the solution to be found"})$$

$l_{V_2}(v_4) = (\text{Requirement, Controller command, "the heating controller should issue the on command at 8 : 45 am and the off command at 4 : 45 pm every day"})$

$$l_{E_2}(e_4) = \{\text{on, off}\}.$$

In order to transform graph G_1 into graph G_2 , we need to define a set of basic production rules. We will demonstrate that the above transformation can be achieved through a particular combination of these rules. In the next section we will define a set of basic rules for problem progression, which are based on the notion of causality.

5.4 Causality-Based Rules for Problem Progression

In this section, we will define three separate classes of basic production rules which we use for progressing problems. They are:

1. the *Reducing through Cause and Effect* rule class: rules in this class generate a new requirement statement by replacing effects with causes, or causes with effects, based on the causal relations identified among events in domain descriptions. This rule class allows us to reason through the properties (behaviours) of a domain, thus allowing the requirement constraint or reference to be restated based on causal chains within domain descriptions.
2. the *Changing Viewpoint* rule class: rules in this class generate a new requirement statement based on the differing perspectives of domains sharing an event: switching from the perspective of a domain controlling the event to that of a domain observing the event, and vice versa. This rule class allows us to reason through the shared phenomena among domains.
3. the *Removing Domain* rule class: rules in this class are used to simplify problem diagrams, allowing us to remove a domain from consideration in the analysis, as long as corresponding assumptions are explicitly stated in the rewritten requirement. This rule class allows us to remove a domain and its shared phenomena in order to simplify further analysis.

As we will show, we can progress problems through a combination of the above rules. For instance, we can regard the transformation in Figure 5.12 as the result of applying the above rules in three steps:

1. applying the *Reducing through Cause and Effect* rule: from the effects, i.e., events $\uparrow is - on$ and $\uparrow is - off$, the original requirement statement is re-expressed as *Command received*: “the heating devices should receive the following commands from the heating controller: the on pulse command at 8:45 am and the off pulse command at 4:45 pm everyday”, which is described in terms of their causes, i.e., events *on* and *off*.

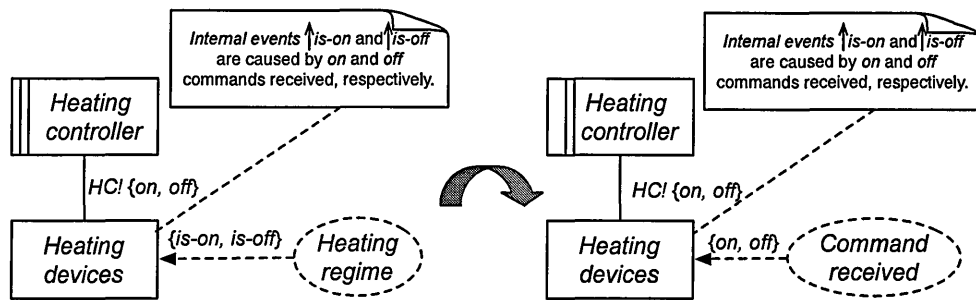
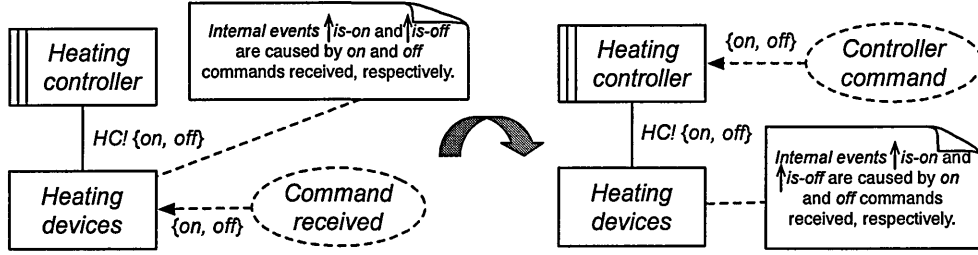


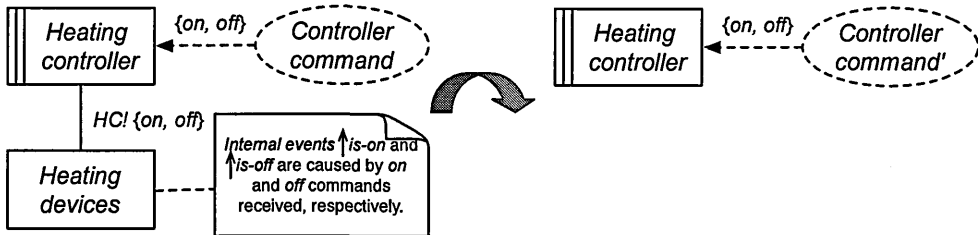
Fig. 5.14: Step1 - applying the *Reducing through Cause and Effect* rule

This rule can be applied because domain *Heating devices*' properties contain a causal relationship: “Internal events $\uparrow is - on$ and $\uparrow is - off$ are caused by *on* and *off* commands received, respectively”, which is represented in a dog-eared box in Figure 5.14.

2. applying the *Changing Viewpoint* rule: switching the viewpoint on the shared events *on* and *off* from the observer domain *Heating devices* to the controller domain *Heating controller*, the requirement is re-expressed in the heating controller's perspective (in Figure 5.15), i.e., *Controller command*: “the heating controller should issue the *on* command at 8:45 am and the *off* command at 4:45 pm every day”.

Fig. 5.15: Step 2 - applying the *Changing Viewpoint* rule

3. applying the *Removing Domain* rule: assuming the *Heating devices*' domain properties (that is, assuming that *on* and *off* commands will cause $\uparrow is - on$ and $\uparrow is - off$ respectively), the domain *Heating devices* and its shared phenomena are removed from the diagram (in Figure 5.16), resulting in the transformed problem diagram in Figure 5.12. The re-expressed requirement *Controller command'* becomes: "assuming that the *Heating devices*' domain properties hold, the heating controller should issue the *on* command at 8:45 am and the *off* command at 4:45 pm every day".

Fig. 5.16: Step 3 - applying the *Removing Domain* rule

The above is only a simple exercise to show that any controller that satisfies the specification *Controller command'* will satisfy the original requirement *Heating regime*.

The next section will present these basic progression rules in more detail.

5.4.1 The Reducing through Cause and Effect Rule Class

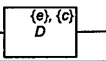
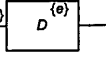
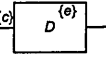
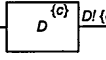
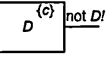
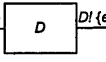
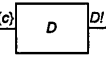
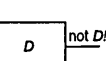
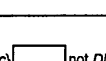
We call the first progression rule class *Reducing through Cause and Effect*. Rules in this class generate a new requirement statement by replacing effects with causes, or causes with effects, based on the causal relations identified among events in domain descriptions. We specialise this rule class into two sub-classes, namely the *effect-to-cause* (ETC) class and the *cause-to-effect* (CTE) class.

5.4.1.1 The Effect-to-Cause (ETC) Rule Class

Let us look at the effect-to-cause rule class first. The basis for this class of progression rules is that a causal relationship exists in a domain D 's description, i.e., $(g) : c \rightsquigarrow e$, where c and e are events of D , g is a Boolean condition that is part of the domain properties of D (g will be omitted when trivially true). To capture patterns of causality in natural language descriptions, we denote by “ ev occurs” any part of a requirement statement that implies an occurrence of event ev , and by “ g holds” the fact that g is true at that occurrence. Under this rule class, the requirement is rewritten so that any occurrence of an effect, say event “... e occurs ...”, is replaced by an occurrence of its guarded cause, say “... c occurs and g holds ...”.

Analysing Different Cases of Problem Topology

Because e and c can be internal to D , shared and controlled, or shared and observed by D , there are nine different cases in which the ETC rule class may apply. Each of these cases is characterised by a unique combination of topological relationships among e , c and D (including control and causal relationships), as shown in Table 5.1.

Case	Admissible (yes / no)	Description / Explanation
(1) 	yes	Both effect e and cause c are internal to D .
(2) 	yes	Effect e is internal to D , cause c is shared and controlled by D .
(3) 	yes	Effect e is internal to D , cause c is shared and observed by D (c is controlled by another domain).
(4) 	yes	Effect e is shared and controlled by D , cause c is internal to D .
(5) 	no	Effect e is shared and observed by D (e is controlled by another domain), and cause c is internal to D . This case is not admissible because e can only be controlled by one domain, and c is internal to D , so c cannot cause e which contradicts that $(g) : c \rightsquigarrow e$ is a domain property of D .
(6) 	yes	Both effect e and cause c are shared and controlled by D .
(7) 	yes	Effect e is shared and controlled by D , and cause c is shared and observed by D (c is controlled by another domain).
(8) 	no	Effect e is shared and observed by D (e is controlled by another domain), and cause c is shared and controlled by D . This case is not admissible because: a. if D shares e and c with different domains, then this is not possible (similar argument to (5)); b. if D shares e and c with the same domain, then the case is similar to (7), except that e and c swap places with each other.
(9) 	no	Both effect e and cause c are shared and observed (e and c are controlled by other domain(s)). This case is not admissible because if D only observes e and c , then $(g) : c \rightsquigarrow e$ is not a domain property of D .

Tab. 5.1: Analysis of all possible cases for the ETC rule class

In the table, the *case* column represents all possible topological relationships among e , c , and D in a problem diagram. However, not all cases are compatible with the fact that $(g) : c \rightsquigarrow e$ must be a property of D , which is indicated under the *admissible* column. The *description / explanation* column gives a brief description of the relationships, with some explanation for those incompatible cases, i.e., (5), (8) and (9).

To summarise Table 5.1, all the cases that are admissible have the following in common: e is either internal to D or shared and controlled by D . This is a condition for application of this rule class.

Defining Rules Based on Problem Topology

We define our set of ETC rules based on the above admissible cases. Here we present and formalise one of these rules and apply it to the simple heating control example. The other rules can be similarly defined after necessary changes. For the complete set of ETC rules, please refer to the Appendix B. We will apply some of those rules to the case studies in the next chapter.

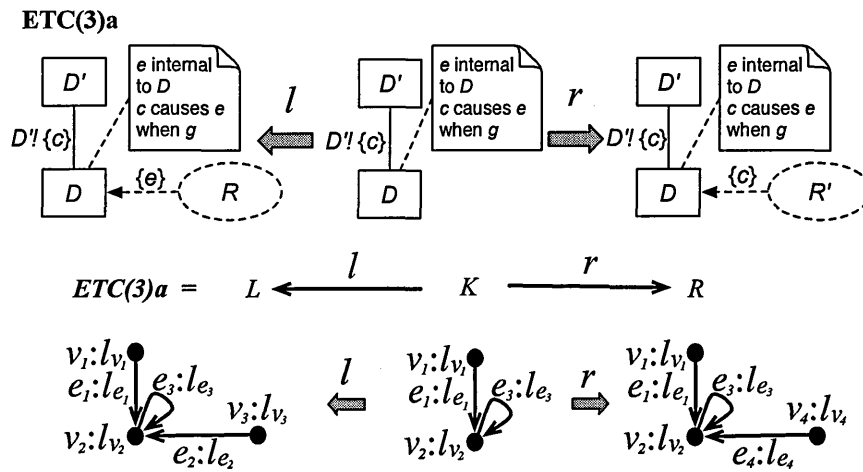


Fig. 5.17: Rule ETC(3)a, derived from admissible case (3) in Table 5.1

We adopt the following convention in uniquely identifying the rules: the name consists of three parts; the first part is the sub-rule acronym in capital letters; the second part is the case number of problem topology to which the rule applies; the third part is either the letter “a”, when the requirement is a constraint, or the letter “b”, when the requirement is a reference. For instance, the rule in Figure 5.17 is named “ETC(3)a”: where “ETC” indicates that it is an effect-to-cause rule; “(3)” indicates that it is derived from case (3) (in Table 5.1); and “a” indicates that the requirement is a constraint.

In Figure 5.17, rule ETC(3)a is derived from admissible case (3) because the prob-

lem topology of e , c and D (including control and causal relationships) exactly matches that of case (3) in Table 5.1. Under this rule, domains D' , D , shared phenomenon c and internal phenomenon are not changed; while requirement R and the requirement phenomenon e are replaced with requirement R' and the requirement phenomenon c . The description in the dog-eared box remains part of the domain properties of D .

More formally, rule ETC(3)a can be represented as a graph production rule (the bottom diagram in Figure 5.17), where the application conditions are:

1. the type of v_1 and v_2 is either *Machine*, *Designed* or *Given*;
2. the type of v_3 and v_4 is *Requirement*;
3. the description of v_3 includes occurrence(s) of "... e occurs ...";
4. the description of v_4 is derived from that of v_3 by replacing "... e occurs ..." with "... c occurs when g holds ...";
5. the description of e_3 must contain statements of causality, e.g., there should be a statement like " e internal to D , c causes e when g " as part of domain D 's properties (internal phenomena as reflexive arc e_3);
6. the time elapsed between the occurrence of the cause c and that of the effect e is short enough to be ignored (if the requirement statement R explicitly or implicitly sets time limits for its satisfaction, e.g., real-time systems, then timed causality should be used, i.e., in the form of $c \xrightarrow{\Delta T} e$, where ΔT should be within those limits).

The justification of the above rule (which is similar to all cause and effect rules) is as follows:

- statements in R on phenomena other than event e are untouched by this rule, and remain the same in the derived requirement R' ; thus all the constraints on such phenomena are the same in both R and R' and are satisfied under the same conditions;
- because $(g) : c \rightsquigarrow e$, a statement of sufficient causality, is part of the behaviour of D , occurrences of “... e occurs ...” are always the effect of “... c occurs when g holds ...”; thus behaviours that satisfy R will also satisfy R' , and vice versa;
- that the timing of the system is not compromised by focusing on event c instead of e means that care has already been taken in considering the time elapsed between the occurrence of c and that of its effect e , so that replacing R with R' does not affect the order of event occurrence in behaviours.

Applying Rule ETC(3)a to an Example

We can now demonstrate in Figure 5.18 how the above rule, ETC(3)a, is applied in step 1 of problem progression in the heating control example:

In the top part of Figure 5.18, l and r represent two injective mapping functions which ensure that domains D' , D , shared phenomenon c (including the control information) and internal phenomena (represented by the dog-eared box with “ e internal to D , c causes e when g ” remain invariant during the application of this rule.

Firstly, rule ETC(3)a can be applied to the bottom-left problem diagram in Figure 5.18 because there exists an injective mapping function m such that:

- $m(D') = \text{Heating controller}$;
- $m(D'!\{c\}) = HC!\{on, off\}$ (at the event level, $m(c) = on$ and $m(c) = off$);

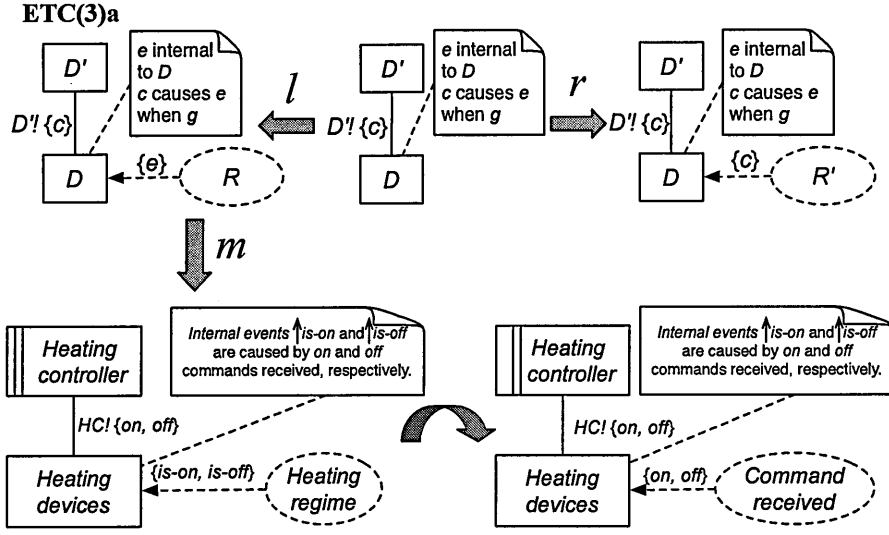


Fig. 5.18: Applying effect-to-cause rule ETC(3)a to the heating control problem diagram

- $m(D) = \text{Heating devices}$;
- $m(\{e\}) = \{is - on, is - off\}$ (at the event level, $m(e) = is - on$ and $m(e) = is - off$);
- $m(R) = \text{Heating regime}$;
- the function m also matches the dog-eared box with "e internal to D, c causes e when g" to the dog-eared box with "Internal events $\uparrow is - on$ and $\uparrow is - off$ are caused by on and off commands received, respectively".

Secondly, in addition to the match m , the application conditions of rule ETC(3)a are met as follows:

- the type of *Heating controller* is *Machine*; the type of *Heating devices* is *Given*;
- the type of *Heating regime* and *Command received* is *Requirement*;

- the description of *Heating regime* is “the heating devices should be switched on [i.e., *is – on* state] at 8:45 am and switched off [i.e., *is – off* state] at 4:45 pm every day”, which matches the pattern “... switched on [*e* occurs] ... switched off [*e* occurs] ...”;
- the description of *Command received* is derived from that of *Heating regime* - “the heating devices should receive on pulse command [replacing its effect *is – on* state] at 8:45 am and off pulse command [replacing its effect *is – off* state] at 4:45 pm everyday”, which matches the pattern “... on pulse command [*c* occurs when *g* holds] ... off pulse command [*c* occurs when *g* holds] ...”, where *g* (domain property of *Heating devices*) is trivially true;
- part of the domain properties of *Heating devices* expresses causality, i.e., its internal phenomenon “Internal events $\uparrow is - on$ and $\uparrow is - off$ can be caused by on [*pulse*] and off [*pulse*] commands received”;
- the time elapsed between the occurrence of “... receive on pulse command ... and off pulse command ...”, and that of their effects “... should be switched on [i.e., *is – on* state] ... and switched off [i.e., *is – off* state]” is short enough to be ignored.

Finally, the bottom-right part of Figure 5.18 (the transformed problem diagram) is derived by following the production rule in the top part of the figure:

- since $\{e\}$ is replaced by $\{c\}$ in the production rule, $\{is - on, is - off\}$ is replaced by $\{on, off\}$;
- since *R* is replaced by *R'* in the production rule, *Heating regime* is replaced by

Command received whose description is derived by following the application conditions of rule ETC(3)a;

- since other graphical elements are untouched by the production rule, other parts of the problem diagram remain invariant.

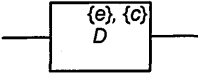
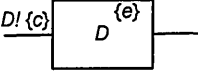
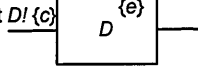
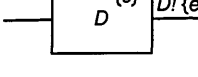
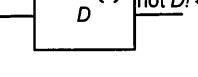
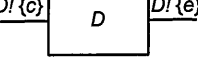

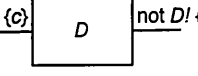
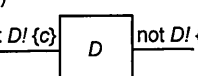
5.4.1.2 The Cause-to-Effect (CTE) Rule Class

Let us look at the cause-to-effect rule class. Similarly, under this rule class, the requirement is rewritten so that any occurrence of a cause, say event “...*c* occurs...”, is replaced by an occurrence of its guarded effect, say “*e* occurs and *g* holds”. These rules are defined based on causal behaviours of domain *D* at the event level, rather than phenomena which are represented as sets of events in a problem diagram.

Analysing Different Cases of Problem Topology

Because events *c* and *e* can be internal to *D*, shared and controlled, or shared and observed by domain *D*, there are nine different cases in which the CTE rule class may apply. Like our analysis of the ETC rules, we consider each of them and discard those cases that are not admissible. Table 5.2 summaries the result of the analysis:

There are three cases that are not admissible in Table 5.2. The analysis and the argument why they are not admissible are similar to those of the ETC rules, thus omitted. Again as a rule of thumb, in all the cases that are admissible, *e* is either internal to *D* or shared and controlled by *D*. This is a condition for application of this rule class.

Case	Admissible (yes / no)	Description / Explanation
(1) 	yes	Both effect e and cause c are internal to D .
(2) 	yes	Effect e is internal to D , cause c is shared and controlled by D .
(3) 	yes	Effect e is internal to D , cause c is shared and observed by D (c is controlled by another domain).
(4) 	yes	Effect e is shared and controlled by D , cause c is internal to D .
(5) 	no	Effect e is shared and observed by D (e is controlled by another domain), and cause c is internal to D . This case is not admissible because e can only be controlled by one domain, and c is internal to D , so c cannot cause e which contradicts that $(g) : c \rightsquigarrow e$ is a domain property of D .
(6) 	yes	Both effect e and cause c are shared and controlled by D .
(7) 	yes	Effect e is shared and controlled by D , and cause c is shared and observed by D (c is controlled by another domain).
(8) 	no	Effect e is shared and observed by D (e is controlled by another domain), and cause c is shared and controlled by D . This case is not admissible because: a. if D shares e and c with different domains, then this is not possible (similar argument to (5)); b. if D shares e and c with the same domain, then the case is the same as (7), except that e and c swap places.
(9) 	no	Both effect e and cause c are shared and observed (e and c are controlled by other domain(s)). This case is not admissible because if D only observes e and c , then $(g) : c \rightsquigarrow e$ is not a domain property of D .

Tab. 5.2: Analysis of all possible cases for the CTE rule class

Defining Rules Based on Problem Topology

We define our set of CTE rules based on the admissible cases in Table 5.2. The representation and justification of these rules are similar to those of ETC rules, thus omitted here. Please refer to the Appendix B for details of these rules.

5.4.2 The Changing Viewpoint Rule Class

We call the second progression rule class *Changing Viewpoint*. Rules in this class generate a new requirement statement based on the differing perspectives of domains sharing an event: switching from the perspective of a domain controlling the event to that of a domain observing the event, and vice versa. We specialise this rule class into two sub-classes, namely the *observe-to-issue* (OTI) class and *issue-to-observe* (ITO) class.

5.4.2.1 The Observe-to-Issue (OTI) Rule Class

Let us look at the the observe-to-issue rule class first. Under this rule class, the requirement is rewritten so that any description of a shared event, say *ev*, is switched from the viewpoint of its “observer” domain, say *D'*, to that of its “controller” domain, say *D*. To capture patterns of control in natural language descriptions, we denote by “... *D issues ev* ...” any part of a requirement statement that implies an occurrence of event *ev* controlled by *D*, and by “... *D' observes ev* ...” any part of a requirement statement that implies an occurrence of event *ev* observed by *D'*.

Defining Rules Based on Problem Topology

Unlike the *reducing through cause and effect*, this rule class focuses on two domains *D* and *D'* and the event they share *ev*. So there is only one admissible case in terms of

topological relationships among D , D' and ev that we need to consider. Therefore, we omit the case number in naming them. From this admissible case, we derive two rules, depending on whether the requirement is expressed in terms of a constraint on ev or a reference to ev , as shown in Figure 5.19:

- event ev is constrained by the requirements, thus this rule is called OTIa;
- event ev is referred to by the requirements, thus this rule is called OTIb.

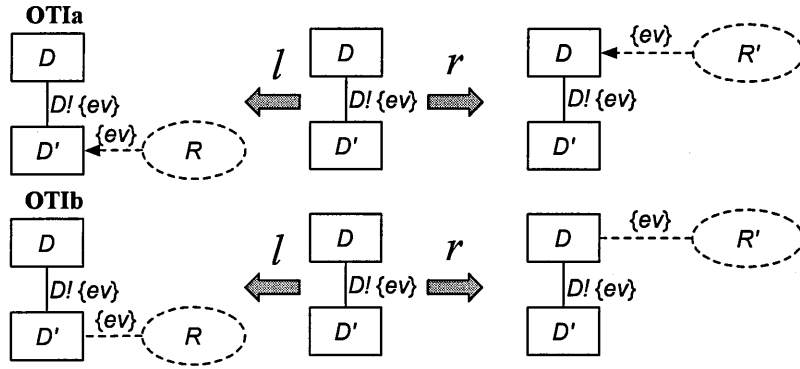


Fig. 5.19: Observe-to-issue rules OTIa and OTIb

In Figure 5.19, OTIa represents the case where an event ev is shared between domains D and D' , and controlled by D . Under this rule, domains D , D' and their shared event ev are not changed; while the requirement R expressed from the viewpoint of the observer D' and its constraint on ev are removed, which is compensated by the addition of a new requirement R' expressed from the viewpoint of the controller D and its constraint on ev which is attached to D .

More formally, OTIa can be represented as a graph production rule (the bottom diagram in Figure 5.20), where the application conditions of the rule are:

1. the type of v_1 and v_2 is either *Machine*, *Designed* or *Given*;

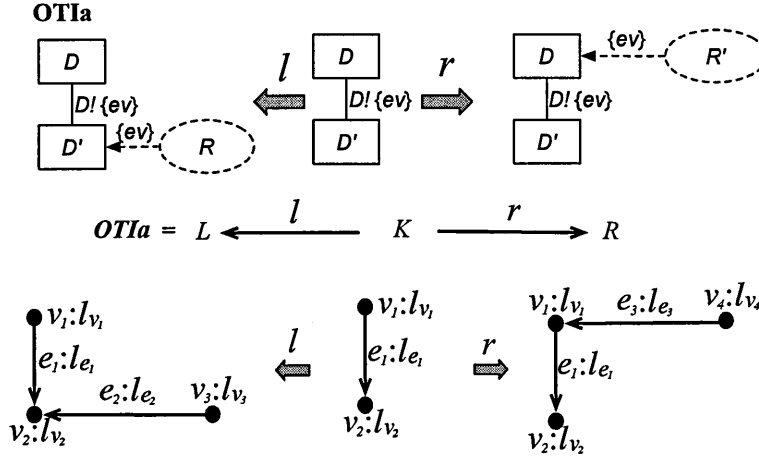


Fig. 5.20: Observe-to-issue rule OTIa represented as a graph production rule

2. the type of v_3 and v_4 is *Requirement*;
3. the description of v_3 includes occurrence(s) of "... D' observes ev ...", which is expressed from the viewpoint of v_2 ;
4. the description of v_4 is derived from that of v_3 by replacing each "... D' observes ev ..." with "... D issues ev ...", which is expressed from the viewpoint of v_1 .

The justification of the above rule (which is similar to all changing viewpoint rules) is as follows:

- statements in R on phenomena other than event ev are untouched by this rule, and remain the same in the derived requirement; thus all constraints on such phenomena remain the same in both R and R' , and are satisfied under the same conditions;
- because ev is shared between D and D' , the occurrence of ev that D' observes is exactly the same as that D issues (controls); thus behaviours satisfying R also satisfy R' , and vice versa.

Applying Rule OTIa to an Example

We can now demonstrate in Figure 5.21 how the above rule, OTIa, is applied in step 2 of problem progression in the heating example:

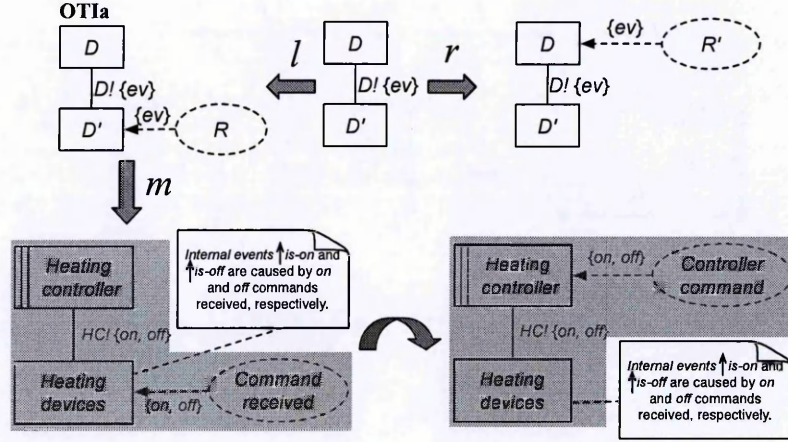


Fig. 5.21: Applying the observe-to-issue rule OTIa to the heating control problem diagram

In the top part of Figure 5.21, l and r represent two injective mapping functions which ensure that domains D' , D , shared phenomenon ev (including the control information) remain invariant during the application of this rule.

Firstly, rule OTIa can be applied to the bottom-left problem diagram in Figure 5.21 because there exists an injective mapping function m such that:

- $m(D') = \text{Heating devices}$;
- $m(D!\{ev\}) = HC!\{on, off\}$ (at the event level, $m(ev) = on$ and $m(ev) = off$);
- $m(D) = \text{Heating controller}$;
- $m(\{ev\}) = \{on, off\}$ (at the event level, $m(ev) = on$ and $m(ev) = off$);
- $m(R) = \text{Command received}$.

Note that in Figure 5.21, the darkened area on the left of the bent arrow indicates those parts that match the left-hand side of the rule - the images of the match m function; the darkened area on the right of the bent arrow indicates those parts that have been derived by following the rule above, which imitates the right-hand side of the rule. Throughout this thesis, we follow this convention when presenting an application of a rule.

Secondly, in addition to the match m , the application conditions of rule OT1a are met as follows:

- the type of *Heating controller* is *Machine*; the type of *Heating devices* is *Given*;
- the type of *Command received* and *Controller command* is *Requirement*;
- the description of *Command received* includes statement: “the heating devices should receive on pulse command at 8:45 am and off pulse command at 4:45 pm everyday”, which is expressed from the viewpoint of the *Heating devices* - the observer domain of *on* and *off*, which matches the pattern “... heating devices should receive on ... and off ... [D' observes ev] ...”;
- the description of *Controller command* is derived from that of *Command received*: “the heating controller should issue the *on* command at 8:45 am and the *off* command at 4:45 pm every day”, which is expressed from the viewpoint of the *Heating controller* - the controller domain of *on* and *off*, which matches the pattern “... heating controller should issue the *on* ... and the *off* ... [D issues ev] ...”.

Finally, the bottom-right part of Figure 5.21 (the transformed problem diagram) is derived by following the production rule in the top part of the figure:

- since requirement R and its constraint on phenomenon ev are removed from domain D' in the production rule, requirement *Command received* and its constraint on phenomena $\{on, off\}$ are removed from *Heating devices* in the problem diagram;
- since R' and its constraint on phenomenon ev are added to domain D in the production rule, *Controller command* and its constraint on phenomena $\{on, off\}$ are added to domain *Heating controller* in the problem diagram. The description of *Controller command* is derived by following the application conditions of rule OTIa (shown previously);
- since other graphical elements are untouched by the production rule, other parts of the problem diagram remain invariant.

5.4.2.2 The Issue-to-Observe (ITO) Rule Class

Let us look at the issue-to-observe rule class. Similarly, under this rule class, the requirement is rewritten so that any description of a shared event is switched from the viewpoint of the “controller” domain to that of the “observer” domain. In other words, for domains D , D' and event ev shared by them and controlled by D' , a requirement statement like “... D' issues ev ...” is replaced by “... D observes ev ...”.

Defining Rules Based on Problem Topology

Like the observe-to-issue rule class, there is only one admissible case in terms of topological relationships among D , D' and ev to be considered. Again we omit the case number in naming them. We derive two working rules, depending on whether the requirement is expressed in terms of a constraint on ev or a reference to ev , as shown in

Figure 5.22:

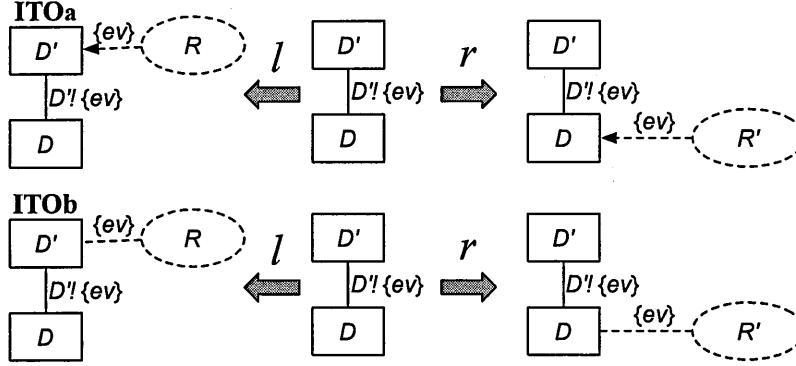


Fig. 5.22: Issue-to-observe rule ITOa and ITOb

- event ev is constrained by the requirements, thus this rule is called ITOa;
- event ev is referred to by the requirements, thus this rule is called ITOb.

The justification of this rule class is similar to that of observe-to-issue rule class, thus omitted. Examples of applying this rule class can be found in the next chapter.

5.4.3 The Removing Domain Rule Class

We call the third progression rule class *Removing Domain*. Rules in this class are used to simplify problem diagrams, allowing us to remove a domain, say domain D' , from consideration in the analysis, as long as corresponding assumptions are explicitly stated in the rewritten requirement, that is, expressed by the following pattern in natural language description: “... *assuming* D' ...”, which is a shorthand for “... *under the assumption that necessary causal relationships exist as part of the domain properties of* D' ...”.

Defining Rules Based on Problem Topology

This rule class focuses on two domains D and D' . Domain D is attached to the requirement R ; while D' is the domain to be removed, as shown in Figure 5.23. Depending on whether the requirement constrains or refers to the event they share or other events that do not belong to D' , there are six possible cases:

- event ev is constrained by the requirement R , and controlled by D , thus this rule is called RD(1)a;
- event ev is referred to by the requirement R , and controlled by D , thus this rule is called RD(1)b;
- event ev is constrained by the requirement R , and controlled by D' , thus this rule is called RD(2)a;
- event ev is referred to by the requirement R , and controlled by D' , thus this rule is called RD(2)b.
- event ev is constrained by the requirement R , and does not belong D' , thus this rule is called RD(3)a;
- event ev is referred to by the requirement R , and does not belong D' , thus this rule is called RD(3)b.

All of the above cases are admissible, from which six rules are derived.

In Figure 5.23, RD(1)a represents the situation where event ev is shared between domains D and D' , and controlled by D . Under this rule, domain D remain unchanged, while domain D' and its constraint on ev are removed away from D , which is compensated by adding a rewritten requirement statement R' and its constraint on ev which is

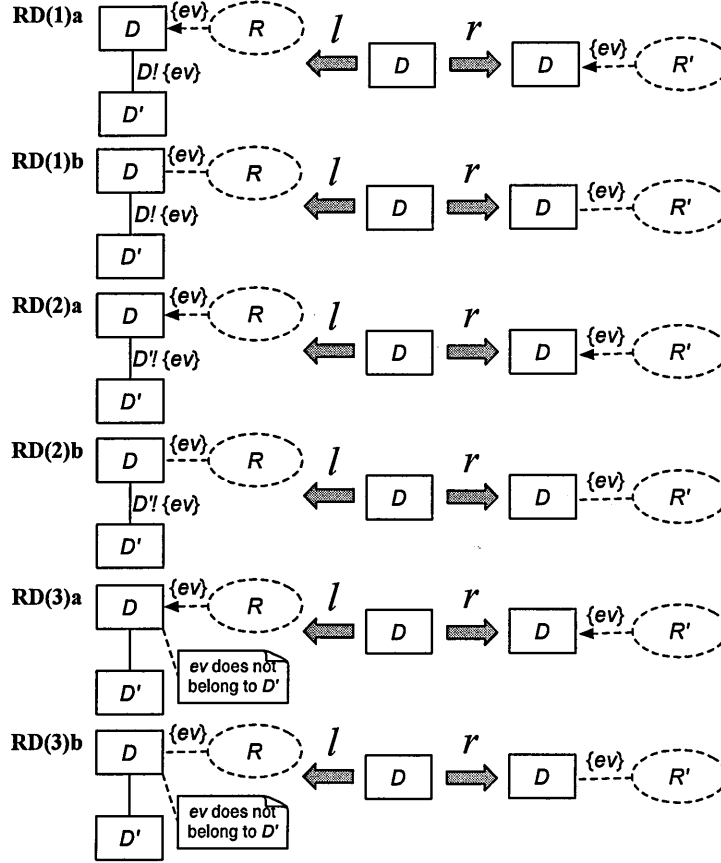


Fig. 5.23: Removing domain rule RD(1)a, RD(1)b, RD(2)a, RD(2)b, RD(3)a and RD(3)b

attached to D . The rewritten requirement statement R' is derived from R , by adding assumptions about the removed domain D' , i.e., in the general form of "... assuming D' , [a repetition of R]".

More formally, RD(1)a can be represented as a graph production rule (the bottom diagram in Figure 5.24), where the application conditions of the rule are:

1. the type of v_1 and v_2 is either *Machine*, *Designed* or *Given*;
2. the type of v_3 and v_4 is *Requirement*;

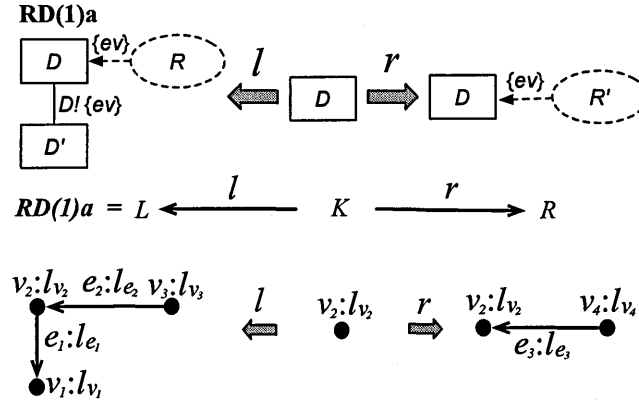


Fig. 5.24: Removing domain rule RD (a) represented as a graph production rule

3. the event that v_3 constrains, i.e., ev is the same event shared between v_1 and v_2 ;
4. with the exception of ev , no more events that belong to v_1 are constrained or referred to by v_3 ; and with the exception of v_2 , no other domain is significant to v_3 .

Note that we only apply this rule when R does not constrain phenomena of D' (except D' 's shared event ev with D), in other words, no more events that belong to D' are constrained by R , therefore, we have the following justification of the rule (which is similar to all removing domain rules):

- statements in R on phenomena other than event ev are untouched by this rule, and remain the same in the derived requirement; since R 's only constraint on domain D' is ev (R may constrain or refer to some internal phenomena that belong to D or D' 's shared phenomena with other domains), removing D' does not touch any phenomena in R , and since R 's constraint on ev is still kept within the rewritten requirement, i.e., R' repeats what is stated in R , thus all constraints or references on such phenomena remain the same in both R and R' .

Applying Rule RD(1)a to an Example

We can now demonstrate in Figure 5.25 how rule RD(1)a is applied in step 3 of problem progression in the heating example:

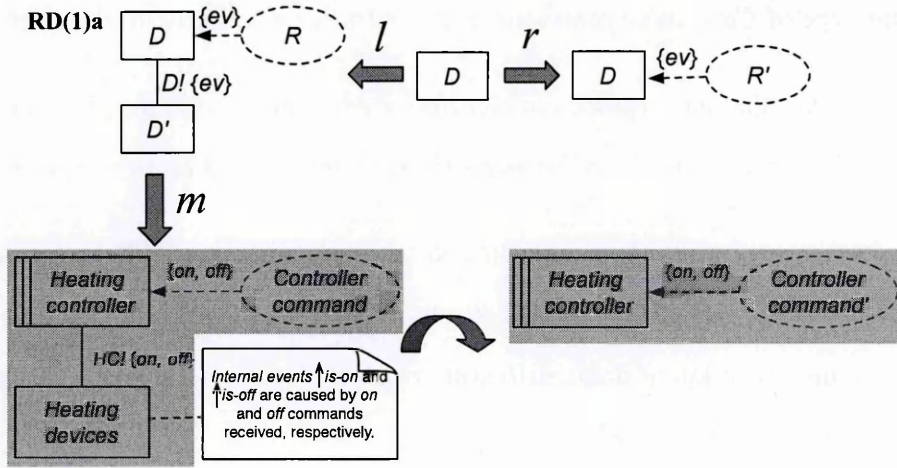


Fig. 5.25: Applying removing domain rule RD(1)a to the heating control problem diagram

In the top part of Figure 5.25, l and r represent two injective mapping functions which ensure that domain D remain invariant during the application of this rule.

Firstly, rule RD(1)a can be applied to the bottom-left problem diagram in Figure 5.25 because there exists an injective mapping function m such that:

- $m(D') = \text{Heating devices}$;
- $m(D!\{ev\}) = HC!\{on, off\}$ (at the event level, $m(ev) = on$ and $m(ev) = off$);
- $m(D) = \text{Heating controller}$;
- $m(\{ev\}) = \{on, off\}$ (at the event level, $m(ev) = on$ and $m(ev) = off$);
- $m(R) = \text{Controller command}$.

Secondly, in addition to the match m , the application conditions of rule RD(1)a are met as follows:

- the type of *Heating devices* is *Given*; the type of *Heating controller* is *Machine*;
- the type of *Controller command* and *Controller command'* is *Requirement*;
- the events that the requirement *Controller command* constrains, i.e., *on* and *off* are the same events shared between *Heating devices* and *Heating controller*;
- with the exception of *on* and *off*, no more phenomena that belong to domain *Heating devices* are constrained or referred to by the *Controller command*; and with the exception of domain *Heating controller*, no other domain is significant to *Controller command*.

Finally, the bottom-right part of Figure 5.25 (the transformed problem diagram) is derived by following the production rule in the top part of the figure:

- since requirement R and its constraint on phenomenon ev are removed from domain D in the production rule, requirement *Controller command* and its constraint on phenomena *on* and *off* are removed from *Heating controller* in the problem diagram;
- since R' and its constraint on phenomenon ev are added to domain D in the production rule, *Controller command'* and its constraint on phenomena *on* and *off* are added to domain *Heating controller* in the problem diagram. The description of *Controller command'* is derived by following the application condition of rule RD(1)a: "... assuming the proper operation of *Heating devices*, the heating controller should issue the *on* command at 8:45 am and the *off* command at 4:45

pm every day”, which matches the pattern “... assuming the proper operation of Heating devices [assuming D'] ... [a repetition of Controller command]”;

- since the dog-eared box is part of *Heating devices*’ domain properties, thus it should be removed when domain *Heating devices* is removed.

5.5 Discussion on Heuristics for Applying the Transformation Rules

In previous sections, we have defined three classes of graph production rules that aim at transforming problem diagrams with arbitrary problem topologies. For example, the cause-to-effect rule class and effect-to-cause rule class cover all possible cases.

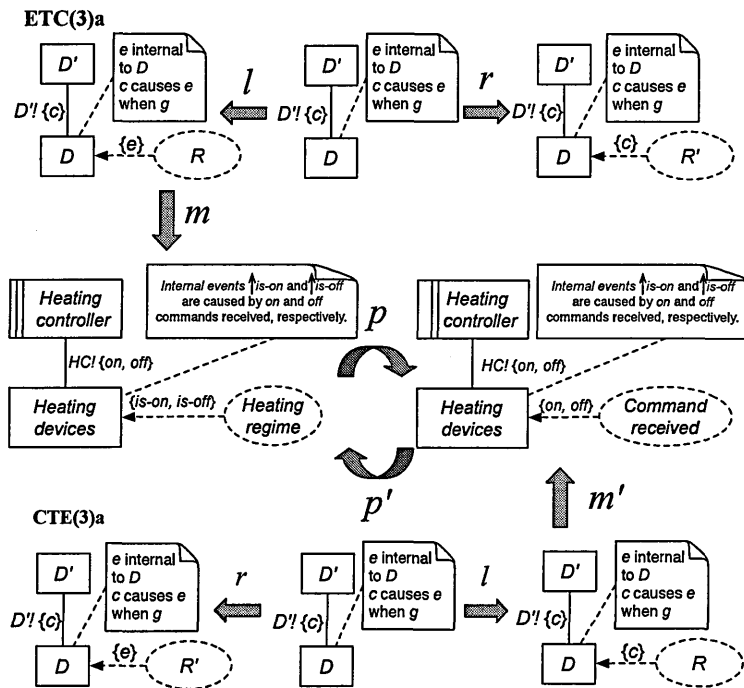


Fig. 5.26: An example of applying effect-to-cause rule ETC(3)a or cause-to-effect rule CTE(3)a to the same problem diagram

Let us investigate what is needed to achieve the goal of problem progression. Let

us take the problem diagram in Figure 5.18 for example, which we recall here in Figure 5.26.

In Figure 5.26, we can find at least two graph production rules that match the same problem diagram (the CTE(3)a rule has been flipped horizontally for the match). If we apply the rules randomly, say rule CTE(3)a, we may end up with an undesired problem diagram (which is the problem diagram on the left-hand side of the bent arrows) after graph transformation p' . Without any heuristics, this kind of undesired transformation can not be prevented.

There is one heuristic that can help us progress problems: problem progression is about transforming problem diagrams in a way that only specification phenomena are described, in other words, we should aim at “moving (the requirement) closer to the machine”. With this heuristic, we should chose rule ETC(3)a, and arrive at the right-hand side of the bent arrows after graph transformation p , instead of p' . The case studies in the next chapter will be based on this heuristic.

We have also defined that our progression rules have to be matched injectively before they can be applied in problem progression. This is an important rule application condition that aims at guaranteeing the convergence of graph transformation process: the formal works by Habel *et al.* [57] have proved that there are many theoretical advantages of injective matching of production rules in graph transformation - the transformation is more likely to terminate and different paths of graph transformation are more likely to converge. Their results provide a formal basis for mechanising our techniques, thus a promising direction for future work.

5.6 Summary

In this chapter, we have introduced a working definition of causality that focuses on cause-and-effect relationships between events. We have given a taxonomy of causality that aims at dealing with more complex domain properties for the purpose of problem progression. For example, conditional causality and timed causality allow us to deal with more elaborate problem descriptions; likewise, biddable causality allows us to express the expected behaviour of a biddable domain.

We have defined a set of causality-based rules for problem progression and illustrated how they can be used for manipulating problem diagrams - problem progression based on these rules can be formalised as graph transformation based on graph production rules. The purpose of this semi-formal approach is not to achieve a complete formalisation of problem progression but to extend the applicability of problem progression based on these rules. The reason for adopting a semi-formal approach rather than a fully formal one is because of the informal nature of problem analysis in early RE: customer requirements start with informal descriptions usually in natural language, so a completely formal treatment is not feasible in the general case; descriptions of complex domain behaviours (e.g., those involving human behaviours) are often too rich to be usefully described by formal models for problem progression. Examples in this chapter have shown that the matching of a rule to part of a problem diagram not only relies on the matching of graphical structures, but also involves finding and matching a fixed pattern of informal expressions in requirement and domain descriptions.

In this chapter, we have applied our causality-based rules for problem progression in a very simple example - the automatic heating control problem. We have demonstrated that the derived specification of the *Heating controller*, i.e., description of

Controller command does indeed satisfy the original requirement *Heating regime* because our causality-based rules can guarantee that the graphical transformation is performed in a solution-preserving way. The simplicity of the problem allows us to have thorough analysis and presentation of our techniques. In order to evaluate the applicability or scalability of our progression rules in a more realistic setting, we will apply our techniques to more complex case studies in the next chapter.

6. CASE STUDIES

This chapter applies the rules in the previous chapter to two case studies adapted from the literature. The first one is the problem of developing software for a point-of-sale (POS) system to help cashiers process purchases in a retail shop environment, which we have also used in Chapter 5. The second one is a package router problem where a computer is required to control the routing of packages to their proper destination bins based on their delivery addresses.

6.1 *The Point-of-Sale (POS) Problem*

Point-of-sale systems are a popular subject for case studies in software engineering, such as in teaching object-orientation and the unified process [104], in industrial experience reports [13], and in software testing [49].

In this case study we assume the following problem statement [128]:

“We consider the development of a point-of-sale (POS) system for a shop. The new POS software system is to be used to process all sales within the shop. The system is to include a controller, to be designed, and some hardware, purchased from a third party. The new POS hardware includes a barcode reader, a credit card reader, a keyboard and display, and a cash drawer.”

The problem is to develop software for the POS system so that cashiers can help customers pay for items they wish to purchase before leaving the shop with a valid receipt.

Figure 6.1 shows the problem diagram.

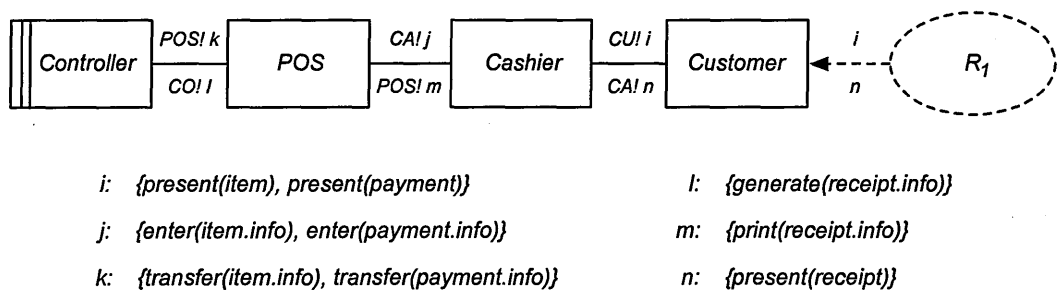


Fig. 6.1: The POS problem diagram

Table 6.1 shows the identified domains and their descriptions.

Name	Description
Customer	A person who wants to buy an item from the shop.
Cashier	A shop employee who is authorised to perform sales.
POS	The new POS hardware which includes a barcode reader, a credit card reader, a keyboard and display, and a cash drawer.
Controller (machine)	The solution to be designed.

Tab. 6.1: Domains and their descriptions

Table 6.2 shows problem phenomena and their designations.

We will progress the requirement through to the specification, that is, repeatedly transform it until the requirement is expressed only in terms of the specification phenomena. The requirement R_1 is as follows:

Name	Type	Designation
<i>present(item)</i>	event	The exchange of an item between the <i>Customer</i> and the <i>Cashier</i> . This event is initiated and controlled by the <i>Customer</i> .
<i>present(payment)</i>	event	The exchange of a payment between the <i>Customer</i> and the <i>Cashier</i> . This event is initiated and controlled by the <i>Customer</i> .
<i>enter(item.info)</i>	event	The action of the <i>Cashier</i> entering item information into the <i>POS</i> , e.g., scanning the items' barcode using the barcode reader. This event is controlled by the <i>Cashier</i> .
<i>enter(payment.info)</i>	event	The action of the <i>Cashier</i> entering payment information into the <i>POS</i> , e.g., swiping a credit card or manually keying in the amount of cash payment into the <i>POS</i> . This event is controlled by the <i>Cashier</i> .
<i>transfer(item.info)</i>	event	The action of the <i>POS</i> transferring item information to the <i>Controller</i> . This event is controlled by the <i>POS</i> .
<i>transfer(payment.info)</i>	event	The action of the <i>POS</i> transferring payment information to the <i>Controller</i> . This event is controlled by the <i>POS</i> .
<i>generate(receipt.info)</i>	event	The action of the <i>Controller</i> making receipt information available to the <i>POS</i> . This event is controlled by the <i>POS</i> .
<i>print(receipt.info)</i>	event	The action of the <i>POS</i> printing receipt. This event is controlled by the <i>POS</i> .
<i>present(receipt)</i>	event	The exchange of a receipt (including due change if cash payment) between the <i>Customer</i> and the <i>Cashier</i> . This event is controlled by the <i>Cashier</i> .

Tab. 6.2: Phenomena and their designations

R_1 = “When the *Customer* issues a number of *present(item)*, followed by one *present(payment)*, if payment is for the correct amount, then the *Customer* should observe *present(receipt)*.”.

Note that R_1 relates a number of *present(item)*, followed by *present(payment)*, which are referred to by R_1 , and *present(receipt)* which is constrained by R_1 . This association should be achieved by the combined interactions among domains *Customer*, *Cashier*, *POS*, and *Controller*.

6.1.1 First Step of Progression

In the first step, we apply the issue-to-observe rule ITOb twice, and switch from the *Customer* to the *Cashier*, as they shared events $present(item)$ and $present(payment)$. The rule application is shown in Figure 6.2 and results in the rewritten requirement:

R_2 = “When the Cashier observes a number of $present(item)$, followed by one $present(payment)$, if payment is for the correct amount, then the Customer should observe $present(receipt)$.”.

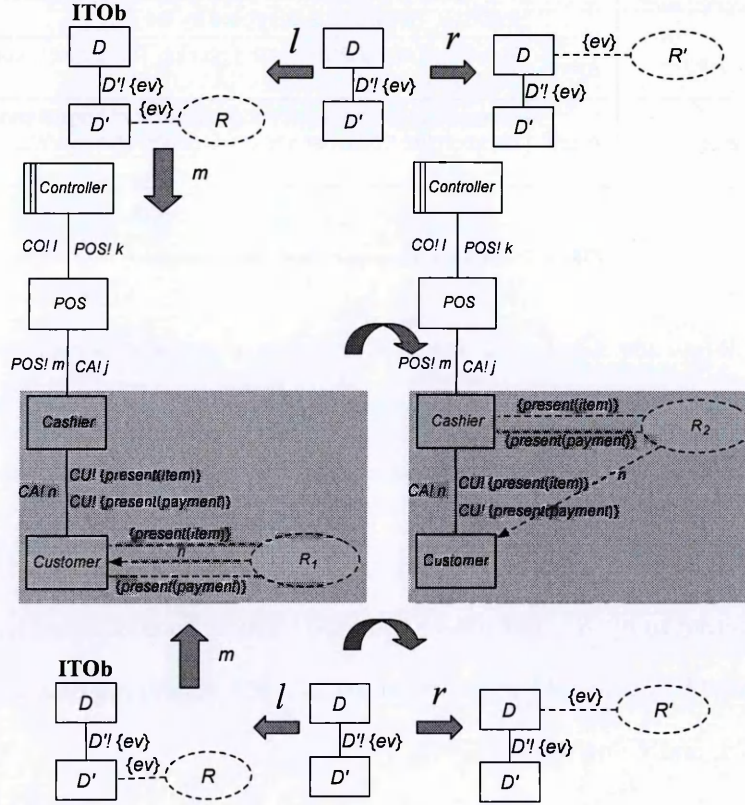


Fig. 6.2: Point-of-sale problem progression step 1: applying the issue-to-observe rule ITOb

6.1.2 Second Step of Progression

In the second step, we apply the observe-to-issue rule OTIa, and switch from the *Customer* to the *Cashier*, as they share event $\text{present}(\text{receipt})$. The rule application is shown in Figure 6.3 and results in the rewritten requirement:

R_3 = “When the *Cashier* observes a number of $\text{present}(\text{item})$, followed by one $\text{present}(\text{payment})$, if payment is for the correct amount, then the *Cashier* should issue $\text{present}(\text{receipt})$ ”.

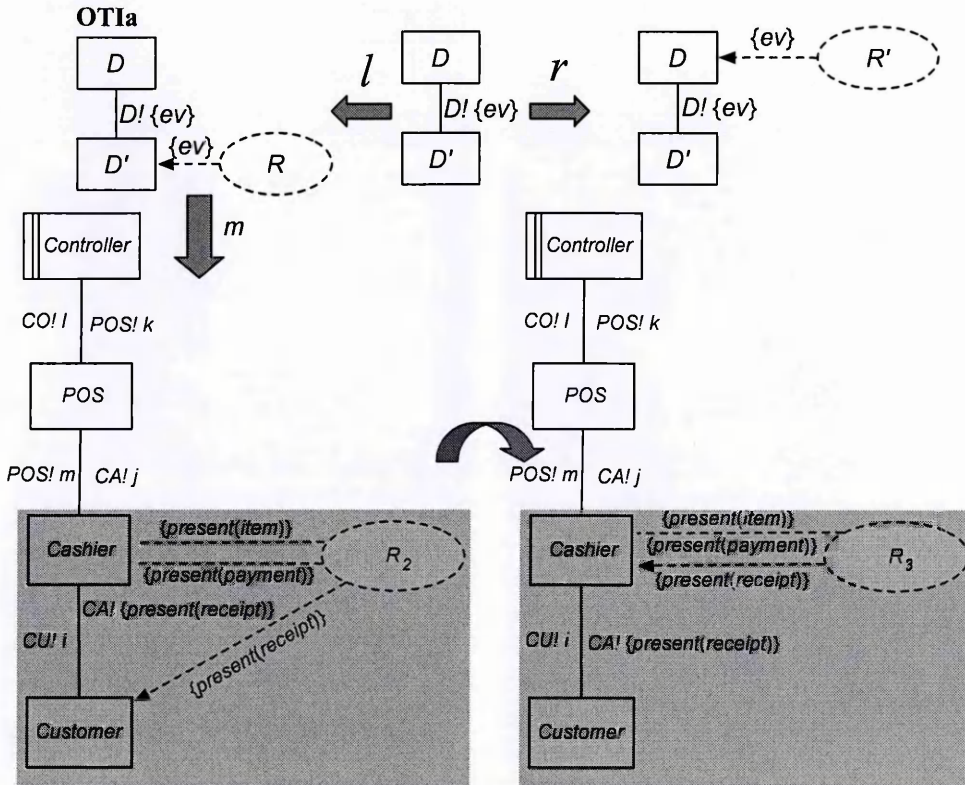


Fig. 6.3: Point-of-sale problem progression step 2: applying the observe-to-issue rule OTIa

6.1.3 Third Step of Progression

In the third step, *Cashier* is expected to have the following domain properties (causal relations)

$present(item) \overset{bidden}{\rightsquigarrow} enter(item.info)$, and

$present(payment) \overset{bidden}{\rightsquigarrow} enter(payment.info)$,

which allow us to apply the cause-to-effect rule CTE(7)b to replace $present(item)$ and $present(payment)$ with $enter(item.info)$ and $enter(payment.info)$ respectively, as shown in Figure 6.4.

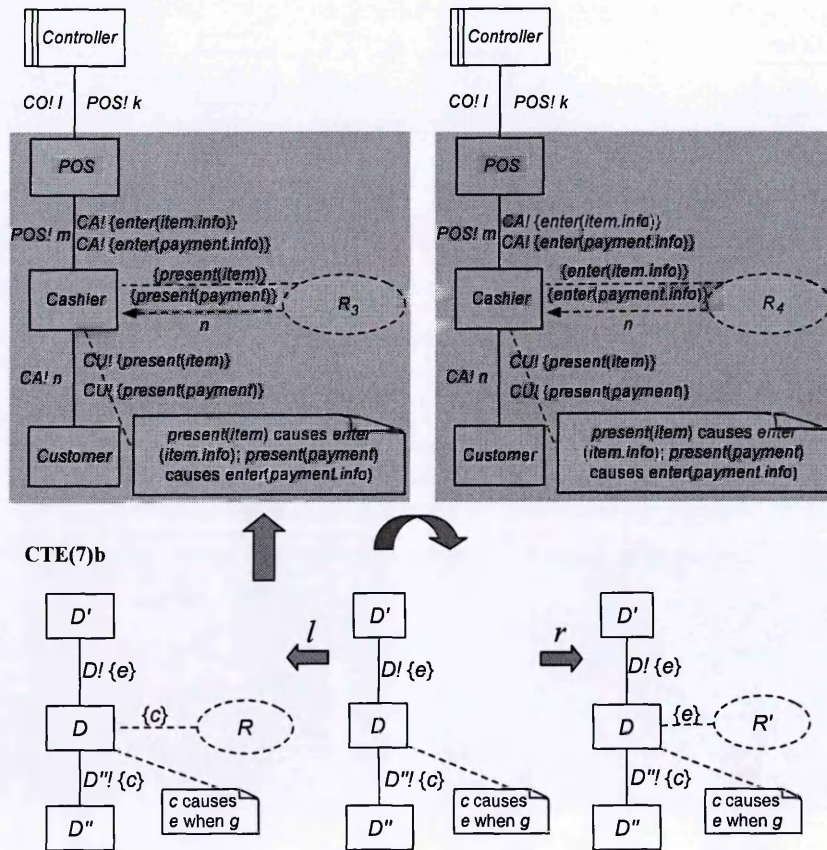


Fig. 6.4: Point-of-sale problem progression step 3: applying the cause-to-effect rule CTE(7)b

By applying the rule, we arrive at the rewritten requirement:

R_4 = “When the Cashier issues a number of $\text{enter}(\text{item.info})$, followed by one $\text{enter}(\text{payment.info})$, if payment is for the correct amount, then the Cashier should issue $\text{present}(\text{receipt})$.”.

6.1.4 Fourth Step of Progression

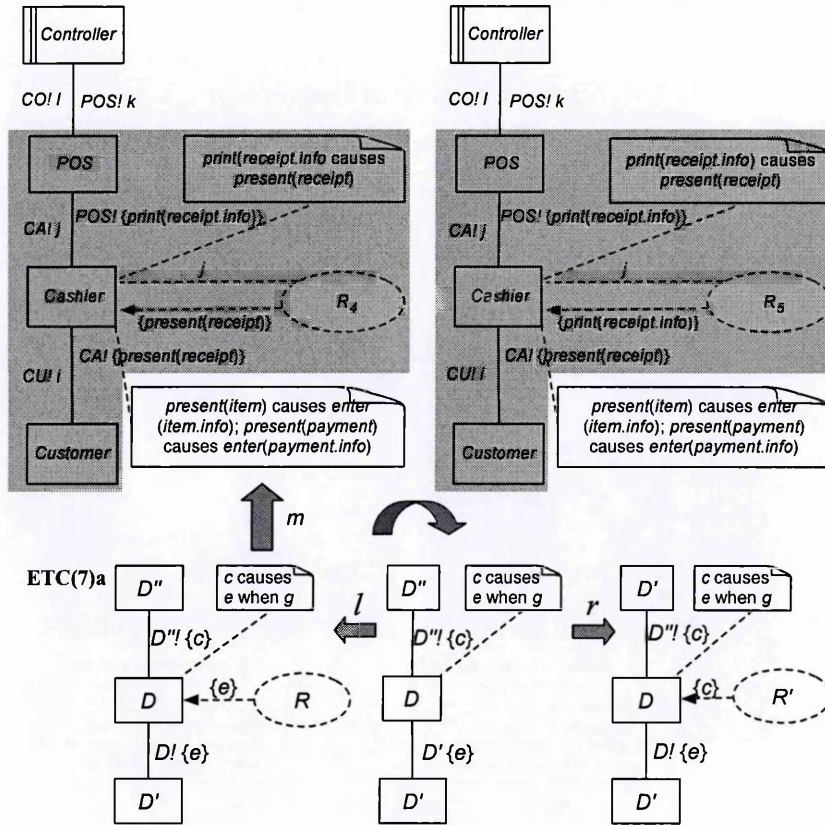


Fig. 6.5: Point-of-sale problem progression step 4: applying the effect-to-cause rule ETC(7)a

In the fourth step, *Cashier* is expected to have the following domain property (causal relation)

$print(receipt.info) \overset{bidden}{\rightsquigarrow} present(receipt),$

which allows us to apply the effect-to-cause rule ETC(7)a to replace $present(receipt)$ with $print(receipt.info)$, as shown in Figure 6.5.

By applying the rule, we arrive at the rewritten requirement:

R_5 = “When the Cashier issues a number of $enter(item.info)$, followed by one $enter(payment.info)$, if payment is for the correct amount, then the Cashier should observe $print(receipt.info)$.”.

6.1.5 Fifth Step of Progression

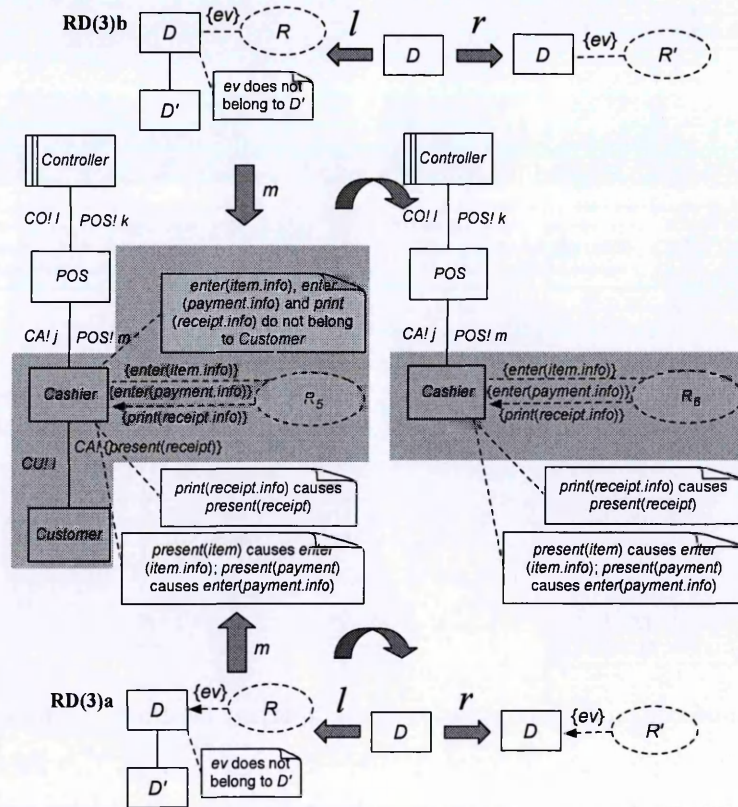


Fig. 6.6: Point-of-sale problem progression step 5: applying the rules RD(3)a and RD(3)b

In the fifth step, by applying the removing domain rules RD(3)a and RD(3)b respectively as shown in Figure 6.6, we arrive at the rewritten requirement:

R_6 = “Assuming *Customer’s* behaviour, when the *Cashier* issues a number of $\text{enter}(\text{item.info})$, followed by one $\text{enter}(\text{payment.info})$, if payment is for the correct amount, then the *Cashier* should observe $\text{print}(\text{receipt.info})$.”.

Application of these rules is justified by the fact that statements in R_5 do not constrain or refer to *Customer’s* behaviour anymore, hence removing the *Customer* domain from the diagram does not touch any phenomena in R_5 .

6.1.6 Sixth Step of Progression

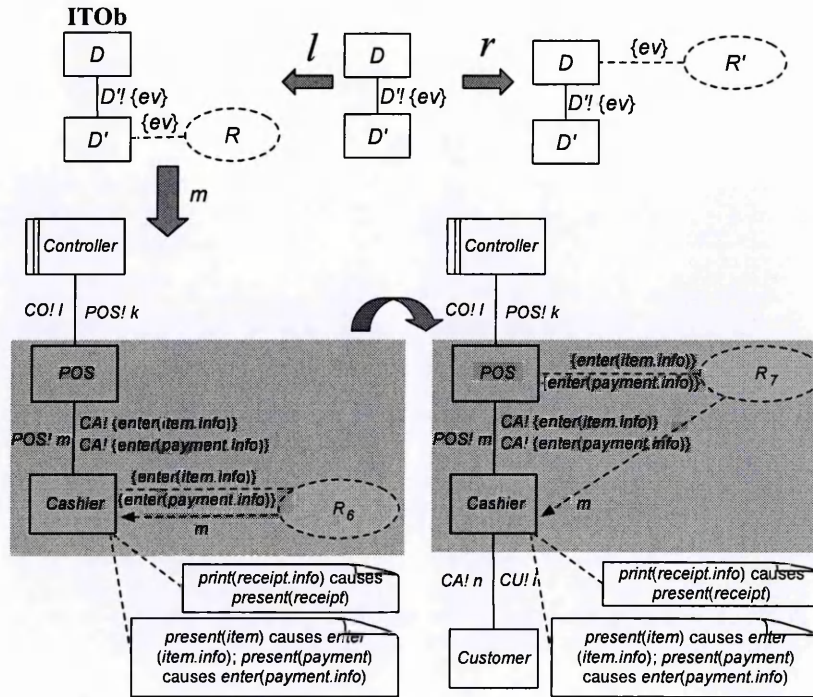


Fig. 6.7: Point-of-sale problem progression step 6: applying the issue-to-observe rule ITOb

In the sixth step, we apply the issue-to-observe rule ITOb and switch from the *Cashier* to the *POS*, as they share event $enter(item.info)$ and $enter(payment.info)$. The rule application is shown in Figure 6.7 and results in the rewritten requirement:

R_7 = “Assuming Customer’s behaviour, when the POS observes a number of $enter(item.info)$, followed by one $enter(payment.info)$, if payment is for the correct amount, then the Cashier should observe $print(receipt.info)$.”.

6.1.7 Seventh Step of Progression

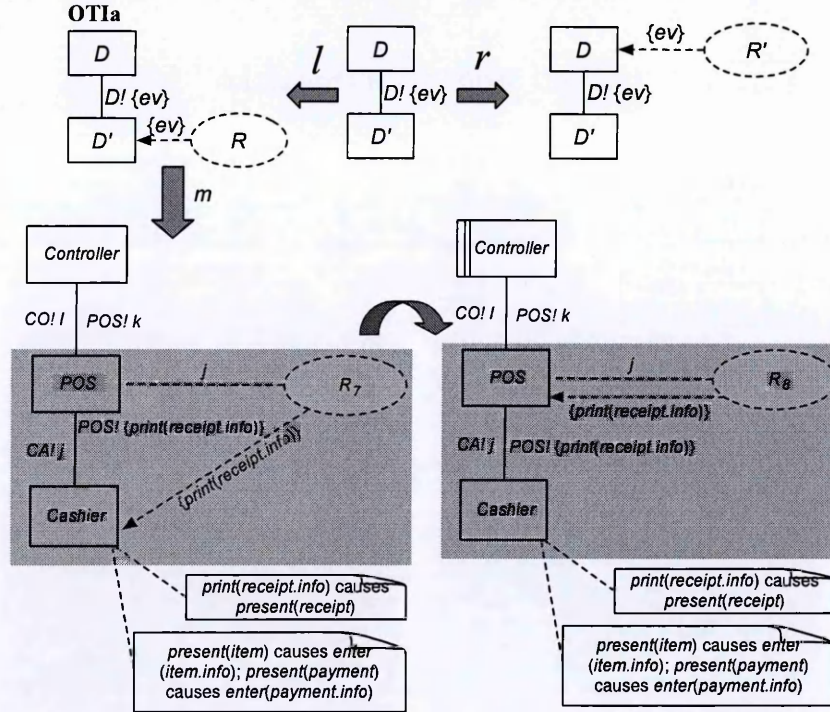


Fig. 6.8: Point-of-sale problem progression step 7: applying the observe-to-issue rule OTIa

In the seventh step, we apply the observe-to-issue rule OTIa and switch from the *Cashier* to the *POS*, as they share event $print(receipt.info)$. The rule application is

shown in Figure 6.8 and results in the rewritten requirement:

R_8 = “Assuming Customer’s behaviour, when the POS observes a number of $\text{enter}(\text{item.info})$, followed by one $\text{enter}(\text{payment.info})$, if payment is for the correct amount, then the POS should issue $\text{print}(\text{receipt.info})$.”.

6.1.8 Eighth Step of Progression

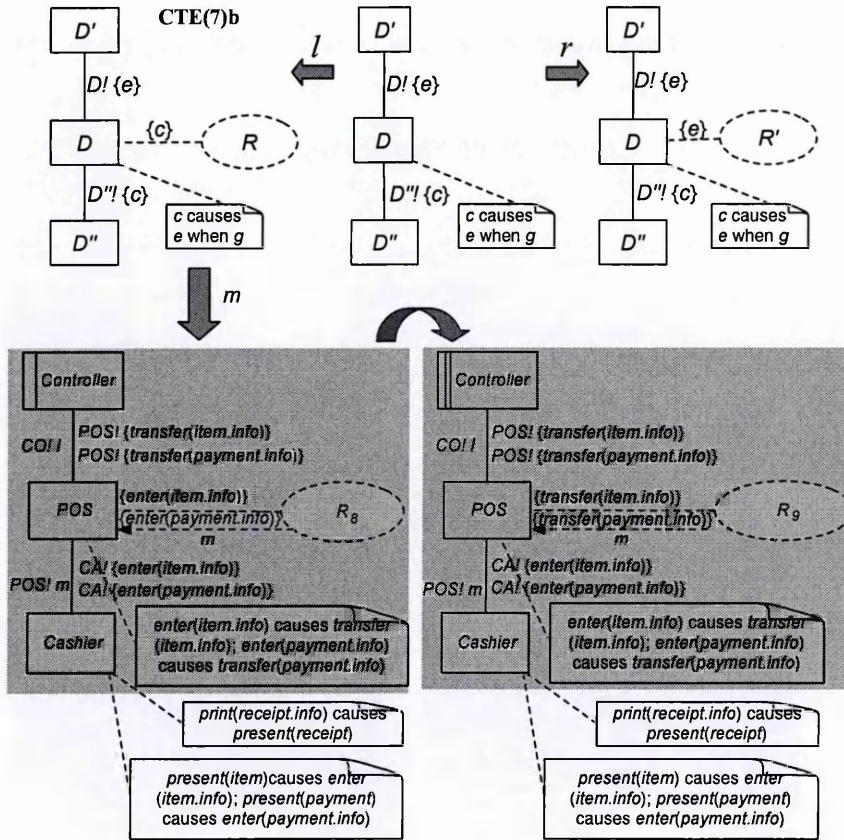


Fig. 6.9: Point-of-sale problem progression step 8: applying the cause-to-effect rule CTE(7)b

In the eighth step, POS has the following domain properties (causal relations)

$\text{enter}(\text{item.info}) \rightsquigarrow \text{transfer}(\text{item.info})$, and

$enter(payment.info) \rightsquigarrow transfer(payment.info),$

which allow us to apply the cause-to-effect rule CTE(7)b to replace $enter(item.info)$ and $enter(payment.info)$ with $transfer(item.info)$ and $transfer(payment.info)$ respectively, as shown in Figure 6.9.

By applying the rule, we arrive at the rewritten requirement:

$R_9 =$ “Assuming Customer’s behaviour, when the POS issues a number of $transfer(item.info)$, followed by one $transfer(payment.info)$, if payment is for the correct amount, then the POS should issue $print(receipt.info)$.”.

6.1.9 Ninth Step of Progression

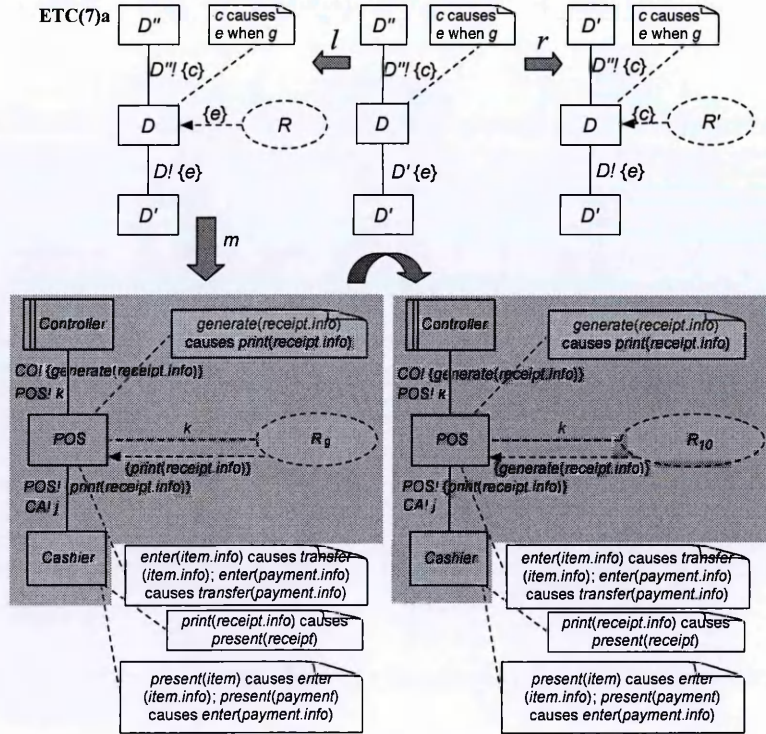


Fig. 6.10: Point-of-sale problem progression step 9: applying the effect-to-cause rule ETC(7)a

In the ninth step, *POS* has the following domain property (causal relation)

$$\text{generate}(\text{receipt.info}) \rightsquigarrow \text{print}(\text{receipt.info}),$$

which allows us to apply the effec-to-cause rule ETC(7)a to replace $\text{print}(\text{receipt.info})$ with $\text{generate}(\text{receipt.info})$, as shown in Figure 6.10.

By applying the rule, we arrive at the rewritten requirement:

R_{10} = “Assuming Customer’s behaviour, when the POS issues a number of $\text{transfer}(\text{item.info})$, followed by one $\text{transfer}(\text{payment.info})$, if payment is for the correct amount, then the POS should observe $\text{print}(\text{receipt.info})$.”.

6.1.10 Tenth Step of Progression

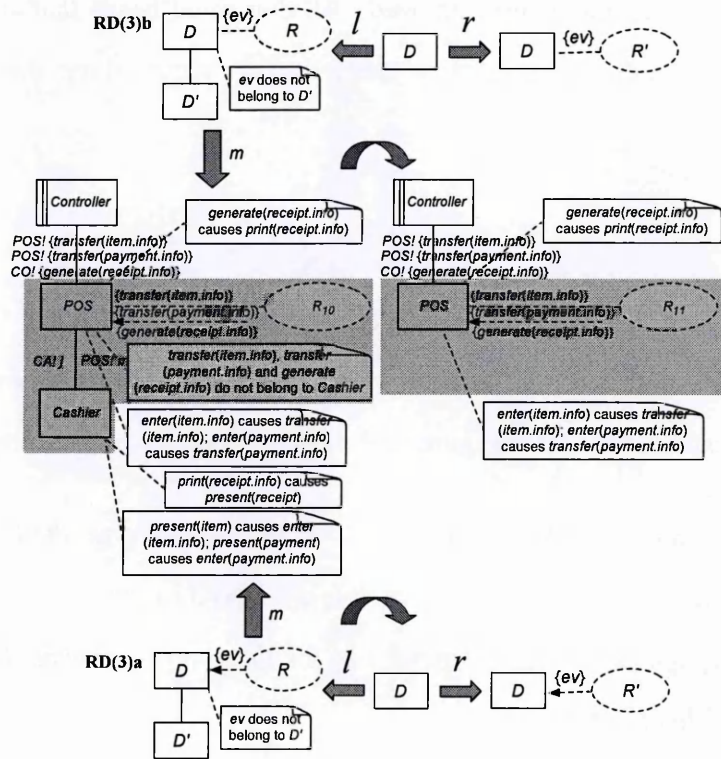


Fig. 6.11: Point-of-sale problem progression step 10: applying the rules RD(3)a and RD(3)b

In the tenth step, by applying the removing domain rules RD(3)a and RD(3)b respectively as shown in Figure 6.11, we arrive at the rewritten requirement:

R₁₁ = “Assuming Customer’s and Cashier’s behaviour, when the POS issues a number of transfer(item.info), followed by one transfer(payment.info), if payment is for the correct amount, then the POS should observe generate(receipt.info).”.

Application of these rules is justified by the fact that statements in R_{10} do not constrain or refer to *Cashier*’s behaviour anymore, hence removing the *Cashier* domain from the diagram does not touch any phenomena in R_{10} . The dog-eared box indicating that events *transfer(item.info)*, *transfer(payment.info)* and *generate(receipt.info)* do not belong to *Cashier* is also removed. All dog-eared boxes that are attached to *Cashier* describe its domain properties, hence they are removed together with the domain.

6.1.11 Eleventh Step of Progression

In the eleventh step, we apply the issue-to-observe rule ITOb and switch from the *POS* to the *Controller*, as they share event *transfer(item.info)* and *transfer(payment.info)*. The rule application is shown in Figure 6.12 and results in the rewritten requirement:

R₁₂ = “Assuming Customer’s and Cashier’s behaviour, when the Controller observes a number of transfer(item.info), followed by one transfer(payment.info), if payment is for the correct amount, then the POS should observe print(receipt.info).”.

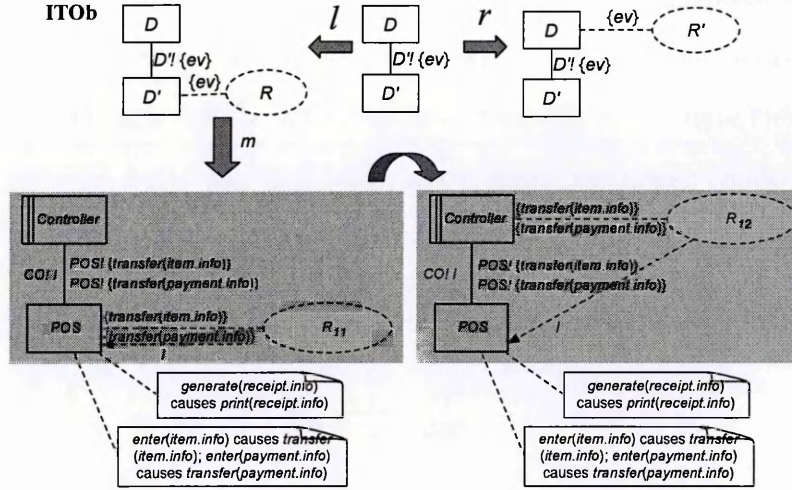


Fig. 6.12: Point-of-sale problem progression step 11: applying the issue-to-observe rule ITOb

6.1.12 Twelfth Step of Progression

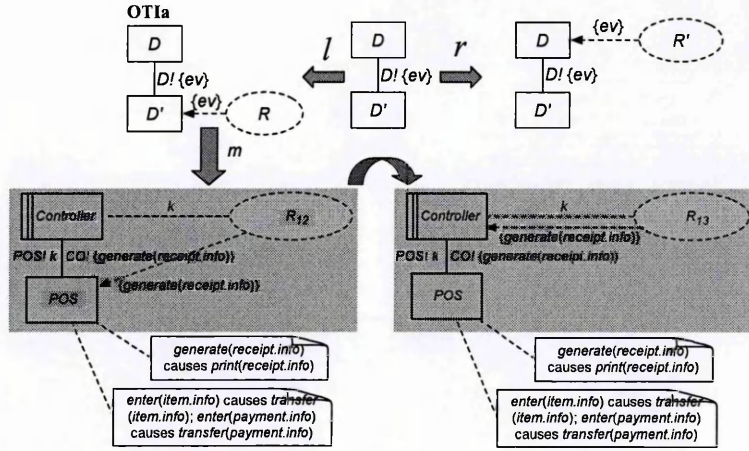


Fig. 6.13: Point-of-sale problem progression step 12: applying the observe-to-issue rule OTIa

In the twelfth step, we apply the observe-to-issue rule OTIa and switch from the *POS* to the *Controller*, as they share event *generate(receipt.info)*. The rule application is shown in Figure 6.13 and results in the rewritten requirement:

R_{13} = “Assuming Customer’s and Cashier’s behaviour, when the Controller observes a number of $\text{transfer}(\text{item.info})$, followed by one $\text{transfer}(\text{payment.info})$, if payment is for the correct amount, then the Controller should issue $\text{print}(\text{receipt.info})$.”.

6.1.13 Thirteenth Step of Progression

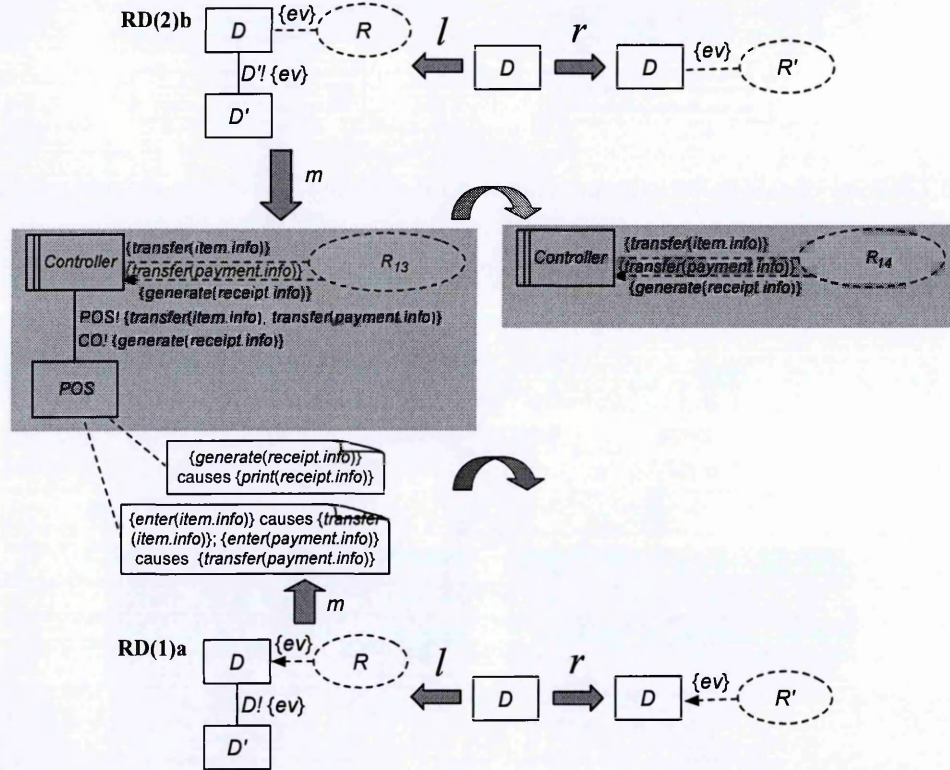


Fig. 6.14: Point-of-sale problem progression step 13: applying the removing domain rules RD(2)b and RD(1)a respectively

In the thirteenth step, by applying the removing domain rule RD(2)b and RD(1)a respectively as shown in Figure 6.14, we arrive at the rewritten requirement:

R_{14} = “Assuming Customer’s, Cashier’s and POS behaviour, when the

Controller observes a number of transfer(item.info), followed by one transfer(payment.info), if payment is for the correct amount, then the Controller should issue print(receipt.info).”.

Notice the following:

- All the dog-eared boxes are part of the domain *POS*, hence they are removed together with the domain.
- The R_{14} expresses a conditional causality (we regard the combination of several *transfer(item.info)* events and one *transfer(payment.info)* event as a single event, which we name *receive(info)* by abstraction):

(payment is correct amount) : receive(info) \rightsquigarrow generate(receipt.info).

This conditional is usually achieved by *Controller* comparing the total value of items via event *transfer(item.info)* with the total value of payment via event *transfer(payment.info)*, and if the latter is greater than or equal to the former, then *generate(receipt.info)* event should happen.

That completes all the steps of problem progression as the requirement statement R_{14} is expressed only in terms of specification phenomena - the *Controller* domain's behaviour. Figure 6.15 shows the final problem diagram after the problem progression.

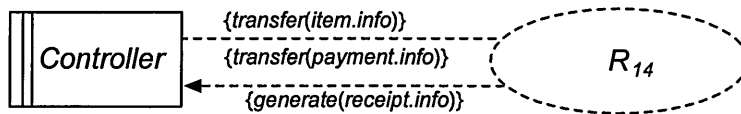


Fig. 6.15: Point-of-sale problem: final problem diagram after problem progression

Table 6.3 summarises the development of the requirement statements throughout the entire process of problem progression (next page).

Name	Description
R_1	When the <i>Customer</i> issues a number of <i>present(item)</i> , followed by one <i>present(payment)</i> , if <i>payment</i> is for the correct amount, then the <i>Customer</i> should observe <i>present(receipt)</i> .
R_2 (by rule ITOb)	When the <i>Cashier</i> observes a number of <i>present(item)</i> , followed by one <i>present(payment)</i> , if <i>payment</i> is for the correct amount, then the <i>Customer</i> should observe <i>present(receipt)</i> .
R_3 (by rule OTIa)	When the <i>Cashier</i> observes a number of <i>present(item)</i> , followed by one <i>present(payment)</i> , if <i>payment</i> is for the correct amount, then the <i>Cashier</i> should issue <i>present(receipt)</i> .
R_4 (by rule CTE(7)b)	When the <i>Cashier</i> issues a number of <i>enter(item.info)</i> , followed by one <i>enter(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>Cashier</i> should issue <i>present(receipt)</i> .
R_5 (by rule ETC(7)a)	When the <i>Cashier</i> issues a number of <i>enter(item.info)</i> , followed by one <i>enter(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>Cashier</i> should observe <i>print(receipt.info)</i> .
R_6 (by rules RD(3)a & b)	Assuming <i>Customer's</i> behaviour, when the <i>Cashier</i> issues a number of <i>enter(item.info)</i> , followed by one <i>enter(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>Cashier</i> should observe <i>print(receipt.info)</i> .
R_7 (by rule ITOb)	Assuming <i>Customer's</i> behaviour, when the <i>POS</i> observes a number of <i>enter(item.info)</i> , followed by one <i>enter(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>Cashier</i> should observe <i>print(receipt.info)</i> .
R_8 (by rule OTIa)	Assuming <i>Customer's</i> behaviour, when the <i>POS</i> observes a number of <i>enter(item.info)</i> , followed by one <i>enter(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>POS</i> should issue <i>print(receipt.info)</i> .
R_9 (by rule CTE(7)b)	Assuming <i>Customer's</i> behaviour, when the <i>POS</i> issues a number of <i>transfer(item.info)</i> , followed by one <i>transfer(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>POS</i> should issue <i>print(receipt.info)</i> .
R_{10} (by rule ETC(7)a)	Assuming <i>Customer's</i> behaviour, when the <i>POS</i> issues a number of <i>transfer(item.info)</i> , followed by one <i>transfer(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>POS</i> should observe <i>print(receipt.info)</i> .
R_{11} (by rules RD(3)a & b)	Assuming <i>Customer's</i> and <i>Cashier's</i> behaviour, when the <i>POS</i> issues a number of <i>transfer(item.info)</i> , followed by one <i>transfer(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>POS</i> should observe <i>generate(receipt.info)</i> .
R_{12} (by rule ITOb)	Assuming <i>Customer's</i> and <i>Cashier's</i> behaviour, when the <i>Controller</i> observes a number of <i>transfer(item.info)</i> , followed by one <i>transfer(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>POS</i> should observe <i>print(receipt.info)</i> .
R_{13} (by rule OTIa)	Assuming <i>Customer's</i> and <i>Cashier's</i> behaviour, when the <i>Controller</i> observes a number of <i>transfer(item.info)</i> , followed by one <i>transfer(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>Controller</i> should issue <i>print(receipt.info)</i> .
R_{14} (by rules RD(2)b & (1)a)	Assuming <i>Customer's</i> , <i>Cashier's</i> and <i>POS</i> behaviour, when the <i>Controller</i> observes a number of <i>transfer(item.info)</i> , followed by one <i>transfer(payment.info)</i> , if <i>payment</i> is for the correct amount, then the <i>Controller</i> should issue <i>print(receipt.info)</i> .

Tab. 6.3: Requirements transformations in the point-of-sale problem progression

6.2 The Package Router Problem

The second case study is a package router problem. It has been used as an example problem in [156, 10, 80, 83, 125], and originates from [73]. The problem statement is as follows [125]:

“A package router is a large machine used by delivery companies to sort packages into bins according to bar-coded destination labels affixed to the packages. Each bin corresponds to a regional area. Packages slide by gravity through a tree of pipes and binary switches. The bins are at the leaves of this tree.

The problem is to control the operation of the package router so that packages are routed to their appropriate bins, obeying the operator’s commands to start and stop the conveyor, and reporting any misrouted packages.”

Figure 6.16 is a schematic of the package router, and Figure 6.17 shows details of the pipes and switches.

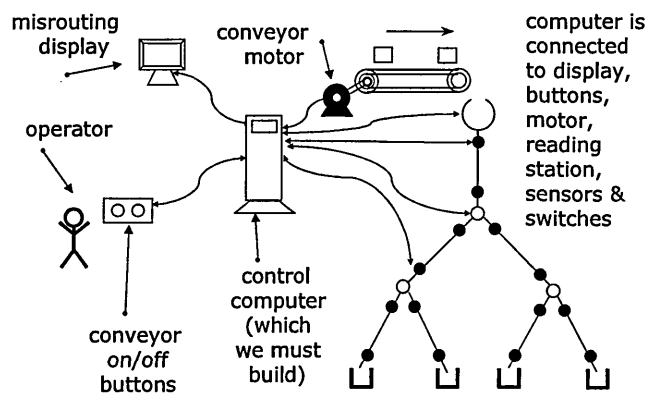


Fig. 6.16: Schematic of the package router problem taken from [125] (based on [83]), unmodified

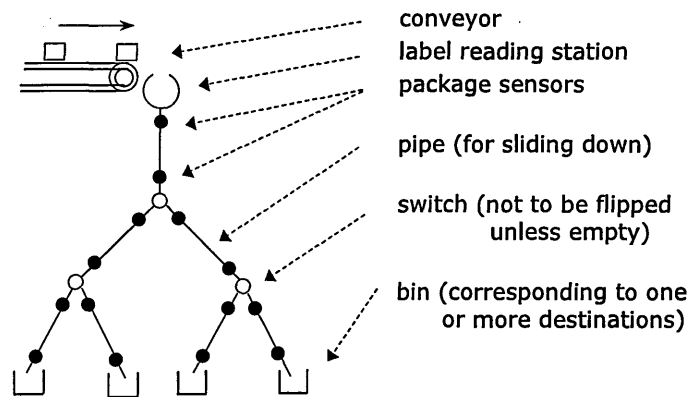


Fig. 6.17: Pipes and switches taken from [125] (based on [83]), unmodified

The analysis in [83] shows that this problem can be decomposed into the following subproblems:

P_1 = "The problem is to control the operation of the package router so that packages are routed to their appropriate bins."

P_2 = "The problem is to let the operation obey the operator's commands to start and stop the conveyor."

P_3 = "The problem is to report any misrouted packages."

Although each of them could be addressed through problem progression, for brevity we will focus on P_1 , which is the most complex of the three subproblems.

The problem statement does not tell us how many switches and bins are in the problem. For simplicity, we consider only two bins in our analysis which represent the situation in which a switch has two outgoing pipes releasing the package into two bins (increasing the number of switches does not affect our treatment of progression).

Let us look at the subproblem in more detail. There are five given domains in this subproblem: the *Reading station*, the *Switch*, the *Package*, the *Bin1*, and the *Bin2*. There

is also the *Controller* machine, which is the solution domain yet to be built. Table 6.4 shows the identified domains and their descriptions.

Name	Description
<i>Package</i>	The physical object (e.g., a mail or parcel) to be sorted to the correct bins for delivery. All packages carry bar-coded labels, which contain its <i>id</i> and destination <i>pkgDst</i> . In this simplified problem, <i>pkgDst</i> is either <i>left</i> or <i>right</i> ; in a problem with more than two bins, <i>pkgDst</i> is the destination bin number. They go through the reading station, after which they slide down through pipes and switches by gravity, and finally stop and arrive at their destination bins.
<i>Bin1, Bin2</i>	The container that the package is finally released. Each bin is dedicated to a group of adjacent areas (addresses) for delivery.
<i>Reading station</i>	The place through which the package is fed from the conveyor and its <i>id</i> and <i>destination</i> are read.
<i>Switch</i>	A two-position device that joins 3 pipes - one <i>incoming</i> pipe, one <i>left</i> pipe and one <i>right</i> pipe. It can be flipped to the <i>left</i> or to the <i>right</i> so that a package can only slide down one of the connected pipes (either <i>left</i> pipe or <i>right</i> pipe). The flipping is controlled by the controller to be built.
<i>Controller</i>	The solution machine to be designed. Its wired connection with the reading station allows it to indirectly access package ids and destinations; its wired connection with the switch allows it to control the flipping of switches.

Tab. 6.4: Domains and their descriptions

Figure 6.18 shows the problem diagram and Table 6.5 details its phenomena.

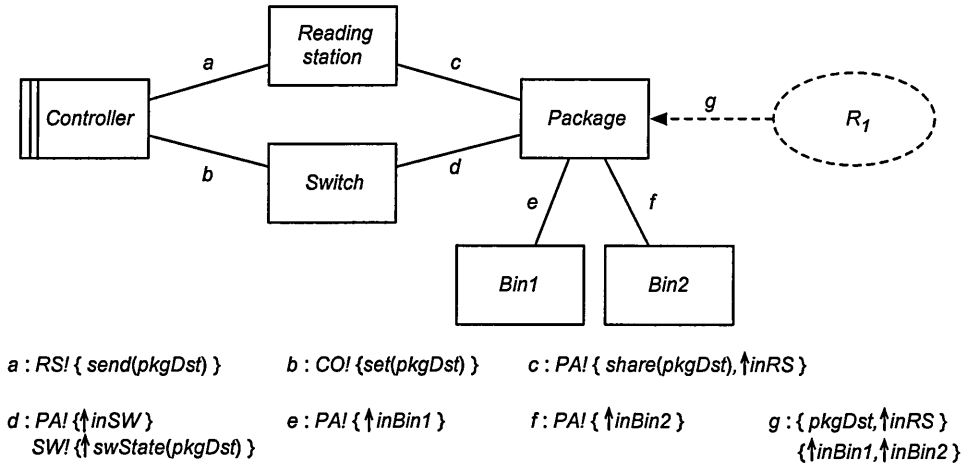


Fig. 6.18: Problem diagram

Name	Type	Designation
$\{send(pkgDst)\}$	shared event	The reading station sends the package destination $pkgDst$ to the controller. The destination $pkgDst$ can be <i>left</i> or <i>right</i> .
$\{set(pkgDst)\}$	shared event	The controller machine sets the switch to <i>left</i> . or <i>right</i> according to the package destination $pkgDst$.
$\{share(pkgDst), \uparrow inRS\}$	shared event	Once the package arrives at the reading station, i.e., $\uparrow inRS$ event occurs, the package's barcode label is shared with the reading station, i.e., $share(pkgDst)$ event occurs.
$\{\uparrow inSW\}$	shared event	Once the package is inside the switch, event $\uparrow inSW$ occurs, which is shared with the <i>Switch</i> domain, e.g., via optical sensors.
$\{\uparrow swState(pkgDst)\}$	shared event	Depending on the package's destination ($pkgDst = left$ or <i>right</i>), the switch is set accordingly, so event $\uparrow swState(pkgDst)$ is shared with the <i>Package</i> domain, which decides whether the package goes to <i>Bin1</i> or <i>Bin2</i> .
$\{\uparrow inBin1\}$	shared event	When the package enters <i>Bin1</i> , event $\uparrow inBin1$ occurs.
$\{\uparrow inBin2\}$	shared event	When the package enters <i>Bin2</i> , event $\uparrow inBin2$ occurs.
$\{pkgDst, \uparrow inRS\}$	internal state/ shared event	The package's destination $pkgDst$ namely <i>left</i> or <i>right</i> in this simplified problem diagram, is encoded in the package's label (barcode). Event $\uparrow inRS$ occurs when the package enters the reading station.
$\{\uparrow inBin1, \uparrow inBin2\}$	shared event	Once the package enters <i>Bin1</i> or <i>Bin2</i> , the event $\uparrow inBin1$ or $\uparrow inBin2$ occurs.

Tab. 6.5: Phenomena and their designations

The *Package* domain is a causal domain with complex behaviours which can be partially expressed by a state machine diagram [116] (assuming it does not break) in Figure 6.19 (next page). The timed transitions capture the time duration a package needs to slide from one part of the routing device to the next.

The following causal relations can be derived from Figure 6.19:

- $\uparrow inRS \xrightarrow{x+y} \uparrow inSW$ means that the package entering the *Reading station* will cause it to enter the *Switch* after $x + y$ seconds.

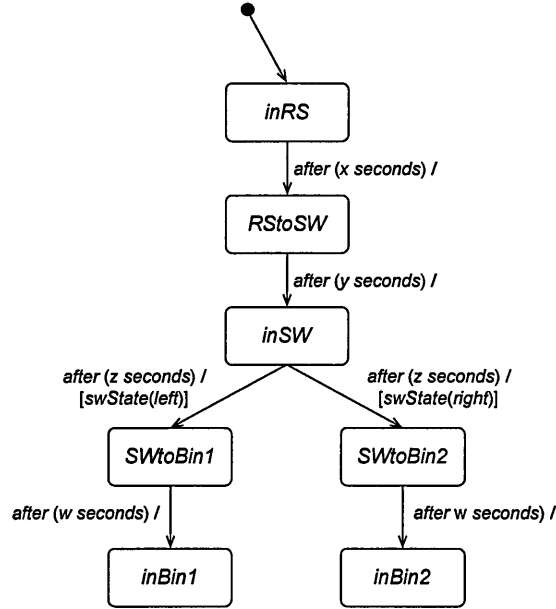


Fig. 6.19: Package behaviour described as a state machine adapted from [127], modified

- $(swState(left)) : \uparrow inSW \xrightarrow{z+w} \uparrow inBin1$ means that if the *Switch* is set to the left, then the package entering the the *Switch* will cause it to enter *Bin1* after $z + w$ seconds.
- $(swState(right)) : \uparrow inSW \xrightarrow{z+w} \uparrow inBin2$ means that if the *Switch* is set to the right, then the package entering the the *Switch* will cause it to enter *Bin2* after $z + w$ seconds.

However, the following phenomena (including shared and internal ones) are not explicitly described in Figure 6.19:

- the internal phenomena that every package has a unique *id* (may be useful for other subproblems, e.g., tracking/displaying/reporting misrouted package) and destination *pkgDst*, which, for this simplified problem, is a state with two val-

ues: either $pkgDst = left$ or $pkgDst = right$ (for problems with more than two bins, $pkgDst$ should be the target bin number);

- the package shares phenomenon $shared(pkgDst)$ with the *Reading station*, where $pkgDst$ represents package destination: either $pkgDst = left$ or $pkgDst = right$. It is controlled by the *Package* domain. There are the following causal relations:

$(pkgDst = left) : \uparrow inRS \rightsquigarrow share(left)$, and

$(pkgDst = right) : \uparrow inRS \rightsquigarrow share(right)$;

- the *Switch*'s state $swState(left)$ or $swState(right)$ is shared between the *Switch* domain and the *Package* domain, and it is controlled by the former. These shared phenomena determine whether the package goes to the *left* bin *Bin1* or the *right* bin *Bin2* (as captured by the causal relations in the package description earlier on).

Bin1 and *Bin2* are simple causal domains with sensors at their entrances. Their shared phenomena with the *Package* domain, namely $\uparrow inBin1$ and $\uparrow inBin2$ will allow the package into them.

The *Reading station* domain is causal, with the following causal relation:

$shared(pkgDst) \rightsquigarrow send(pkgDst)$, where $pkgDst \in \{left, right\}$,

which means that the bar-code for the package's destination $pkgDst$, namely *left* or *right*, is shared with (or scanned by) the reading station, which will cause the reading station to send the package's destination information to the *Controller* domain.

The *Switch* domain is causal, with the following causal relation:

$set(pkgDst) \rightsquigarrow \uparrow swState(pkgDst)$, where $pkgDst \in \{left, right\}$,

which means that the *Controller* issuing $set(left)$ will cause the switch's state $swState$ to

become *left*, and the *Controller* issuing *set(right)* will cause the switch's state *swState* to become *right*.

The requirement, R_1 can be stated as follows:

$R_1 =$ “If the package's destination is *pkgDst*, with $\text{pkgDst} = \text{left}$ or $\text{pkgDst} = \text{right}$, and the package enters the reading station (i.e., $\uparrow \text{inRS}$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow \text{inBin1}$ or $\uparrow \text{inBin2}$ occurs) after $x + y + z + w$ seconds.”.

Notice that we have expressed the requirement in terms of the identified problem phenomena. This will allow us to progress it through to specification by repeatedly transforming it until the requirement is expressed only in terms of the specification phenomena.

The above requirement statement R_1 relates two separate sets of phenomena, namely those that R_1 refers to, i.e., $\{\text{pkgDst}, \uparrow \text{inRS}\}$, and those that R_1 constrains, i.e., $\{\uparrow \text{inBin1}, \uparrow \text{inBin2}\}$. As causality is timed in this problem, a time constraint is also expressed by R_1 on the total travelling time of the package through the router. This relation should be achieved by the entire routing device including the *Reading station* and the *Switch* domains, which are directly connected to the *Package* domain, and the *Controller* domain, which is indirectly connected to *Package*.

6.2.1 First Step of Progression

Let us look at *pkgDst*, which is internal to *Package*, and $\uparrow \text{inRS}$, which is shared between *Package* and *Reading station*. Recall that the following causal relations exist:

$(\text{pkgDst} = \text{left}) : \uparrow \text{inRS} \rightsquigarrow \text{share}(\text{left})$, and

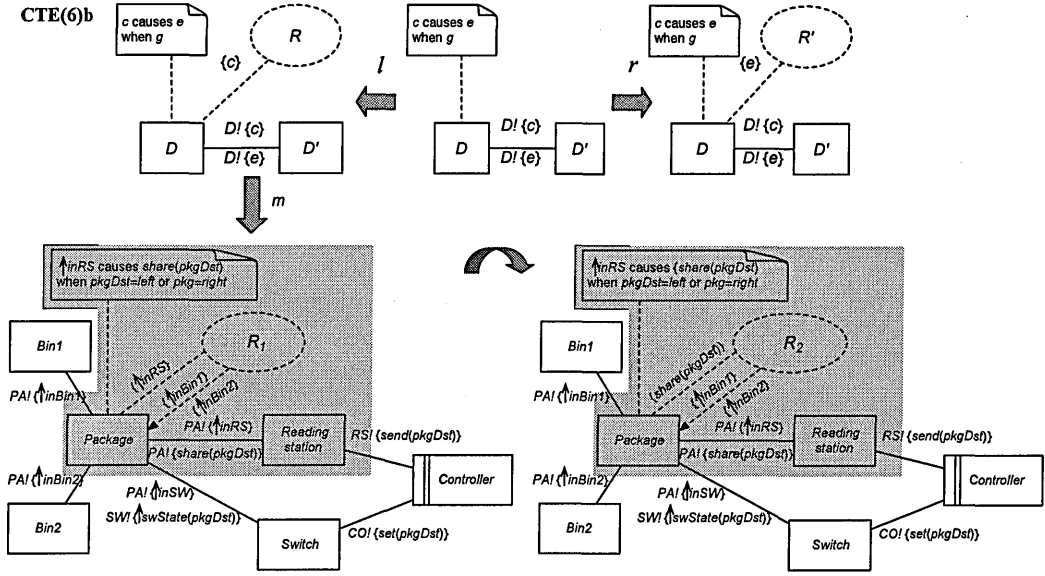


Fig. 6.20: Appropriate package routing progression step 1: applying the cause-to-effect rule CET(6)b

$(pkgDst = right) : \uparrow inRS \rightsquigarrow share(right),$

which allow us to apply the cause-to-effect rule CET(6)b to replace event $\uparrow inRS$ with event $share(pkgDst)$, as shown in Figure 6.20.

By applying the rule, we arrive at the following requirement statement:

$R_2 =$ “If the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the package shares $pkgDst$ with the reading station (i.e., $share(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x + y + z + w$ seconds.”.

6.2.2 Second Step of Progression

In the second step, we apply the issue-to-observe rule ITOb, and switch from the *Package* to the *Reading station*, as they share event $share(pkgDst)$, as shown in Figure 6.21.

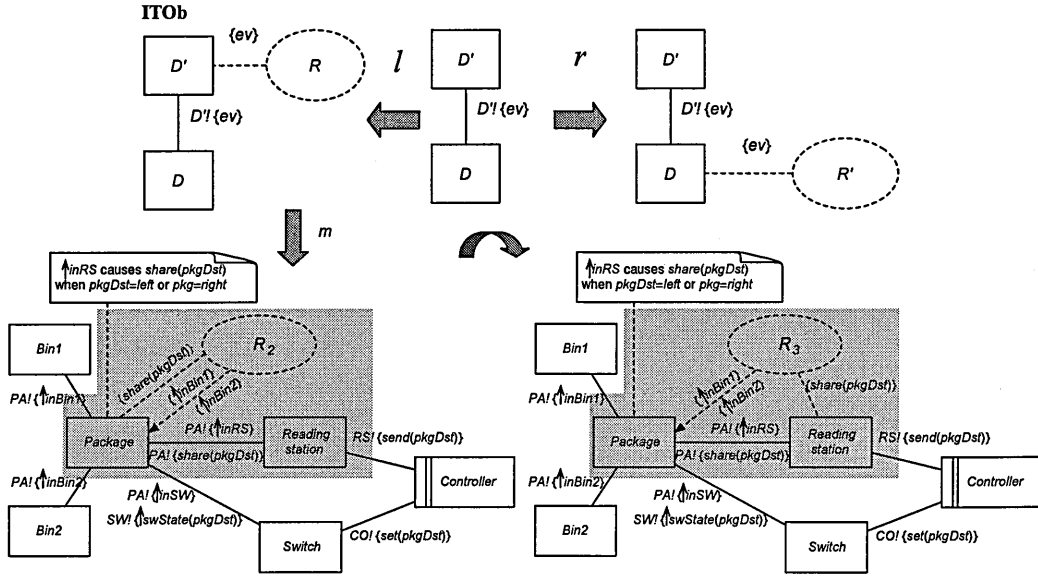


Fig. 6.21: Appropriate package routing progression step 2: applying the issue-to-observe rule ITOb

By applying the rule, we arrive at the following requirement statement:

R_3 = “If the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the reading station reads $pkgDst$ (i.e., $share(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x + y + z + w$ seconds.”.

6.2.3 Third Step of Progression

In the third step, *Reading station* has the following domain properties (causal relations):

$shared(pkgDst) \rightsquigarrow send(pkgDst)$, where $pkgDst \in \{left, right\}$,

which allow us to apply the cause-to-effect rule CTE(7)b to replace $share(pkgDst)$ with $send(pkgDst)$, as shown in Figure 6.22.

By applying the rule, we arrive at the following requirement statement:

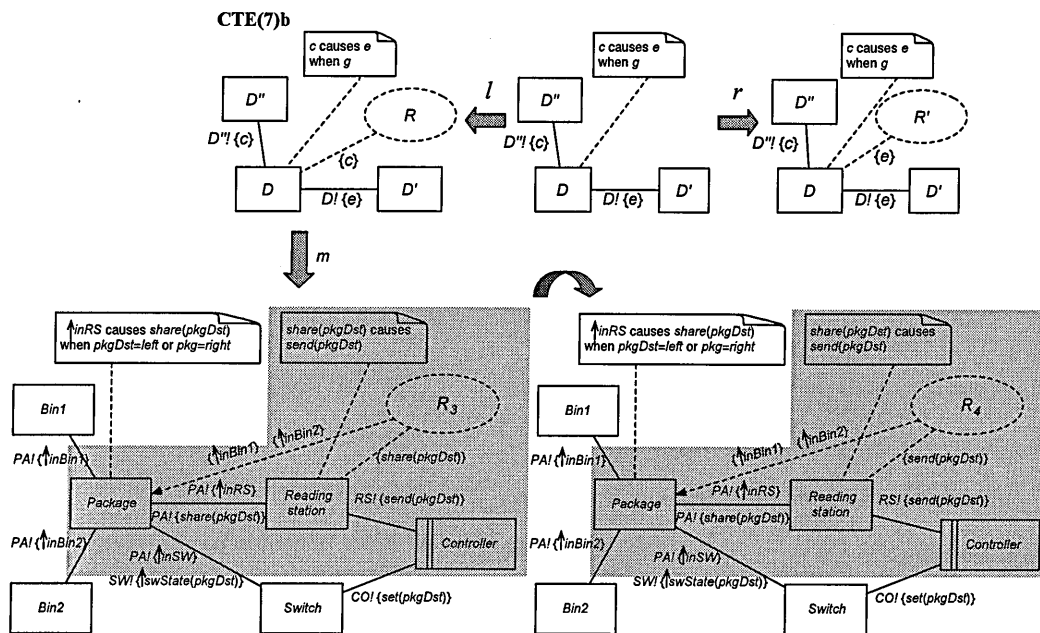


Fig. 6.22: Appropriate package routing progression step 3: applying the cause-to-effect rule
CTE(7)b

$R_4 =$ “If the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the reading station sends $pkgDst$ to the controller (i.e., $send(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x + y + z + w$ seconds.”.

6.2.4 Fourth Step of Progression

In the fourth step, we apply the issue-to-observe rule ITOb, and switch from the *Reading station* to the *Controller*, as they share $send(pkgDst)$, as shown in Figure 6.23.

By applying the rule, we arrive at the following requirement statement:

$R_5 =$ “If the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then

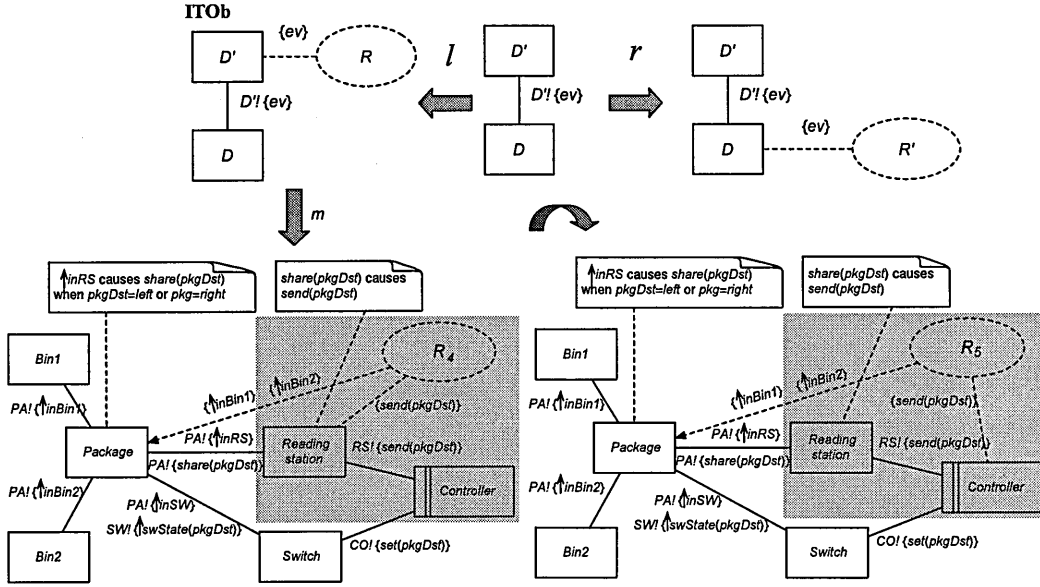


Fig. 6.23: Appropriate package routing progression step 4: applying the issue-to-observe rule ITOb

the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x + y + z + w$ seconds.”.

From progression step 1 through to step 4, we have partially progressed the original requirement statement R_1 to R_5 , by rewriting the first half of the statement each time. Next we will progress the second half of the statement.

6.2.5 Fifth Step of Progression

In the fifth step, *Package* has the following domain properties (causal relations):

$$\begin{aligned}
 (swState(left)) : \uparrow inSW &\overset{z+w}{\rightsquigarrow^+} \uparrow inBin1, \text{ and} \\
 (swState(right)) : \uparrow inSW &\overset{z+w}{\rightsquigarrow^+} \uparrow inBin2,
 \end{aligned}$$

which allow us to apply the effect-to-cause rule ETC(7)a twice to replace $\uparrow inBin1$ and $\uparrow inBin2$ with $\uparrow inSW$ and $swState(pkgDst)$ holds, as shown in Figure 6.24. Note state

$swState(pkgDst)$ is shared with the *Package* domain and controlled by *Switch* domain, as shown in Figure 6.24. In the figure, for brevity we use event $\uparrow swState(pkgDst)$ as a short form of $\uparrow swState(left)$ or $\uparrow swState(right)$.

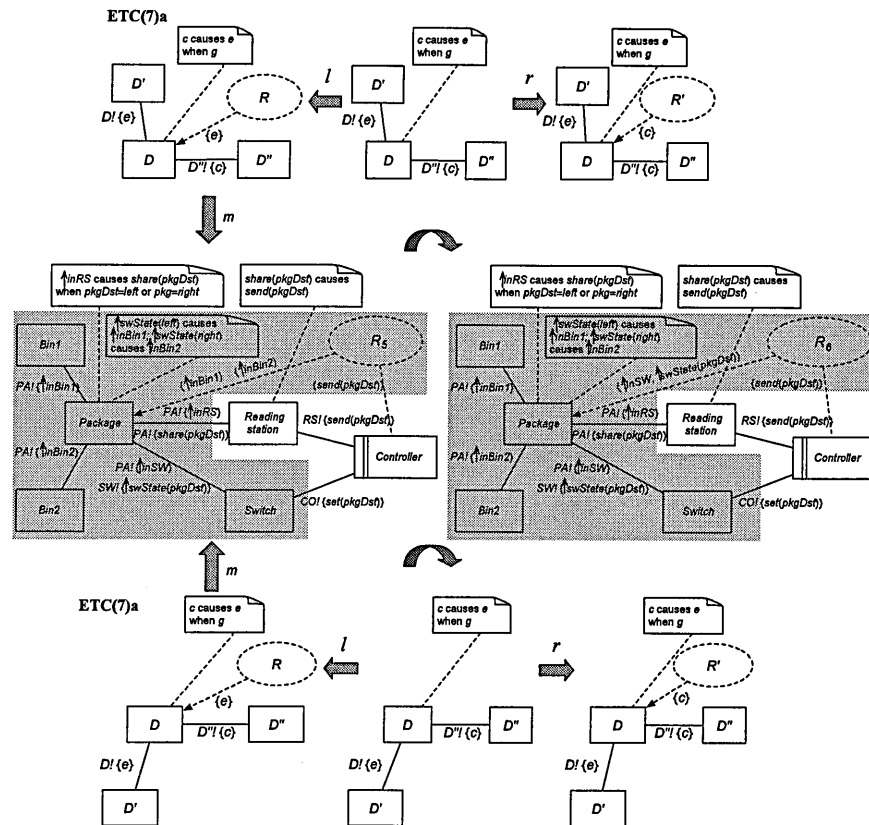


Fig. 6.24: Appropriate package routing progression step 5: applying the rule ETC(7)a

By applying the rule twice, we arrive at the following requirement statement:

$R_6 =$ “If the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the package should enter the switch (i.e., $\uparrow inSW$ occurs) with the switch state appropriately set (i.e., either $swState(left)$ or $swState(right)$), depending on the value of $pkgDst$ after $x + y$ seconds.”.

Note that the time $z + w$ of transit of the package from the switch to the bin has been taken into account in the rewritten R_6 .

6.2.6 Sixth Step of Progression

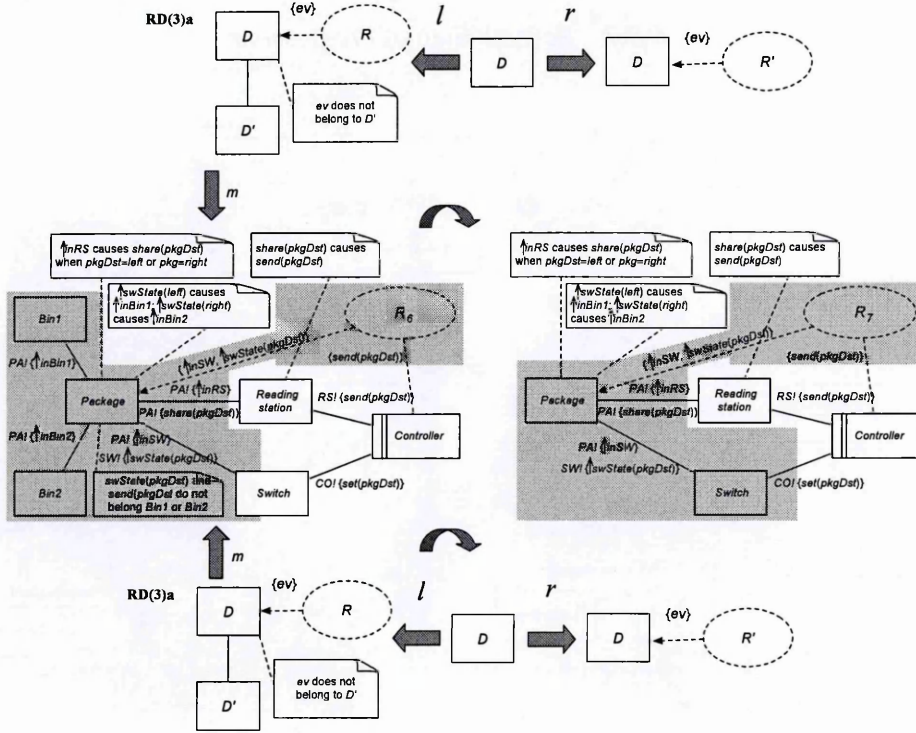


Fig. 6.25: Appropriate package routing progression step 6: applying the rule RD(3)a twice

In the sixth step, we apply the removing domain rule RD(3)a twice to remove $Bin1$ and $Bin2$ as shown in Figure 6.25, and we arrive at the rewritten requirement:

R_7 = "Assuming the behaviour of $Bin1$ and $Bin2$, if the package's destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the package should enter the switch (i.e., $\uparrow inSW$ occurs) with the switch state appropriately set (i.e.,

either $swState(left)$ or $swState(right)$) depending on the value of $pkgDst$ after $x + y$ seconds.”.

Application of these rules is justified by the fact that R_6 does not constrain or refer to $Bin1$'s or $Bin2$'s phenomena anymore, hence they can be removed from the diagram.

6.2.7 Seventh Step of Progression

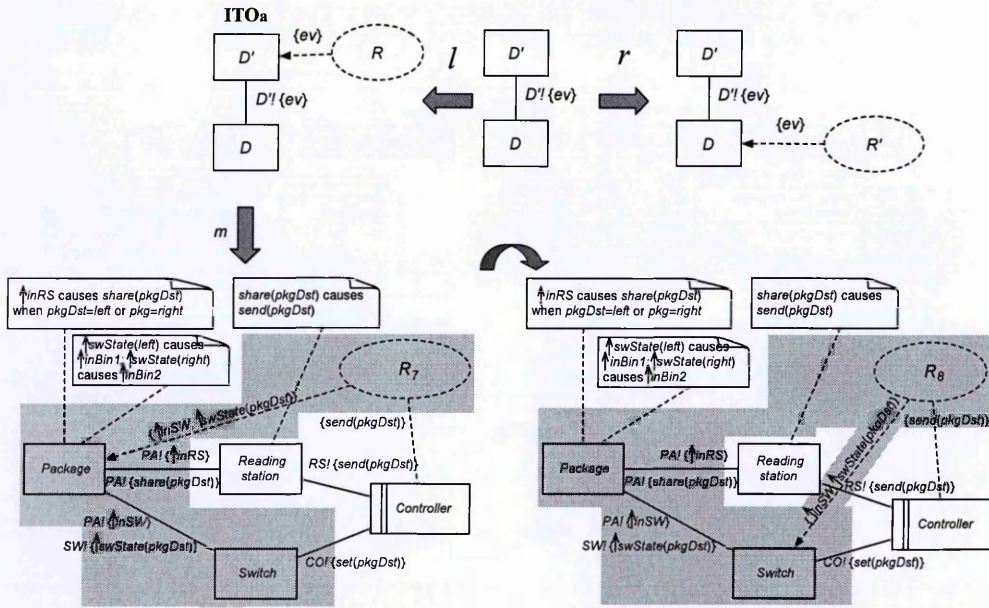


Fig. 6.26: Appropriate package routing progression step 7: applying the issue-to-observe rule ITOa

In the seventh step, we apply the issue-to-observe rule ITOa twice, and switch from the *Package* to the *Switch*, as they share event $\uparrow inSW$, and $\uparrow swState(pkgDst)$, as shown in Figure 6.26.

By applying the rule, we arrive at the following requirement statement:

R_8 = “Assuming the behaviour of *Bin1* and *Bin2*, if the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the controller

receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the switch observes the package entering (i.e., $\uparrow inSW$ occurs), with the switch state appropriately set (i.e., either $swState(left)$ or $swState(right)$), depending on the value of $pkgDst$ after $x + y$ seconds.”.

6.2.8 Eighth Step of Progression

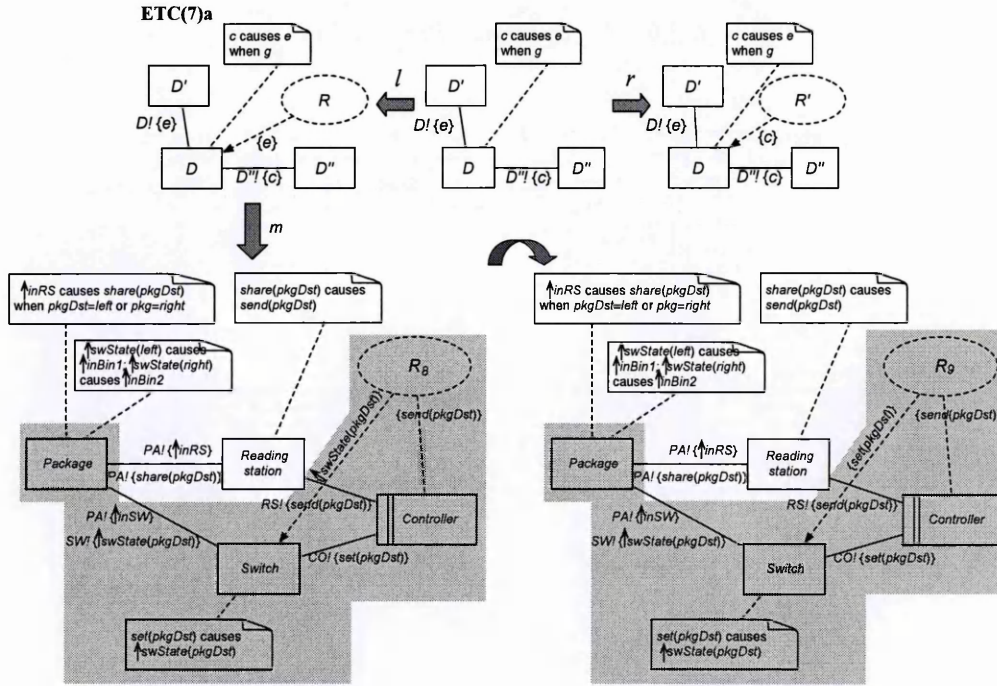


Fig. 6.27: Appropriate package routing progression step 8: applying the effect-to-cause rule ETC(7)a

In the eighth step, *Switch* has the following domain properties (causal relations):

$set(pkgDst) \rightsquigarrow \uparrow swState(pkgDst)$, where $pkgDst \in \{left, right\}$,

which allow us to apply the effect-to-cause rule ETC(7)a to replace $swState(pkgDst)$ with $set(pkgDst)$, as shown in Figure 6.27.

By applying the rule, we arrive at the following requirement statement:

R_9 = “Assuming the behaviour of Bin1 and Bin2, if the package’s destination is $pkgDst$, with $pkgDst = left$ or $pkgDst = right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the switch should receive commands from the controller to set its state appropriately set (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x+y$ seconds.”.

6.2.9 Ninth Step of Progression

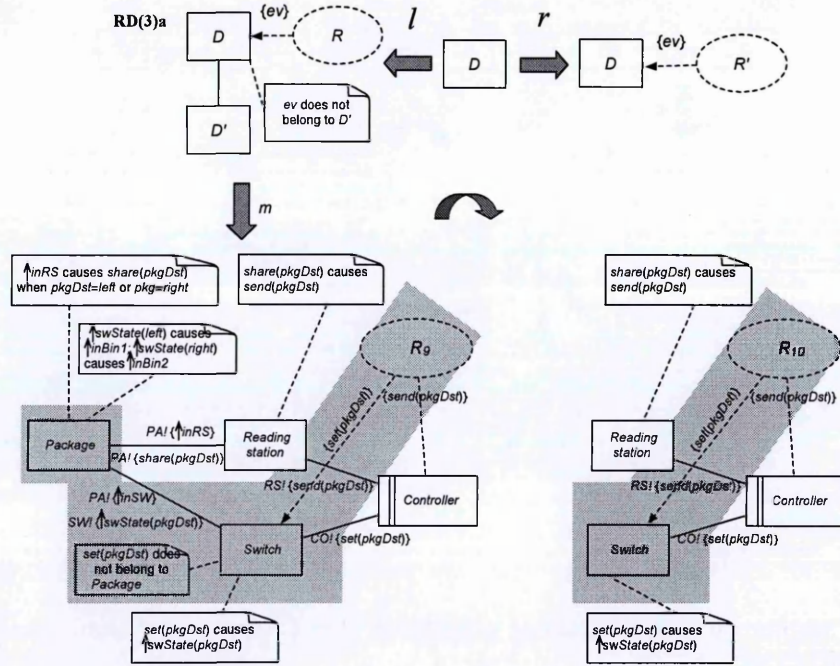


Fig. 6.28: Appropriate package routing progression step 9: applying the rule RD(3)a

In the ninth step, we apply the removing domain rule RD(3)a to remove *Package* as shown in Figure 6.28, and we arrive at the rewritten requirement:

R_{10} = “Assuming the behaviour of Bin1, Bin2 and *Package*, if the con-

troller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), with $pkgDst = left$ or $pkgDst = right$, then the switch should receive commands from the controller to set its state appropriately set (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x + y$ seconds.”.

Application of this rule is justified by the fact that R_9 does not constrain or refer to *Package*’s phenomena anymore, hence it can be removed from the diagram. Any dog-eared box that is attached to *Package* is part of *Package*’s domain properties, hence is removed together with *Package*.

6.2.10 Tenth Step of Progression

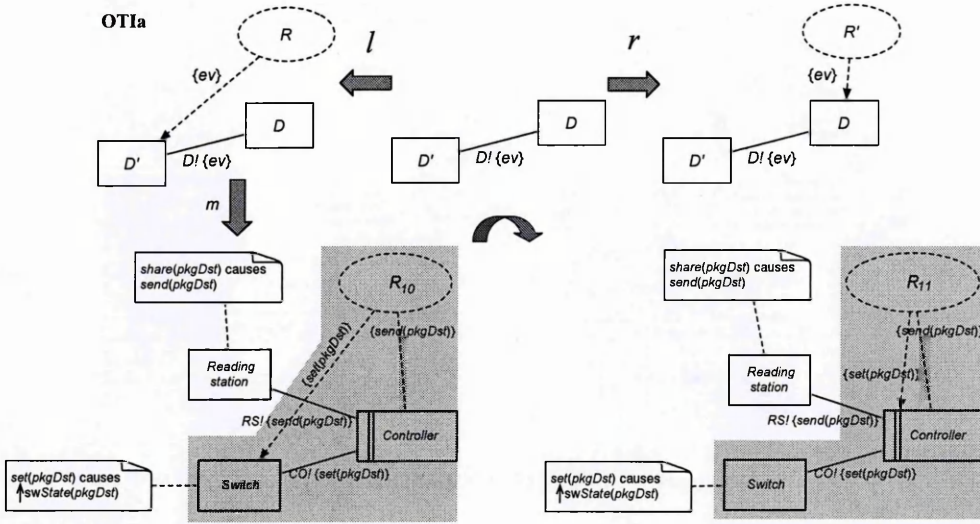


Fig. 6.29: Appropriate package routing progression step 10: applying the observe-to-issue rule OTIa

In the tenth step, we apply the observe-to-issue rule OTIa, and switch from domain *Switch* to the *Controller*, as they share $set(pkgDst)$, as shown in Figure 6.29.

By applying the rule, we arrive at the following requirement statement:

R_{11} = “Assuming the behaviour of *Bin1*, *Bin2* and *Package*, if the controller receives *pkgDst* (i.e., $\text{send}(\text{pkgDst})$ occurs), with $\text{pkgDst} = \text{left}$ or $\text{pkgDst} = \text{right}$, then the controller should issue commands to set the switch state appropriately set (i.e., either $\text{set}(\text{left})$ or $\text{set}(\text{right})$ occurs), depending on the value of *pkgDst* after $x + y$ seconds.”.

6.2.11 Eleventh Step of Progression

In the eleventh step, we apply the removing domain rule RD(1)a first to remove *Switch* (see the Figure 6.30), and then rule RD(2)b to remove *Reading station* (see Figure 6.31 on the next page),

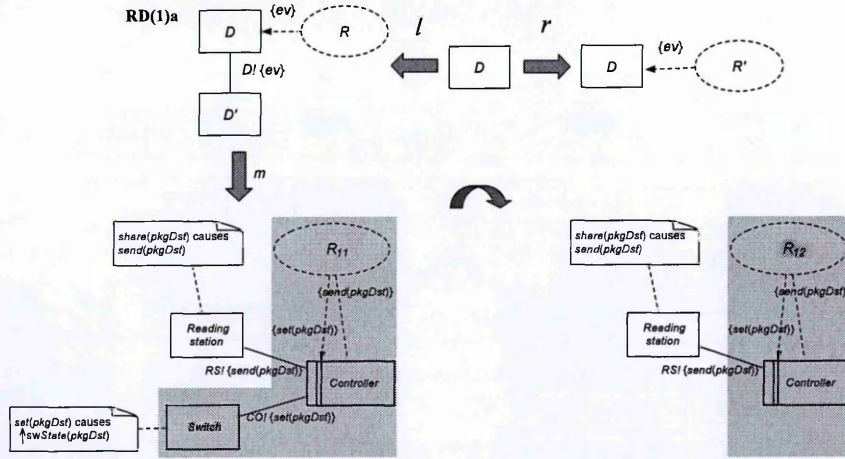


Fig. 6.30: Appropriate package routing progression step 11 (1): applying rule RD(1)a

By applying the rules, we arrive at the following requirement statement:

R_{12} = “Assuming the behaviour of *Bin1*, *Bin2*, *Package*, *Switch* and *Reading station*, if the controller receives *pkgDst* (i.e., $\text{send}(\text{pkgDst})$ occurs), with $\text{pkgDst} = \text{left}$ or $\text{pkgDst} = \text{right}$, then the controller should issue appro-

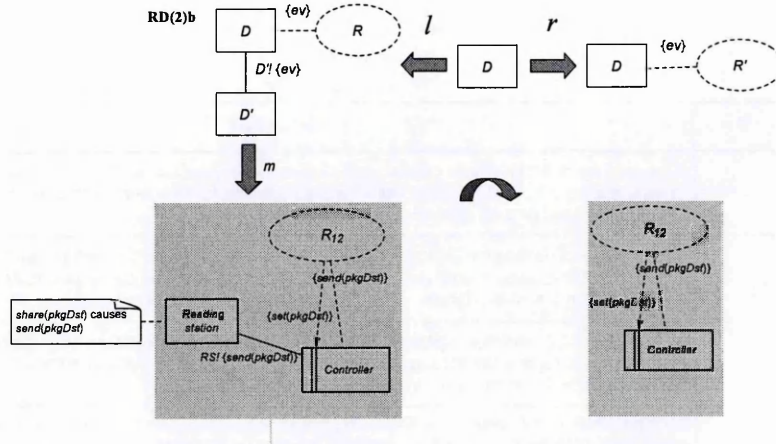


Fig. 6.31: Appropriate package routing progression step 11 (2): applying rule RD(2)b

priate commands (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x + y$ seconds.”.

That completes all the steps of problem progression as the requirement statement R_{12} is expressed only in terms of specification phenomena, i.e., all *Controller*’s phenomena. Figure 6.32 shows the final problem diagram after the problem progression.

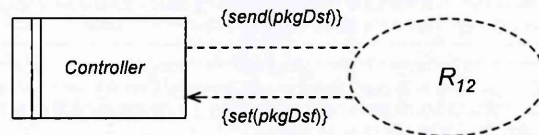


Fig. 6.32: Final problem diagram after problem progression

Table 6.6 summarises the development of the requirement statements throughout the entire process of problem progression (next page).

Name	Description
R_1	If the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the package enters the reading station (i.e., $\uparrow inRS$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x+y+z+w$ seconds.
R_2 (by rule CET(6)b)	If the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the package shares $pkgDst$ with the reading station (i.e., $share(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x+y+z+w$ seconds.
R_3 (by rule ITOb)	If the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the reading station reads $pkgDst$ (i.e., $share(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x+y+z+w$ seconds.
R_4 (by rule CTE(7)b)	If the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the reading station sends $pkgDst$ to the controller (i.e., $send(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x+y+z+w$ seconds.
R_5 (by rule ITOb)	If the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the package should enter the appropriate bin (i.e., either $\uparrow inBin1$ or $\uparrow inBin2$ occurs) after $x+y+z+w$ seconds.
R_6 (by rule ETC(7)a)	If the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the package should enter the switch (i.e., $\uparrow inSW$ occurs) with the switch state appropriately set (i.e., either $swState(left)$ or $swState(right)$), depending on the value of $pkgDst$ after $x+y$ seconds.
R_7 (by rule RD(3)a)	Assuming the behaviour of $Bin1$ and $Bin2$, if the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the package should enter the switch (i.e., $\uparrow inSW$ occurs) with the switch state appropriately set (i.e., either $swState(left)$ or $swState(right)$) depending on the value of $pkgDst$ after $x+y$ seconds.
R_8 (by rule ITOa)	Assuming the behaviour of $Bin1$ and $Bin2$, if the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the package should enter the switch (i.e., $\uparrow inSW$ occurs), with the switch state appropriately set (i.e., either $swState(left)$ or $swState(right)$), depending on the value of $pkgDst$ after $x+y$ seconds.
R_9 (by rule ETC(7)a)	Assuming the behaviour of $Bin1$ and $Bin2$, if the package's destination is $pkgDst$, with $pkgDst=left$ or $pkgDst=right$, and the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), then the switch should receive commands from the controller to set its state appropriately set (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x+y$ seconds.
$R_{10} =$ (by rules RD(3)a)	Assuming the behaviour of $Bin1$, $Bin2$ and $Package$, if the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), with $pkgDst=left$ or $pkgDst=right$, then the switch should receive commands from the controller to set its state appropriately set (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x+y$ seconds.
$R_{11} =$ (by rules OT1a)	Assuming the behaviour of $Bin1$, $Bin2$ and $Package$, if the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), with $pkgDst=left$ or $pkgDst=right$, then the controller should issue commands to set the switch state appropriately set (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x+y$ seconds.
$R_{12} =$ (by rules RD(1)a&(2)b)	Assuming the behaviour of $Bin1$, $Bin2$, $Package$, $Switch$ and $Reading station$, if the controller receives $pkgDst$ (i.e., $send(pkgDst)$ occurs), with $pkgDst=left$ or $pkgDst=right$, then the controller should issue appropriate commands (i.e., either $set(left)$ or $set(right)$ occurs), depending on the value of $pkgDst$ after $x+y$ seconds.

Tab. 6.6: Requirements transformations in the package router problem progression

6.3 Discussions

The POS example has demonstrated the progression of a simple problem: the domains are linearly arranged; there are no timing issues and causality is not conditional. Progression rules were applied to a matched part of the problem diagram in a stepwise manner. In each step, a small portion of the texts was manipulated according to the templates set out by the application conditions of the rule. Assumptions about the removed domains were explicitly stated in the rewritten requirements, which guaranteed that the transformation was solution-preserving. In this case study, our progression only arrives at a high-level behaviour description of the *Controller* machine, while low-level software design is left the developer to decide. In comparison, the formal approach in Chapter 4 applied to a similar problem has forced us to reason more rigourously about this low-level design, thus leading to more detailed design.

The package router example has addressed more complex causal relations than the POS problem. In particular, the *Package*'s domain properties involve time considerations in the causal relation which capture the passage of time as the package travels through the router. This has required us to add timing constraints in requirement statements.

There are issues in the problem which we have not explored. For instance, it was not decided whether the routing device should serve one package at one time or multiple packages. In the latter case, the minimum time lag between two packages would have to be enforced so that the flipping of the switches can be co-ordinated to avoid conflicts. This might require the machine to be constantly updated with each package's position in the device to achieve maximum efficiency. In this situation, a model domain that is connected to the sensors reflecting the real time positions of all packages should be

built. The requirement would be more complex, but the progression should be still addressable with our techniques.

6.4 Chapter Summary

This chapter has demonstrated how the notion of causality and associated rule-based techniques can be applied in the context of problem frames to address problem progression. The case studies illustrated a systematic process of deriving a machine specification from the requirement, including cases in which biddable and timed causality should be considered.

7. DISCUSSIONS, CONCLUSIONS AND FUTURE WORK

In this chapter, we review the aim of this thesis and assess the extent to which our techniques fulfil this aim. Based on their applications to the case studies, we compare and evaluate the two different classes of techniques which have been introduced in Chapter 4 and Chapter 5. Finally, conclusions on the work are drawn and an agenda for future work is proposed.

7.1 *Aim of the Thesis and Contribution Evaluation*

In the beginning of the thesis, we have set out the following aim of this thesis:

to derive specifications from requirements in a systematic way by defining practical techniques to implement problem progression.

We presented two contributions of this thesis to fulfil the above aim. The first is a *formal approach* incorporating Lai's quotient operator and other CSP notations for the derivation of specifications from requirements which can be formally described. The second is a *semi-formal approach* incorporating the notion of causality and associated rule-based techniques for the practical derivation of specifications from requirements in a wider range of problems.

In this discussion, we will examine both approaches and associated techniques in terms of the following aspects: whether they provide a *systematic* solution, the *scope* of

their application, and the *practicality* of their application.

7.1.1 How Systematic Are They?

According to the Oxford English Dictionary [76], the word “systematic” means “arranged or conducted according to a system, plan, or organized method”. Therefore, the question is: “Can our techniques and methods be applied in an orderly manner so that useful results can be achieved?”.

The formal approach is systematic due to the nature of the operations defined over process and specification terms. Within this approach, various CSP operators, particularly Lai’s quotient operator and the parallel composition operator allow us to derive systematically specifications from requirements. The case study in Chapter 4 demonstrates how such techniques can be applied systematically to construct a correct specification. The results were checked rigourously through the FDR tool as a way of validating the correctness of its construction.

The semi-formal approach is also systematic because our classes of progression rules give a complete coverage of all possible problem topologies. In other words, for any valid problem diagram, we can systematically match and find a progression rule to reason through a domain’s causal behaviours.

7.1.2 Scope of Their Application

The formal approach has limited scope of application in RE. We can only apply the techniques when we can express domain properties and requirements as CSP expressions. The case study in Chapter 4 suggests that if we can express the domains and requirements using CSP descriptions, we can construct the solution specification in a

systematic way. The study also indicates that the formal techniques become very complex and are unlikely to scale up to real-world problems.

The semi-formal approach has a much wider scope of application. We can apply the progression rules as long as causal relationships can be established about domain properties, and certain chains of causality can be identified in a problem diagram. Since the definition and application conditions of the progression rules are based on a fixed pattern of natural language descriptions, we argue that this approach is more general for RE. A comparison of the case studies in Chapter 4 and in Chapter 6 shows how the semi-formal approach can tackle much more complex problems than the formal one.

7.1.3 *Practicality of Their Application*

Let us evaluate how the techniques can be practically applied in RE.

The formal approach has limited practicality of application in RE. A large amount of complex formal manipulations is needed for progressing even a very simple problem, as shown in the case study in Chapter 4. It is not very realistic to expect RE practitioners to have sufficient knowledge of CSP and the predicate calculus, and the ability to perform the formal manipulations.

The semi-formal approach is based on causality, and its complexity lies in identifying causal relationships within domain descriptions. However, in this thesis, we have classified and elaborated the notion of causality in order to facilitate the organisation and representation of complex causal relationships. This may help in eliciting the required knowledge from problem stakeholders for the analysis of a particular problem. Therefore, we argue that our causality-based techniques could fit within many RE practices, thus having the potential to be adopted by practitioners.

7.2 *Conclusion and Future Work*

Reflecting back on our work presented in this thesis, we conclude that our aim of deriving specifications from requirements in a systematic way was achieved by our work. That such aim was worth investigating was justified by the literature survey in Chapter 2, which suggested that the systematic derivation of specifications from requirements is a challenging but important open problem in software engineering. We have investigated two approaches, one formal and one semi-formal, to address this problem. Here is a summary of our investigation:

The difference between the formal and semi-formal approaches has been well emphasised by the relevant chapters. Formality, whilst appropriate in the most critical of developmental situations, requires too much work in terms of the production of formal descriptions and working with them to produce a closed-form solution. Application of the formal technique outside this scope is less likely to work for the reasons we have discussed in this thesis. Instead our semi-formal technique has a much wider scope of application and a better chance of integration in current requirements engineering practices.

One promising direction for the semi-formal technique is developing tool support. The problem progression process in Chapter 6 requires many tedious steps. There is a need for simplifying this process without sacrificing the rigour. As an initial step, perhaps the tool will allow practitioners to help the identification of causal phenomena, which will be used for justifying the injective matching of our progression rules, then the tool will mechanically search and identify all sound instances of graph transformation convergence, which will be chosen by the requirements engineer.

The solution we have presented is partial: as can be seen from Chapters 4 and 6,

there are problems that require other problem-solving techniques in addition to those we have detailed. From requirements to specifications, it may be close to the best we can do: the application domain will always require the manipulation of informal descriptions, and we have by necessity been limited to the recognition and manipulation of unambiguous descriptions of causal relations.

For the semi-formal techniques we have proposed, one difficulty we have not addressed is that in any real-world development context there will typically be many validating stakeholders, such as customers, legislators and regulators, each of whom will have a different view on what are the important (and obvious) causal relations. This leads us to consider whether the conceptual basis we have worked with is indeed a complete picture: it may be that, because of the differing views of stakeholders, problems need to be parameterised for each of them. In this case, it is the intersection of the stakeholders' solutions that must be found. Future work may consider how our approach can be extended to generate a solution within that intersection. One remedy might be to begin with descriptions whose meaning is agreed by all stakeholders before commencing the solution process we have presented. In this case, the framework we have provided becomes as general as possible.

Another area for future work is that we have tried, in this thesis, to provide a framework for constructing solutions to problems, ensuring that if we start from a valid problem description, through transformation the solution will be valid too. We note that a framework for solution synthesis is much more demanding than a framework for problem analysis: solution synthesis requires problem analysis as an initial part, as well as creative steps that generate solutions from problems. We have gone some small way to show how this can be done with our techniques, but there is still some way to go to provide tools adequate for computing as engineering.

APPENDIX

A. DETAILS OF DISTINGUISHING “CONTROL” AND “OBSERVE” IN CSP DESCRIPTIONS

In CSP, a process may appear in any of the following syntax:

$$P ::= STOP_A \mid CHAOS_A \mid c!e \rightarrow P \mid c?x \rightarrow P \mid P \sqcap Q \mid \\ P \sqcup Q \mid P \parallel Q \mid P \setminus c \mid \mu X : A.F(X),$$

and only some have the above property. For instance, $c?x \rightarrow STOP \parallel c!1 \rightarrow STOP$ does not.

In the following, in order to make $P! \cap P? = \{\}$ hold, we need to restrict each part of P , shown below:

(A). According to definition (a) and (b), and the semantics of $STOP_A$ (A is its alphabet),

$$STOP_A! = \{d \mid d!v \in STOP_A\} = \{\}, \text{ and}$$

$$STOP_A? = \{d \mid d?x \in STOP_A\} = \{\},$$

Since $STOP_A! \cap STOP_A? = \{\}$, there is no need to restrict $STOP_A$.

(B). According to definition (a) and (b), and the semantics of $CHAOS_A$ (non-empty set A is its alphabet),

$$CHAOS_A! = \{d \mid P = CHAOS_A \wedge d!v \in A\} \subseteq A, \text{ and}$$

$$CHAOS_A? = \{d \mid P = CHAOS_A \wedge d?x \in A\} \subseteq A.$$

In this thesis, we do not model a domain as $CHAOS$.

(C). According to definition (a) and (b), and the semantics of $c!e \rightarrow P$,

$$(c!e \rightarrow P)! = \{d \mid d!v \in \alpha(c!e \rightarrow P)\} = \{c\} \cup P!, \text{ and}$$

$$(c!e \rightarrow P)? = \{d \mid d?x \in \alpha(c!e \rightarrow P)\} = P?.$$

Therefore,

$$(c!e \rightarrow P)! \cap (c!e \rightarrow P)?$$

$$= (\{c\} \cup P!) \cap P?$$

$$= (\{c\} \cap P?) \cup (P! \cap P?)$$

$$= (\{c\} \cap P?) \cup \{\}$$

$$= \{c\} \cap P?.$$

In order to make it an empty set, $\{c\} \cap P?$ needs to be empty, in other words, $c \notin P?$ is the restriction we need for $c!e \rightarrow P$.

(D). Similar to (C), $c \notin P!$ is the restriction we need for $c?x \rightarrow P$.

(E). According to definition (a) and (b), and the semantics of $P \sqcap Q$,

$$(P \sqcap Q)! = \{d \mid d!v \in \alpha(P \sqcap Q)\} = P! \cup Q!, \text{ and}$$

$$(P \sqcap Q)? = \{d \mid d?x \in \alpha(P \sqcap Q)\} = P? \cup Q?.$$

Therefore,

$$(P \sqcap Q)! \cap (P \sqcap Q)?$$

$$= (P! \cup Q!) \cap (P? \cup Q?)$$

$$= ((P! \cup Q!) \cap P?) \cup ((P! \cup Q!) \cap Q?)$$

$$= (P! \cap P?) \cup (Q! \cap P?) \cup (P! \cap Q?) \cup (Q! \cap Q?)$$

$$= \{\} \cup (Q! \cap P?) \cup (P! \cap Q?) \cup \{\}$$

$$= (Q! \cap P?) \cup (P! \cap Q?).$$

In order to it an empty set, $(Q! \cap P?) = \{\} \wedge (P! \cap Q?) = \{\}$ is the restriction we need for $P \sqcap Q$.

(F). Similar to (E), $(Q! \cap P?) = \{\} \wedge (P! \cap Q?) = \{\}$ is the restriction we need for $P \sqcap Q$.

(G). Similar to (E), $(Q! \cap P?) = \{\} \wedge (P! \cap Q?) = \{\}$ is the restriction we need for $P \parallel Q$.

(H). According to definition (a) and (b), and the semantics of $P \setminus c$,

$(P \setminus c)! = \{d \mid d!v \text{ appears in } P \setminus c\} = P! \setminus \{c\}$, and

$(P \setminus c)? = \{d \mid d?x \text{ appears in } P \setminus c\} = P? \setminus \{c\}$.

Therefore (U is the universal set),

$$\begin{aligned}
 & (P \setminus c)! \cap (P \setminus c)? \\
 &= (P! \setminus \{c\}) \cap (P? \setminus \{c\}) \\
 &= P! \cap (P \setminus c)! \cap (P \setminus c)? \\
 &= (P! \setminus \{c\}) \cap (P? \setminus \{c\}) \\
 &= P! \cap (U \setminus \{c\}) \cap P? \cap (U \setminus \{c\}) \\
 &= P! \cap P? \cap (U \setminus \{c\}) \\
 &= \{\} \cap (U \setminus \{c\}) \\
 &= \{\}.
 \end{aligned}$$

There is no need to restrict $P \setminus c$.

(I). According to definition (a) and (b), and the semantics of $\mu X.F(X)$,

$(\mu X.F(X))! = \{d \mid d!v \in \alpha(\mu X.F(X))\} = F!$, and

$(\mu X.F(X))? = \{d \mid d?x \in \alpha(\mu X.F(X))\} = F?$.

Therefore,

$$\begin{aligned}
 & (\mu X.F(X))! \cap (\mu X.F(X))? \\
 &= F! \cap F? \\
 &= \{\}.
 \end{aligned}$$

There is no need to restrict $\mu X.F(X)$.

B. DETAILS OF PROBLEM PROGRESSION RULES

B.1 The Reducing through Cause and Effect Rule Class

This rule class generates a new requirement statement by replacing effects with causes, or causes with effect, based on the causal relations identified among events in domain descriptions. We specialise this rule class into two sub-rule classes, namely the effect-to-cause rule class and the cause-to-effect rule class.

The Effect-To-Cause (ETC) Rule Class

Under this sub-rule class, the requirement statement is rewritten so that any occurrence of an effect, say event “... e occurs ...” is replaced by an occurrence of its guarded cause, say “... c occurs and g holds ...”. This rule class contains nine possible cases depending on whether e and c are internal, shared and controlled, or shared and observed by domain D , as shown in Table 5.1.

Each individual working rule is derived from one of the admissible cases in Table 5.1. These working rules are shown in Figure B.1, Figure B.2 and Figure B.3 below.

Note that in Figure B.2, rule ETC(6)a has two possible problem topologies:

1. domain D shares $\{e\}$ and $\{c\}$ with two different domains, i.e., it shares $\{c\}$ with domain D'' , and $\{e\}$ with domain D' ;
2. domain D shares $\{e\}$ and $\{c\}$ with the same domain D' .

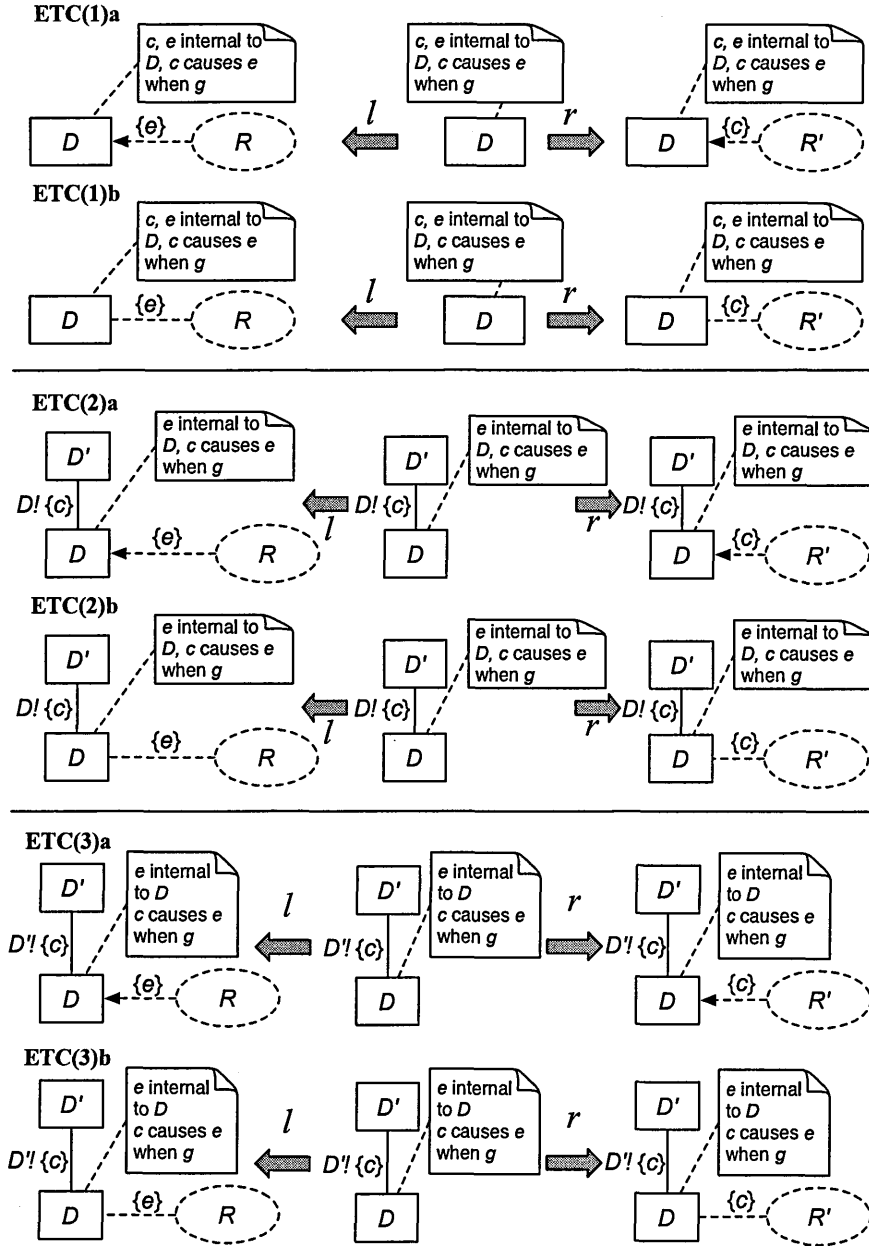


Fig. B.1: Rules ETC(1)a, ETC(1)b, ETC(2)a, ETC(2)b, ETC(3)a, and ETC(3)b, derived from admissible cases (1), (2) and (3) in Table 5.1, respectively

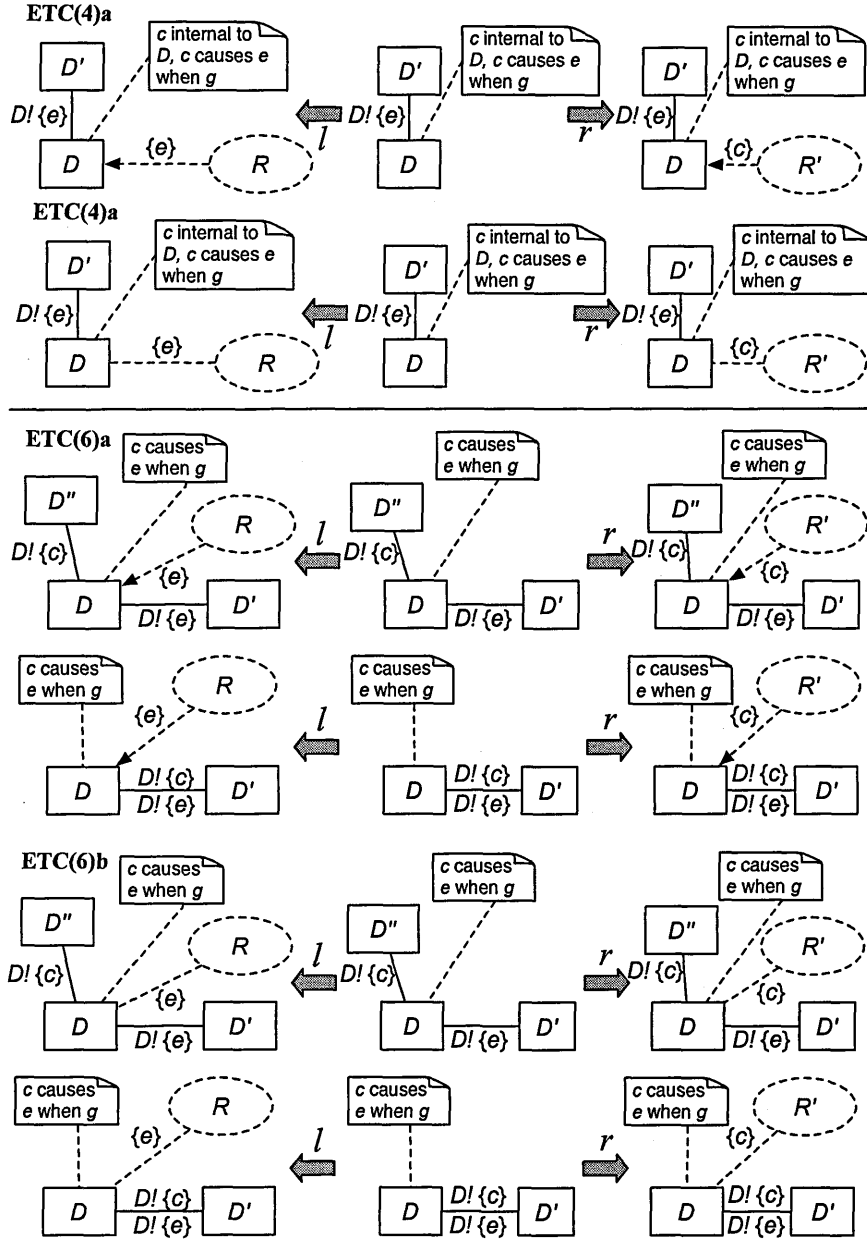


Fig. B.2: Rules ETC(4)a, ETC(4)b, ETC(6)a, and ETC(6)b, derived from admissible cases (4) and (6) in Table 5.1, respectively

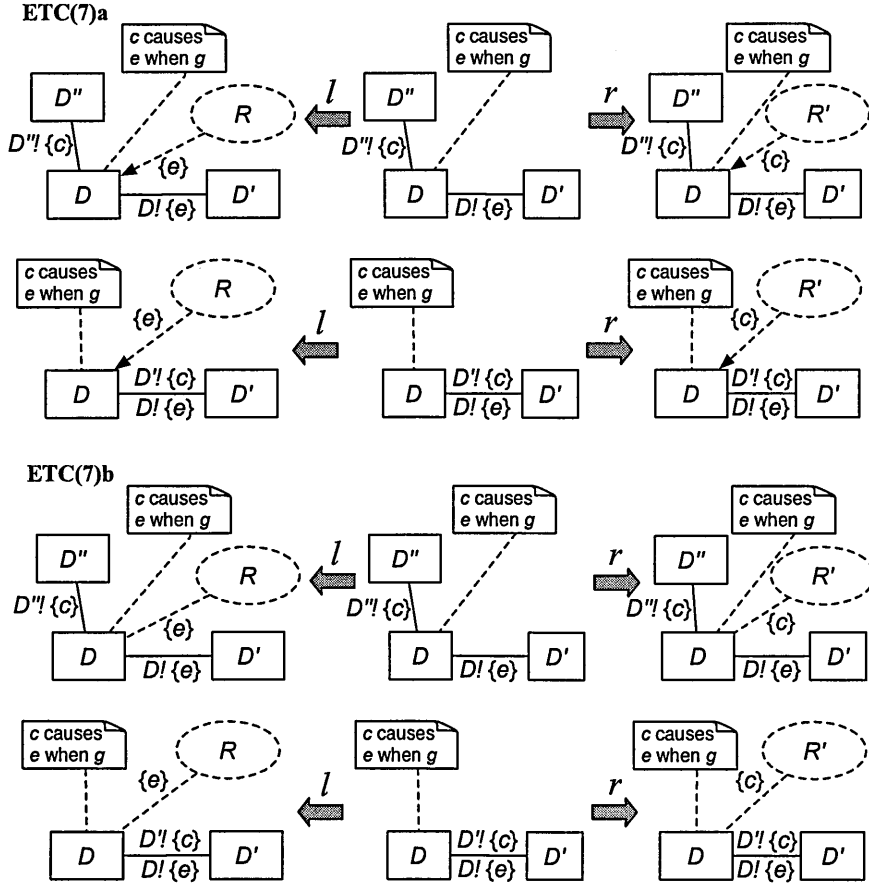


Fig. B.3: Rules ETC(7)a and ETC(7)b, derived from admissible cases (7) in Table 5.1

Since we always draw these diagrams when applying them, there is no need to distinguish them using different rule names (we also preserve our naming convention in this way). For similar reasons, rule ETC(6)b, ETC(7)a and ETC(7)b all have two possible problem topologies.

The Cause-To-Effect (CTE) Rule Class

Under this sub-rule class, the requirement statement is rewritten so that any occurrence of a cause and its conditional guard, say event “... c occurs and g holds ...” is replaced

by an occurrence of its effect, say “... e occurs ...”. This rule class contains nine possible cases depending on whether c and e are internal, shared and controlled, or shared and observed by domain D , as shown in Table 5.2.

Each individual working rule is derived from one of the admissible cases in Table 5.2. These working rules are shown in Figure B.4, Figure B.5 and Figure B.6.

Note that in Figure B.5, rule CTE(6)a has two possible problem topologies:

1. domain D shares $\{e\}$ and $\{c\}$ with two different domains, i.e., it shares $\{c\}$ with domain D'' , and $\{e\}$ with domain D' ;
2. domain D shares $\{e\}$ and $\{c\}$ with the same domain D' .

Since we always draw these diagrams when applying them, there is no need to distinguish them using different rule names (we also preserve our naming convention in this way). For similar reasons, rule CTE(6)b, CTE(7)a and CTE(7)b all have two possible problem topologies.

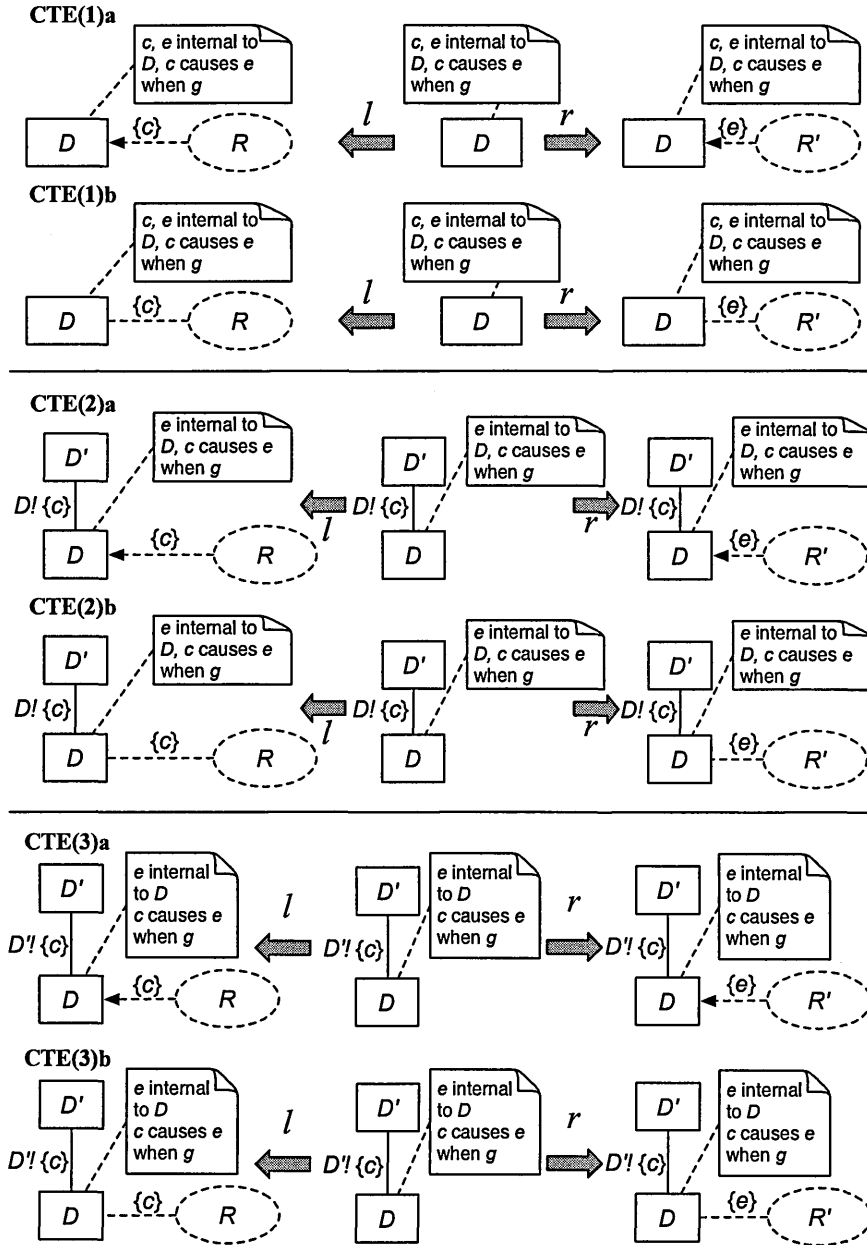


Fig. B.4: Rules CTE (1) a & b, CTE (2) a & b, and CTE (3) a & b, derived from admissible cases (1), (2) and (3) in Table 5.2, respectively

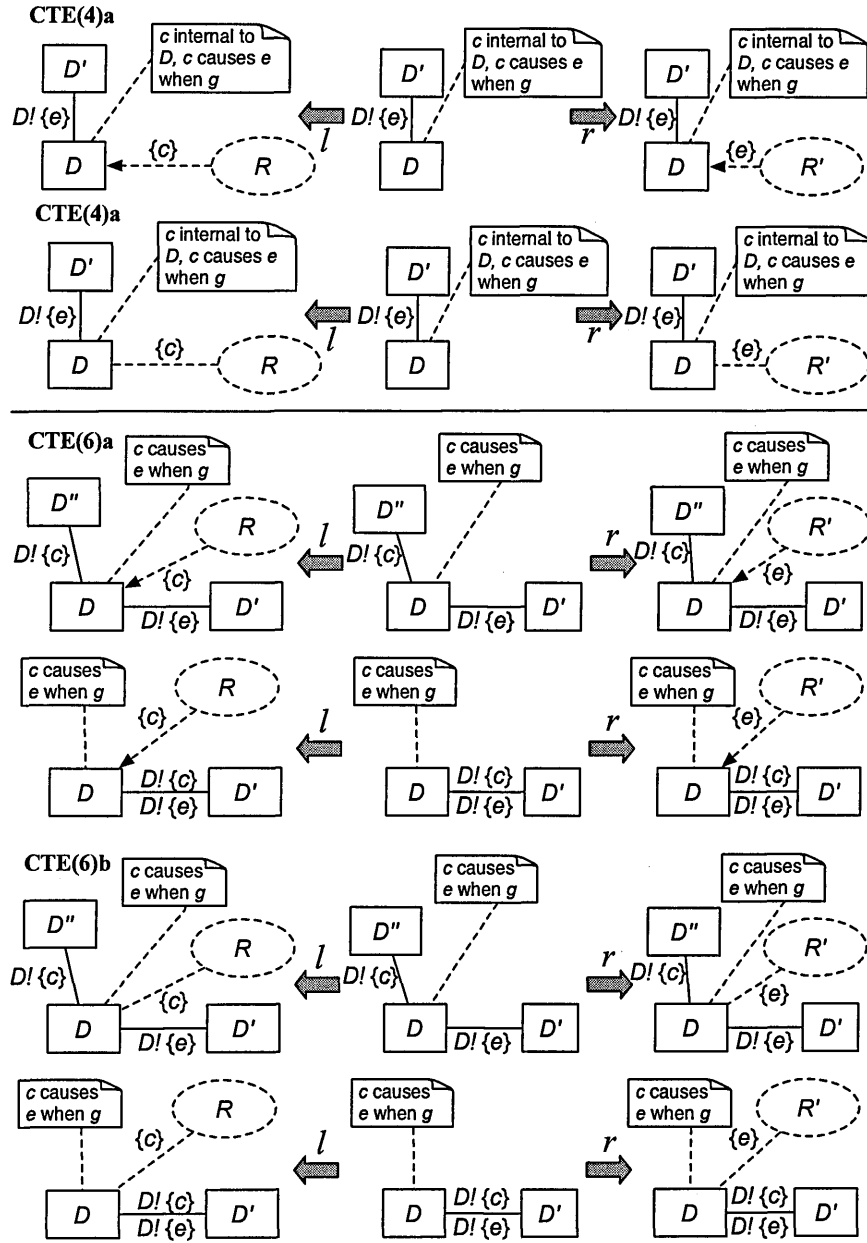


Fig. B.5: Rules CTE (4) a & b, and CTE (6) a & b, derived from admissible cases (4) and (6) in Table 5.2, respectively

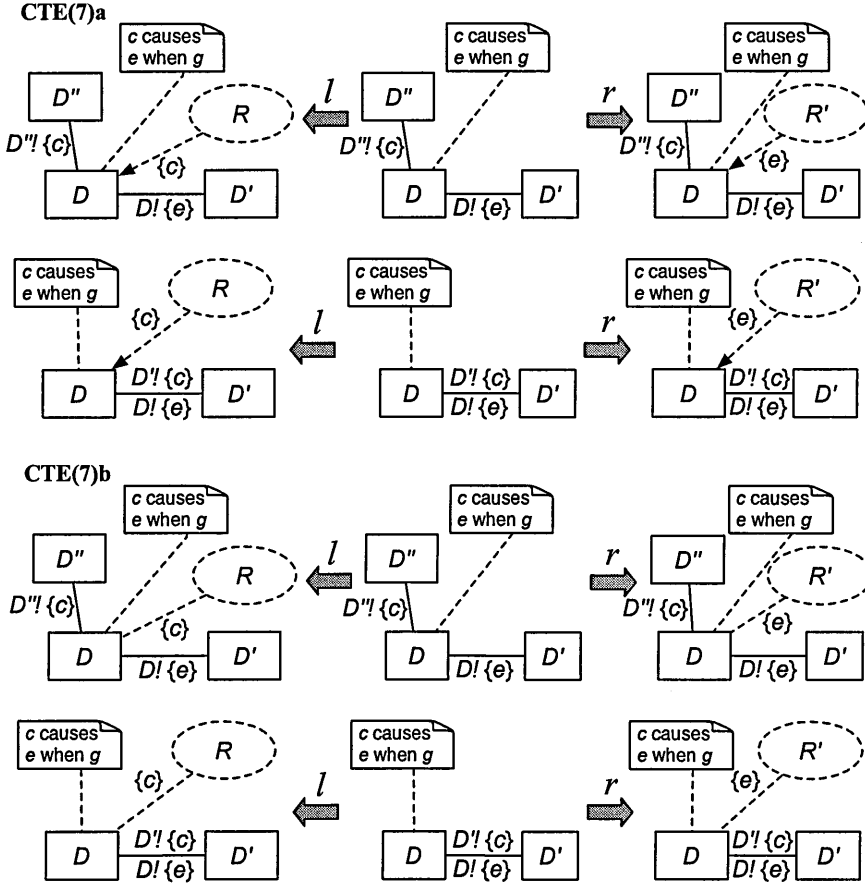


Fig. B.6: Rules CTE (7) a & b, derived from admissible cases (7) in Table 5.2

BIBLIOGRAPHY

- [1] <http://www.objectiver.com/>.
- [2] Software engineering : Report on a conference sponsored by the nato science committee, 7th to 11th october 1968. page 231, Garmisch, Germany, January 1969. Brussels, Scientific Affairs Division.
- [3] I. F. Alexander and N. Maiden, editors. *Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle*. John Wiley and Sons, Ltd., 2004.
- [4] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th international conference on Software engineering*, pages 71–80, 1994.
- [5] T. A. Alspaugh, A. I. Anton, T. Barnes, and B. Mott. An integrated scenario management strategy. In *International Symposium on Requirements Engineering (RE'99)*, pages 142–149, Limerick, Ireland, June 1999.
- [6] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the a7-e aircraft. In *NRL Memorandum Report 3876, Naval Research Laboratory*, Washington DC, August 1992.
- [7] J. S. Anderson and S. Fickas. A proposed perspective shift: Viewing specification design as a planning problem. In *Proc. IWSSD-5, Fifth Int'l Workshop Software Specification and Design*, pages 177–184. IEEE, 1989.

-
- [8] A. I. Anton, W. M. McCracken, and C. Potts. Goal decomposition and scenario analysis in business process reengineering. In *Proc. CAISE'94, Sixth Conf. Advanced Information Systems Eng.*, pages 94–104. Lecture Notes in Computer Science 811, Springer-Verlag, 1994.
 - [9] R.-J. Back and J. von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, 1994.
 - [10] R. M. Balzer, N. M. Goldman, and D. S. Wile. Operational specification as the basis for rapid prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5):3–16, December 1982.
 - [11] L. Baresi and R. Heikel. Tutorial introduction to graph transformation: A software engineering perspective. In *2nd International Conference on Graph Transformation*, Rome, Italy, October 2004.
 - [12] T. E. Bell and T. A. Thayer. Software requirements: Are they really a problem? In *Proceedings of the 2nd international conference on Software engineering*, pages 61 – 68, San Francisco, USA, 1976. IEEE Computer Society Press.
 - [13] S. Berner. About the development of a point of sale system: an experience report. In *25th International Conference on Software Engineering*, pages 528–533, Portland, Oregon, USA, 2003. IEEE Computer Society.
 - [14] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
 - [15] P. Bohm, H. R. Fonio, and A. Habel. Amalgamation of graph transformations: a

- synchronization mechanism. *Journal of Computer and System Science*, 34:377–408, 1987.
- [16] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [17] I. Bray and K. Cox. The simulator: Another elementary problem frame? In B. Regnell and E. Kamsties, editors, *9th International Workshop on Requirements Engineering: Foundation for Software Quality - REFSQ'03*, pages 121–4, Velden, Austria, 2003. Essener Informatik Beitrage.
- [18] S. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.
- [19] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.
- [20] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.
- [21] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering 20th Anniversary Edition*. Addison-Wesley, 1995.
- [22] T. Bui, D. Kersten, and P. C. Ma. Supporting negotiation with scenario management. In *Proc. 29th Hawaii International Conference on System Sciences*, volume III. IEEE Computer Soc. Press, 1993.
- [23] J. M. Carroll. *Scenario-Based Design: Envisioning Work and Technology in System Development*. Wiley, 1995.

-
- [24] J. M. Carroll. Introduction to the special issue on “scenario-based systems development”. *Interacting with Computers*, 13(1):41–42, 2000.
 - [25] J. M. Carroll and M. B. Bosson. Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Transactions on Information Systems*, 10(2):181–212, 1992.
 - [26] J. M. Carroll and M. B. Rosson. Narrowing the specification implementation gap in scenario-based design. In J. M. Carroll, editor, *Scenario-Based Design: Envisioning Work and Technology in System Development*, pages 247–278. John Wiley and Sons, 1995.
 - [27] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *CAiSE 2001*, June 2001.
 - [28] Y. Chen and J. W. Sanders. Weakest specifunctions for bsp. *Parallel Processing Letters*, 11(4):439–454, 2001.
 - [29] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.
 - [30] A. Cockburn. *Structuring Use Cases with Goals*. Human and Technology, 1995.
 - [31] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.
 - [32] A. Corradini, H. Ehrig, M. Loewe, U. Montanari, and F. Rossi. *Algebraic Approaches to Graph Transformation, Part II: Models of Computation in the Double Pushout Approach*.

-
- [33] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Loewe. Algebraic approaches to graph transformation, part i: Basic concepts and double pushout approach. Technical Report TR-96-17, University of Pisa, March 1996.
- [34] A. Corradini and F. Rossi. Hyperedge replacement jungle rewriting for term rewriting systems and logic programming. *Theoretical Computer Science*, 109:7–48, 1993.
- [35] K. Cox, J. G. Hall, and L. Rapanotti. Editorial: A roadmap of problem frames research. *Information and Software Technology*, 47(14):891–902, 2005.
- [36] K. Cox, J. G. Hall, and L. Rapanotti. 1st international workshop on advances and applications of problem frames. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 754–755, Edinburgh, May 2004. IEEE Computer Society Press.
- [37] S. Creese. Industrial strength csp: Opportunities and challenges in model-checking. *5 Years Communicating Sequential Processes*, pages 292–292, 2004.
- [38] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.
- [39] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: An environment for goal-driven requirements engineering. In *Proceedings of 19th International Conference on Software Engineering (ICSE'97)*, pages 612–623, Boston, May 1997.
- [40] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven

- requirements elaboration. In *Proc. FSE'4 - Fourth ACM SIGSOFT Symp. Foundations of Software Eng.*, pages 179–190, San Francisco, October 1996.
- [41] A. M. Davis. The comparison of techniques for the specification of external system behaviour. *Communications of the ACM*, 31(9):1098–1115, 1988.
- [42] E. W. Dijkstra. Guarded commands, non-determinacy and the formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
- [43] M. Dorfman. *Requirements Engineering*. Software Requirements Engineering, Second Edition. IEEE Computer Society Press, 1997.
- [44] E. Dubois, P. Du Bois, and M. Petit. Object-oriented requirement analysis: an agent perspective. In *Proceedings of the ECOOP'93 - Seventh European Conf. Object-Oriented Programming*, pages 458–481. Lecture Notes in Computer Science 707, Springer-Verlag, 1993.
- [45] M. Shaw (editor). Software engineering for 21st century: A basis for rethinking the curriculum. Technical Report CMU-ISRI-05-108, Institute for Software Research International, School of Computer Science, Carnegie Mellon University, February 2005.
- [46] H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 3–14. Springer Verlag, 1987.
- [47] H. Ehrig, P. Bohm, U. Hummert, and M. Lowe. Distributed parallelism of graph

- transformation. In *13th International Workshop on Graph Theoretic Concepts in Computer Science*, volume 314 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, 1988.
- [48] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 1st edition, March 2006.
- [49] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *25th International Conference on Software Engineering*, pages 49–59, 2003.
- [50] S. Faulk, J. Brackett, P. Ward, and J. Kirby. The core method for real-time requirements. *IEEE Software, IEEE Computer Society Press*, 9(5):22–33, 1992.
- [51] S. Faulk, L. Finneran, J. Kirby, S. Shah, and J. Sutton. Experience applying the core method to the lockheed c-130j software requirements. In *The 9th Annual Conference on Computer on Computer Assurance*, pages 3–8, Gaithersburg, MD, June 1994.
- [52] M. S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2):198–234, 1987.
- [53] S. Fickas and R. Helm. Knowledge representation and reasoning in the design of composite systems. *IEEE Trans. on Software Engineering*, pages 470–482, June 1992.
- [54] O. Gotel and A. Finkelstein. An analysis of the requirements traceability prob-

-
- lem. In *First International Conference on Requirements Engineering*, pages 94–101. IEEE Comp Society Press, 1994.
- [55] S. Greenspan, J. Mylopoulos, and A. Borgida. On formal requirements modeling languages: Rml revisited. In *Proceedings of the 16th International Conference on Software Engineering*, pages 135 – 147, Sorrento, Italy, 1994. IEEE Computer Society Press.
- [56] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 2000.
- [57] A. Habel, J. Muller, and D. Plump. Double-pushout approach with injective matching. In H. Ehrig et al., editor, *Graph Transformation*. Springer Verlag, Lecture Notes in Computer Science 1764, 2000.
- [58] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.
- [59] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *RE'02*, pages 137–144, Essen, Germany, 2002. IEEE Computer Society Press.
- [60] J. G. Hall and L. Rapanotti. A reference model for requirements engineering. In *11th IEEE Int. Conf. on Requirements Engineering*, pages 181–187, Monterey, California, 2003. IEEE.
- [61] J. G. Hall and L. Rapanotti. Problem frames for socio-technical systems. In J. Mate and A. Silva, editors, *Requirements Engineering for Sociotechnical Systems*. Idea Group, Inc., 2004.

-
- [62] J. G. Hall, L. Rapanotti, K. Cox, and Z. Jin. 2nd international workshop on advances and applications of problem frames. In *Proceedings of 28th International Conference on Software Engineering (ICSE 2006)*. ACM Press, May 2006.
- [63] J. G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Software and Systems Modeling*, 4(2):189–198, May 2005.
- [64] J. G. Hall, L. Rapanotti, and M. A. Jackson. Problem-oriented software engineering. Technical report, Department of Computing, The Open University, Walton Hall, Milton Keynes, UK, October 2006.
- [65] M. Harrison and P. Barnard. On defining requirements for interaction. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 50–54, San Diego, CA, USA, 1993.
- [66] P. Haumer. *Requirements Engineering with Interrelated Conceptual Models and Real World Scenes*. PhD thesis, 2000.
- [67] C. A . R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [68] C. A . R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [69] C. A . R. Hoare and J. He. The weakest prespecification i, ii. *Fundamenta Informatica*, 9:51–84, 217 252, 1986.
- [70] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

-
- [71] T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of ACM*, pages 63–69, Jan 2003.
- [72] T. Hoare and J. Misra. Verified software: theories, tools, experiments vision of a grand challenge project. *The EASST (European Association of Software Science and Technology) newsletter*, 11:5–30, December 2005.
- [73] G. Hommel. Vergleich verschiedener spezifikationsverfahren am beispiel einer paketverteilanlage. Technical report, Kernforschungszentrum Karlsruhe GmbH, August 1980.
- [74] B. Hopkins. *Causality and development: Past, present and future*, chapter 1, pages 1–17. John Benjamins Publishing Company, Lancaster University, 2004.
- [75] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis. *IEEE Software*, 12(2):33–41, 1994.
- [76] <http://dictionary.oed.com/>.
- [77] <http://www.fsel.com/>. Formal systems (europe) ltd.
- [78] INMOS. *occam 2.1 Reference Manual*. SGS-THOMSON Microelectronics Ltd., May 1995.
- [79] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, April 2006.
- [80] D. Jackson and M. Jackson. Problem decomposition for reuse. *Software Engineering Journal*, 11(1):19–30, 1996.

-
- [81] M. Jackson. Problems, descriptions and objects. In *OOIS'94: 1994 International Conference on Object Oriented Information Systems*, pages 25–35. Springer Verlag, 1994.
- [82] M. Jackson. *Software Requirements & Specifications: A lexicon of principles, practices and prejudices*. ACM Press. Addison-Wesley Publishing Company, 1995.
- [83] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Publishing Company, 2001.
- [84] M. Jackson. Problem frames and software engineering. *Elsevier IST special issue on the 1st International Workshop on Advances and Applications of Problem Frames*, 47(14):903–912, 2005.
- [85] M. Jackson. The world and the machine (keynote). In *17th Int. Conf. on Software Engineering (ICSE'95)*, pages 282–292, Seattle, USA, April 1995. IEEE/ACM.
- [86] M. Jackson. Three aspects of requirements engineering. Course M883, Computing Research Centre, The Open University, October 2005.
- [87] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings of the 17th international conference on Software engineering (ICSE'95)*, pages 15–24, Seattle, Washington, United States, 1995. ACM Press.
- [88] M. A. Jackson. Software development method. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C A R Hoare*. Prentice-Hall International, 1994.
- [89] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

-
- [90] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, Reading, Mass., 1999.
 - [91] W. Johnson. Overview of the knowledge-based specification assistant. In *Proceedings 2nd Knowledge-Based Software Assistant Conference*, 1987.
 - [92] W. Johnson. Deriving specifications from requirements. In *Proceedings ICSE'1988*, pages 428–438. IEEE CS Press, 1988.
 - [93] P. N. Johnson-Laird. *The Computer and the Mind: An Introduction to Cognitive Science*. Cambridge MA: Harvard University Press, 1988.
 - [94] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, April 2006.
 - [95] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, NJ., 1964.
 - [96] M. Korff. Graph-interpreted graph transformations for concurrent object-oriented systems. In *5th International Workshop on Graph Grammars and their Application to Computer Science*, 1994.
 - [97] J. Kramer. Distributed software engineering. In *Proceedings of the 16th international conference on Software engineering ICSE'94*, pages 253–263. IEEE Computer Society Press, 1994.
 - [98] H. J. Kreowski. Is parallelism already concurrency? part 1: Derivations in graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd International Workshop on Graph-Grammars and Their Application*

- to *Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 343–360. Springer Verlag, 1987.
- [99] H. J. Kreowski and A. Wilharm. Is parallelism already concurrency? part 2: Non-sequential processes in graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 361–377. Springer Verlag, 1987.
- [100] L. Lai. The decomposition of concurrent systems. Transfer of status report, p.r.g., o.u.c.l., Oxford University, 1987.
- [101] L. Lai and J. W. Sanders. A weakest-environment calculus for communicating processes. Research report PRG-TR-12-95, Programming Research Group, Oxford University Computing Laboratory, 03-1995 1995.
- [102] R. D. Landtsheer, E. Letier, and A. van Lamsweerde. Deriving tabular event-based specifications from goal-oriented requirements models. *Journal of Requirements Engineering*, 9(2):104–120, 2004.
- [103] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 1st edition, 1998.
- [104] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 2nd edition, 2002.
- [105] B. Lawrence, K. Wiegers, and C. Ebert. The top risks of requirements engineering. *IEEE Software*, 2001.

-
- [106] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *SIGSOFT 2002/FSE-10*, Charleston, SC, USA, November 2002.
- [107] L. Lin, B. Nuseibeh, D. Ince, and M. Jackson. Using abuse frames to bound the scope of security problems. In *Proceedings of the 12th IEEE International Requirements Engineering Conference*. IEEE Computer Society, 2004.
- [108] N. Maiden. Crews-savre: Scenarios for acquiring and validating requirements. *Automated Software Engineering*, 5(4):419–446, 1998.
- [109] S. P. Miller and K. F. Hoech. Specifying the mode logic of a flight guidance system in core. Technical report, Technical Report WP97-2011, Rockwell Collins, Cedar Rapids, IA, August 1997.
- [110] S. P. Miller and A. C. Tribble. Extending the four-variable model to bridge the system-software gap. In *The 20th Digital Avionics Systems Conference (DASC'01)*, 2001.
- [111] J. D. Moffett, J. G. Hall, A. Coombes, and J. A. McDermid. A model for a causal logic for requirements engineering. *Journal of Requirements Engineering*, 1(1):27–46, 1996.
- [112] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1994.
- [113] J. Mylopoulos and J. Castro. Tropos: A framework for requirements-driven software development. In J. Brinkkemper and A. Solvberg, editors, *Information*

- Systems Engineering: State of the Art and Research Themes*. Lecture Notes in Computer Science, Springer Verlag, June 2000.
- [114] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, 2001.
- [115] B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. *The Future of Software Engineering*, ACM, 2000.
- [116] OMG. Unified modeling language (uml), version 2.0.
- [117] P. Padawitz. Graph grammars and operational semantics. *Theoretical Computer Science*, 19:117–141, 1982.
- [118] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.
- [119] J. Peleska. Applied formal methods - from csp to executable hybrid specifications. *25 Years Communicating Sequential Processes*, pages 293–320, 2004.
- [120] A. Perini, P. Bresciani, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Towards an agent oriented approach to software engineering. In *WOA*, pages 74–79. *WOA* 2001, 2001.
- [121] M. Piff. *Discrete Mathematics: an Introduction for Software Engineers*. Cambridge University Press, 1991.
- [122] K. Pohl. *Process Centred Requirements Engineering*. RSP marketed by J. Wiley and Sons Ltd., England, 1996.

-
- [123] C. Potts, K. Takahashi, and A. I. Anton. Inquiry based requirements analysis. *IEEE Software*, 11(2):21–32, April 1994.
- [124] J. Preece, Y. Rogers, and H. Sharp. *Interaction Design: Beyond Human-Computer Interaction*. John Wiley and Sons, 2002.
- [125] L. Rapanotti, J. G. Hall, and M. Jackson. Problem transformations in solving the package router control problem. Technical Report 2006/07, Department of Computing, The Open University, Walton Hall, Milton Keynes, Buckinghamshire, UK, July 2006.
- [126] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *The 12th IEEE International Requirements Engineering Conference (RE'04)*, Kyoto, Japan, 2004. IEEE.
- [127] L. Rapanotti, J. G. Hall, and M. A. Jackson. *Problem Oriented Software Engineering: solving the Package Router Control problem*. Centre for Research in Computing, The Open University, Walton Hall, Milton Keynes, Buckinghamshire, UK, 2007.
- [128] L. Rapanotti, J. G. Hall, and Z. Li. Problem reduction: a systematic technique for deriving specifications from requirements. Technical report, Department of Computing, Centre for Research in Computing, The Open University, UK, February 2006.
- [129] L. Rapanotti, J. G. Hall, and Z. Li. Deriving specifications from requirements through problem reduction. *IEE Proceedings - Software*, 153(5):183–198, October 2006.

-
- [130] B. Regnell, M. Andersson, and J. Bergstrand. A hierarchical use case model with graphical representation. In *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems*, March 1996.
- [131] J. Robertson. On setting the context - some notes. *The Atlantic Systems Guild Inc.*, 2006.
- [132] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.
- [133] S. Robertson and J. Robertson. *Volere requirements: How to get started*. 2004.
- [134] G. F. C. Rogers. *The Nature of Engineering: A Philosophy of Technology*. Palgrave Macmillan, 1983.
- [135] G-C. Roman. Specifying software/hardware interactions in distributed systems. In *Proceedings of the 9th international conference on Software Engineering ICSE '87*, pages 126–139, 1987.
- [136] D. Rosca, S. Greenspan, M. Febloviitz, and C. Wild. A decision marking methodology in support of the business rules lifecycle. In *Pro. RE'97 - Third Int'l Symp. Requirements Eng.*, pages 236–246, Anapolis, 1997.
- [137] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
- [138] K. S. Rubin and J. Goldberg. Object behaviour analysis. *Comm. ACM*, 35(9):48–62, September 1992.
- [139] J. Rumbaugh. Getting started: Using use cases to capture requirements. *Journal of Object-Oriented Programming*, September 1994.

-
- [140] J. Rumbaugh, M. Blaha, W. Prmerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.
- [141] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The MOdelling and Analysis of Security Protocols: The CSP Approach*. Addison-Wesley, 2000.
- [142] J. Ryser and M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. In *12th International Conference on Software and Systems Engineering and their Applications ICSSEA'99*, Paris, France, 1999.
- [143] USA San Diego, editor. *First IEEE International Symposium on Requirements Engineering (RE'93)*. IEEE Comp Society Press, January 4-6, 1993.
- [144] G. Schneider and J. P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1998.
- [145] H. J. Schneider. Describing distributed systems by categorial graph grammars. In *15th International Workshop on Graph-theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 121–135. Springer Verlag, 1990.
- [146] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, Ltd., 2000.
- [147] B. S. W. Schroder. *Ordered sets: an introduction*. Birkhauser, 2003.
- [148] R. Seater and D. Jackson. Problem frame transformations: deriving specifications from requirements. In *Proceedings 2nd International Workshop on Advances and Applications of Problem Frames*, Shanghai, China, 2006.

-
- [149] M. Shaw. Writing good software engineering research papers. In *25th International Conference on Software Engineering (ICSE 2003)*, pages 726–736. IEEE Computer Society, 2003.
- [150] D. R. Smith. Comprehension by derivation. In *13th International Workshop on Program Comprehension IWPC*, pages 3–9, May 2005.
- [151] I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley and Sons, 1997.
- [152] StandishGroup. The chaos report. Technical report, The Standish Group International, Inc, 1994.
- [153] A. Sutcliffe. A technique combination approach to requirements engineering. In *Proc. RE'97 - Third Int'l. Symp. Requirements Eng.*, pages 65–74, Anapolis, 1997.
- [154] A. Sutcliffe. Scenario-based requirements engineering. In *RE 2003 Mini-tutorial*, 2003.
- [155] A. G. Sutcliffe. *User-Centred Requirements Engineering*. Springer-Verlag, London, 2002.
- [156] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.
- [157] W. M. Turski. And no philosophers' stone, either. *Information Processing* 86, pages 1077–1080, 1986.
- [158] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. 2000.

-
- [159] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *5th IEEE International Symposium on Requirements Engineering*, pages 249–263, Toronto, August 2001.
- [160] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. on Software Engineering*, 24(12):1089–1114, 1998.
- [161] W. G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. The Johns Hopkins University Press, Baltimore, paperback edition, 1993.
- [162] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenario usage in system development: A report on current practice. *IEEE Software*, 15(2):34–45, 1998.
- [163] www.wikipedia.org. Soft goal.
- [164] www.wikipedia.org. Use case.
- [165] E. Yu. i* website.
- [166] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, pages 226–235, Washington D.C, USA, Jan 6-8, 1997.
- [167] E. Yu and J. Mylopoulos. Why goal-oriented requirements engineering. In E. Dubois, A.L. Opdahl, and K. Pohl, editors, *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*, pages 15–22, Pisa, Italy, 1998. Presses Universitaires de Namur.

-
- [168] P. Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.
- [169] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.
- [170] P. Zave and M. A. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4), October 1993.