Decomposition for Compositional Verification

Genehmigte Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Dipl.-Inform. Björn Metzler

Paderborn, im Mai 2010

Mitglieder der Promotionskommission:

- Prof. Dr. Heike Wehrheim (Vorsitzende, Gutachterin)
- Prof. Dr. Steve Schneider (Gutachter)
- Prof. Dr. Gitta Domik
- Prof. Dr. Wilhelm Schäfer
- Dr. Matthias Tichy

Die Dissertation wurde am 15. Januar 2010 bei der Fakultät für Elektrotechnik, Informtik und Mathematik der Universität Paderborn eingereicht und am 27. April 2010 vor der Promotionskommission verteidigt und durch die Fakultät angenommen.

Abstract

Within the domain of safety-critical systems, software engineering becomes a major challenge, as failures of a system may have life-threatening ramifications. In order to ensure the reliability of software, its correctness is essential. For the correctness proof of a model, integrated formalisms with an underlying formal semantics can be used.

Several obstacles complicate a successful application of model checking software models. The main challenge is to cope with the *state explosion problem*, that is, the exponential growth of the system's state space in the size of the model. Several approaches deal with this well-known problem. One of them is *compositional verification*.

The basic idea of compositional verification is that the check of correctness of a complex system can be divided into smaller verification tasks. The technique avoids to build up the entire state space of the model, as it solely needs to deal with the individual state spaces of the single components of a system.

In order to facilitate an application of this technique, two problems need to be addressed: the model itself must be *assembled from several components* which is, in general, not the case. Furthermore, an application of compositional reasoning must provide an *efficiency advantage* over monolithic model checking.

Within this thesis, we develop a technique on how to *decompose* software models specified in the integrated formalism *CSP-OZ*. Such a decomposition results in two components suitable for the application of compositional reasoning.

A first challenge is posed by a *proof of correctness*, showing the equivalence of the original specification and a decomposition in our semantic domain. In order to achieve this, we carry out a *dependence analysis* by means of a specification's dependence graph. The analysis leads to a set of *correctness criteria*, based on which the graph is fragmented into two parts. The fragmentation then results in the decomposition of the specification. In addition, we introduce several techniques and algorithms to restore the specification's original control flow and its data flow.

As a second challenge, we address the *practicability* of compositional reasoning: we identify *heuristics* for measuring the quality of a valid decomposition. Here, we *neglect* inefficient decompositions. This allows us to consider only those, which most likely result in an *effective* compositional verification.

Overall, our approach facilitates a general application of compositional reasoning, as it does not rely on systems composed of several components. Moreover, valid decompositions, which are assessed as good by our heuristics, are beneficial for a compositional verification.

The whole approach is tool-supported due to an integration into a graphical modelling environment, allowing for the *modelling*, *analysis*, *decomposition* and *(compositional) verification* of integrated specifications. Model checking itself is performed within an *assume-guarantee-based* verification framework. Here, we use two proof rules, which are shown to be valid in our semantic domain. Along with this, we provide several case studies and experimental results.

Zusammenfassung

Die Softwareentwicklung im Bereich von sicherheitskritischen Systemen stellt eine große Herausforderung dar, da Systemfehler lebensgefährliche Konsequenzen haben können. Die Korrektheit von Software ist essentiell, um ihre Verlässlichkeit zu garantieren. Für den Korrektheitsbeweis eines Softwaremodells eignen sich integrierte Formalismen, welchen eine formale Semantik zu Grunde liegt.

Das Model Checking von Softwaremodellen wird durch verschiedene Hindernisse erschwert. Die größte Herausforderung ist die Bewältigung der *Zustandsexplosion*, des exponentiellen Wachstums des Zustandsraums mit der Größe des betrachteten Systems. Eine Reihe von Techniken beschäftigt sich mit diesem populären Problem, unter anderem die *kompositionelle Verifikation*.

Die grundlegende Idee bei der kompositionellen Verifikation ist die Zerlegung des Korrektheitsbeweises in Teilaufgaben. Diese Methodik vermeidet die Konstruktion des Zustandsraums des gesamten Systems, stattdessen werden die Zustandsräume der einzelnen Systemkomponenten betrachtet.

Die Anwendbarkeit dieser Technik ist an zwei Voraussetzungen gebunden. Zum einen muss das Softwaremodell *aus mehreren Einzelkomponenten zusammengesetzt sein*, was im Allgemeinen nicht der Fall ist. Des Weiteren muss die Anwendung der kompositionellen Verifikation einen *Effizienzvorteil* gegenüber dem direkten Model Checking erbringen.

Diese Arbeit beschäftigt sich mit der Dekomposition von Softwaremodellen, spezifiziert in dem integrierten Formalismus *CSP-OZ*. Eine solche Zerlegung definiert zwei Komponenten, welche sich für die kompositionelle Verifikation eignen.

Eine erste Herausforderung dieser Arbeit stellt ein *Korrektheitsbeweis* dar, welcher die Äquivalenz der ursprünglichen Spezifikation und einer Dekomposition in der zugrunde liegenden semantischen Domäne zeigt. Dazu wird eine *Abhängigkeitsanalyse* durchgeführt, die auf dem Abhängigkeitsgraphen einer Spezifikation basiert. Diese Analyse führt zu einer Menge von *Korrektheitsbedingungen*, auf deren Basis der Graph in zwei Teile zerlegt wird. Daraus ergibt sich die Dekomposition der Spezifikation. Zusätzlich werden Techniken und Algorithmen zur Wiederherstellung des Kontroll- und Datenflusses der ursprünglichen Spezifikation vorgestellt.

Eine zweite Schwierigkeit betrifft die *Praktikabilität* der kompositionellen Verifikation. Dazu werden in dieser Arbeit *Heuristiken* zur Messung der Qualität einer validen Dekomposition ermittelt, wobei ineffiziente Dekompositionen vernachlässigt werden. Dies erlaubt es, ausschließlich solche Zerlegungen zu betrachten, die eine *effektive* kompositionelle Verifikation in Aussicht stellen.

Insgesamt ermöglicht die beschriebene Technik die Anwendung von kompositioneller Verifikation, da sich der Ansatz nicht nur auf zusammengesetzte Systeme beschränkt. Außerdem sind durch die Heuristiken favorisierte valide Dekompositionen vorteilhaft für die Anwendung der kompositionellen Verifikation.

Für den gesamten Ansatz existiert eine Werkzeugunterstützung. Diese basiert auf einer Integration in eine grafische Modellierungsumgebung, welche die *Modellierung*,

Analyse, *Dekomposition* und *(kompositionelle) Verifikation* von integrierten Spezifikationen erlaubt. Das Model Checking wird im Rahmen eines Frameworks im Kontext des *Assume-Guarantee Beweisverfahrens* durchgeführt. Dabei werden zwei Beweisregeln verwendet, deren Korrektheit gezeigt wird. Schließlich werden einige Fallstudien sowie experimentelle Ergebnisse präsentiert.

Acknowledgments

I am grateful to a number of persons for their support, guidance and patience over the last couple of years.

First of all, I would like to express my profound gratitude to Professor Dr. Heike Wehrheim for the supervision, the everlasting support and the opportunity to write this PhD thesis. She constantly pointed the right direction and helped in a dedicated manner, which is anything but naturally.

I also would like to thank the members of my PhD committee, Professor Dr. Steve Schneider, Professor Dr. Wilhelm Schäfer, Professor Dr. Gitta Domik and Dr. Matthias Tichy for their assistance and invaluable advice.

For the proof reading of this thesis and the English review, sincere thanks to Isabela Anciutti, Dr. Uwe Bubeck, Christian Estler, Dr. Dorina Ghindici, Dr. Andreas Goebels, Nils Timm, Simon Titz and Daniel Wonisch.

In our research group, I enjoyed a very warm and collaborative atmosphere. Here, special acknowledgements to Thomas Ruhroth, who always lent a helping hand and never backed down from assisting on all kinds of problems.

There are several students to whom I am greatly thankful for doing most of the implementation of the thesis' approach within Syspect: Klaus Herbold for implementing the decomposition, Meik Piepmeyer for developing the heuristics-based mass validation, Sebastian Micus for integrating the counterexample analysis and, last but not least, Daniel Wonisch for doing an excellent job in writing a compiler for the translation of Syspect export into CSP_M, integrating `FDR2` into Syspect and developing the learning-based CSPLChecker.

I am also grateful to the members of the research group "Correct System Design" supervised by Professor Dr. Ernst-Rüdiger Olderog. In particular, I would like to thank Johannes Faber, who provided assistance to our extension of Syspect in many aspects, always coming up with helpful ideas or additional features. Furthermore, I am in debt to Ingo Brückner and Sven Linker who theoretically and technically provided the basis for this work by developing the slicing approach. It was a great pleasure to work with Ingo on several papers and discuss our related topics.

Most of all, there are three persons to whom my gratefulness is never-ending: my parents, Peter and Inge Metzler, for taking care of me in so many different aspects of life and for their limitless encouragement and patience. Finally, my heartfelt gratitude to my beloved girlfriend Celina: you are an inspiration to my life and the most warm-hearted and affectionate person I ever met. No words can describe the love and emotions I have for you. Your perpetual care, support and love mean the world to me.

Contents

1 Introduction **1**
- 1.1 A Vision of Correct Software . . . 1
- 1.2 Formal Methods and their Combination . . . 2
- 1.3 Compositional Verification . . . 3
- 1.4 Contributions . . . 5
- 1.5 Thesis Structure . . . 6

2 Background: Integrated Formal Methods **9**
- 2.1 A Survey of (Integrated) Formal Methods . . . 9
- 2.2 The Integrated Formalism CSP-OZ . . . 11
  - 2.2.1 Case Study: Candy Machine . . . 11
  - 2.2.2 Object-Z . . . 16
  - 2.2.3 CSP . . . 20
  - 2.2.4 Semantics of CSP-OZ . . . 24
- 2.3 Dependence Analysis . . . 27
  - 2.3.1 Dependence Analysis for CSP-OZ: Motivation . . . 27
  - 2.3.2 Definition of the Control Flow Graph . . . 29
  - 2.3.3 Definition of the Data Dependence Graph . . . 32
  - 2.3.4 Definition of the Dependence Graph . . . 36

3 Background: Compositional Reasoning **41**
- 3.1 Approaches to the State Space Explosion . . . 42
- 3.2 Compositional Reasoning . . . 43
  - 3.2.1 Assume Guarantee Proof Rules . . . 43
  - 3.2.2 Obstacles to the Application of Assume Guarantee Reasoning . . . 45
  - 3.2.3 Learning for Compositional Verification . . . 45
- 3.3 Assume-Guarantee Reasoning for CSP . . . 47
  - 3.3.1 Application Example: Elevator System . . . 49
  - 3.3.2 Soundness of Assume-Guarantee Proof Rules . . . 50
- 3.4 Related Work . . . 53

4 Decomposition of a Specification **55**
- 4.1 Overview . . . 56
- 4.2 Cut of a Dependence Graph . . . 58
  - 4.2.1 Fragmentation of the Control Flow Graph . . . 58
  - 4.2.2 Correctness Criteria for the Fragmentation . . . 61
  - 4.2.3 Definition of a Cut . . . 66
  - 4.2.4 Candy Machine Revisited: Cut of the Dependence Graph . . . 70

4.3 Decomposing CSP-OZ Specifications . . . 72
4.3.1 Intermediate Definition of the Decomposition . . . 75
4.3.2 Preservation of the Data Dependences . . . 81
4.3.3 Preservation of the Control Flow . . . 86
4.3.4 Renaming for the Decomposition . . . 98
4.3.5 Definition of the Decomposition . . . 100
4.3.6 Candy Machine Revisited: Decomposition . . . 101
4.3.7 Improvement of the Decomposition . . . 103
4.4 Decomposition for the General Case: Number Swapper . . . 106
4.5 Related Work . . . 109

5 Correctness of the Decomposition 111
5.1 Ensuring Correct Synchronisation . . . 113
5.2 Correctness for the CSP Part . . . 119
5.2.1 Properties of the Decomposition: CSP Part . . . 119
5.2.2 Correctness of the Decomposition: CSP part . . . 132
5.3 Correctness for the Object-Z Part . . . 138
5.3.1 Properties of the Decomposition: Object-Z Part . . . 140
5.3.2 Correctness of the Decomposition: Object-Z part . . . 146
5.4 Correctness of the Renaming for the Decomposition . . . 159
5.5 CSP Laws for Parallel Composition . . . 164
5.6 Proof of the Main Theorem . . . 166

6 Finding Reasonable Decompositions 167
6.1 Decomposition Heuristics . . . 168
6.1.1 First Heuristic: Cut Size . . . 169
6.1.2 Second Heuristic: Even Distribution . . . 170
6.1.3 Third Heuristic: Few Transmission . . . 170
6.1.4 Fourth Heuristic: Few Addressing . . . 172
6.2 Evaluation of Decomposition Heuristics . . . 172
6.3 Candy Machine Revisited: Evaluation of Cuts . . . 174
6.4 Case Study: Two Phase Commit Protocol . . . 175
6.5 Discussion . . . 180
6.6 Related Work . . . 181

7 Implementation and Experimental Results 183
7.1 Syspect . . . 184
7.1.1 Class Diagrams . . . 184
7.1.2 State Machines . . . 186
7.1.3 Component Diagrams . . . 187
7.1.4 Export to CSP-OZ . . . 187
7.2 Decomposition Framework for Syspect . . . 188
7.2.1 Decomposition Plug-In . . . 189
7.2.2 Mass Validation . . . 191

7.2.3 Model Checking with FDR2 and the CSPLChecker 192
7.2.4 Counterexample Analysis 196
7.2.5 Overall Workflow 198
7.3 Experimental Results 200
7.3.1 Overview 200
7.3.2 Verification Results for the Candy Machine 201
7.3.3 Verification Results for the Two Phase Commit Protocol 204
7.3.4 Verification Results for the Number Swapper 207
7.3.5 Discussion 207

8 Conclusion **217**
8.1 Summary 217
8.2 Future Work 219

Glossary of Symbols **223**

Bibliography **229**

List of Figures **239**

List of Tables **243**

Index **245**

1 Introduction

Contents

1.1 A Vision of Correct Software 1
1.2 Formal Methods and their Combination 2
1.3 Compositional Verification 3
1.4 Contributions 5
1.5 Thesis Structure 6

1.1 A Vision of Correct Software

Over the last decades, research in Computer Science underwent a major focus change: as hardware and software systems influence our daily lives in many critical and even life-threatening situations, systematic approaches to ensure their quality in terms of correct functionality are essential. Trustworthiness and safety-critical hardware and software are required in many areas such as aerospace manufacturing, the automotive industry and medical care, to mention only a few. The more we depend on these systems, the more confidence we need to have in their reliability.

Software quality assurance (SQA) [Gal04] is an approach to observe the engineering process regarding to the quality of the resulting software. Since weaknesses and errors can be introduced at any given point in the process of software development, they need to be excluded at an early stage of the design process.

One SQA methodology is the model-driven development (MDD) [MDA]: software systems are described as models in some (domain specific) language. For modelling object-oriented systems, the Unified Modelling Language (UML) [BJR99] is the current *de facto* standard.

In order to ensure software quality, techniques for early model analysis have been developed, which makes MDD highly useful. One specific analysis technique is software testing [Xie96], aiming at the detection of errors in the model. Automated testing can be of great benefit if hidden faults can be determined and corrected early in the development process. However, correctness of a program can never be achieved by testing:

> *Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.* *[Dij72]*

Since malfunctions are in many cases unacceptable, errors in critical parts of the system have to be ruled out completely. Limited computing resources make the verification of

large models practically impossible. Therefore, a possible strategy is to verify vital parts of a system complemented by testing the system's functionality and non-critical aspects.

For a system's verification, the model needs to be specified in some mathematical formalism incorporating a well-defined semantics. One particular kind of mathematical-based techniques are formal specification languages (formal methods [CW96]). Based on their precise semantics, they allow for the application of verification techniques. In order to guarantee a reliable system, the developer needs to adhere to the specification, which has to be proven correct.

1.2 Formal Methods and their Combination

An informal description of a software model – such as by using an intuitive description based on natural language – is not sufficient for mathematical-based proof techniques due to its missing formal semantics. Owed to its expressive power, the UML does not have a common mathematical-based representation and is often referred to as a semi formal modelling language. This lack of a precise underlying semantics makes the verification of UML models generally impossible.

Formal methods [CW96] are widely-used as a mathematical-based approach of designing software and hardware systems and they receive considerable attention from the research community. The existence of a well-defined underlying semantics, making a precise analysis of the system technically feasible, is common to all formal languages. There are a lot of different notations and techniques, with all of them holding their specific advantages and disadvantages. In general, formal methods can be classified into several categories for describing different aspects of a system, among them are state-based techniques using set theory and predicate logic such as Z [ISO00] and the B-method [Abr96]. In contrast, process algebras like CSP [Hoa85] and CCS [Mil89], for instance, specify the system's behavioural aspects.

Individual formalisms do not cover all relevant aspects to describe a complex system as a whole. Instead of redefining existing methods moving away from the original intention for a specific method, recent research has shifted to the domain of *integrated* formal methods. Focusing on more than one specific facet, they combine different languages to model different viewpoints of a system within one, well-defined formalism. By defining a common and consistent semantics, these notations incorporate the advantages of each individual formalism. Some examples are the combination of the process algebra CSP with the state-based formalism B into CSP||B [ST02], the formalism Event-B [AH06], a combination of the B method [Abr96] with events, and the method we are focusing on in this thesis, CSP-OZ [Fis97], a combination of CSP with the object-oriented extension of Z, Object-Z [Smi00].

In terms of the overall goal (that is, the verification of a system model), the system has to be specified in an (integrated) formalism and needs to be proven correct against certain requirements of the system. This act is called formal verification.

1.3 Compositional Verification

Besides theorem proving, model checking [CGP99] is the most widespread formal verification method. Given a specification S, specified as a finite state-transition system, and a requirement P, formulated in some logical formalism, model checking fully automatically proves or disproves that the system meets the requirement. This is in general denoted by $S \models P$.

As the complexity of software and hardware systems increases, so does the complexity of its models. The most common and major problem for the applicability of model checking is the state explosion: the size of the software model, represented by a state transition system, exponentially grows with the size and number of its components and data domains. In particular, model checking for integrated specifications needs to deal with the state explosion problem: for instance, the behavioural part of the specification can incorporate concurrency, leading to an exponential blow-up of its branching structure. In addition, its state space can be large or even infinite due to its possibly infinite data types.

In general, building up the full state space of a model is infeasible. In order to allow model checking to scale to complex systems, several techniques to tackle this problem were proposed. To mention some of them, symbolic model checking aims at an efficient representation of the model's state space whereas partial order reduction and data abstraction techniques try to reduce the state space of a model by exploring its concurrency structure and by abstracting from concrete data values, respectively. These techniques complement each other and can be combined.

Amongst these techniques, compositional verification [dRHH$^+$01] is one promising approach: instead of verifying a software model as a whole, the components of the model are analysed separately. The verification results can then be combined into one global result. For an application of this *divide-and-conquer* approach, the system needs to be structured into several (parallel) components. That being the case, different strategies can be applied in order to incrementally prove a system correct without ever building up its full state space.

The main technique of compositional verification is *assume-guarantee reasoning* [FP78, Jon83, MC81], applied to a system usually structured into two components. For a given property P on the overall system composed of S_1 and S_2, both components can be verified separately without building the global state space. In order to do so, an environment assumption A needs to be identified, describing the connection and interdependences between the components. The application of an appropriate proof rule, employing A, yields the correctness of the system with respect to P.

Assume-guarantee reasoning has been researched for more than three decades. Recently, a new strategy to fully automatically generate the assumption [CGP03, BGP03] gave a new impulse to this area of research. The strategy is based on automatic learning, thereby freeing the user from a manual computation of the assumptions used in assume-guarantee reasoning.

However, the technique relies on a given structuring of the system into parallel components. Moreover, the efficiency of this approach depends on several factors: if the

generated assumption is too large or the size of the components is not well-balanced, applying the approach can again lead to large state spaces and even worse verification run-times compared to monolithic (direct) verification. It is essential to think about good decompositions to ensure applicability and scalability of the approach [CAC06].

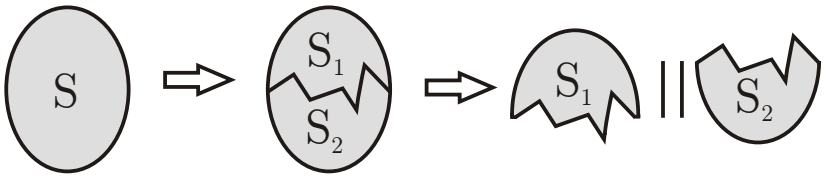


Figure 1.1: Decomposition of a specification S into S_1 and S_2

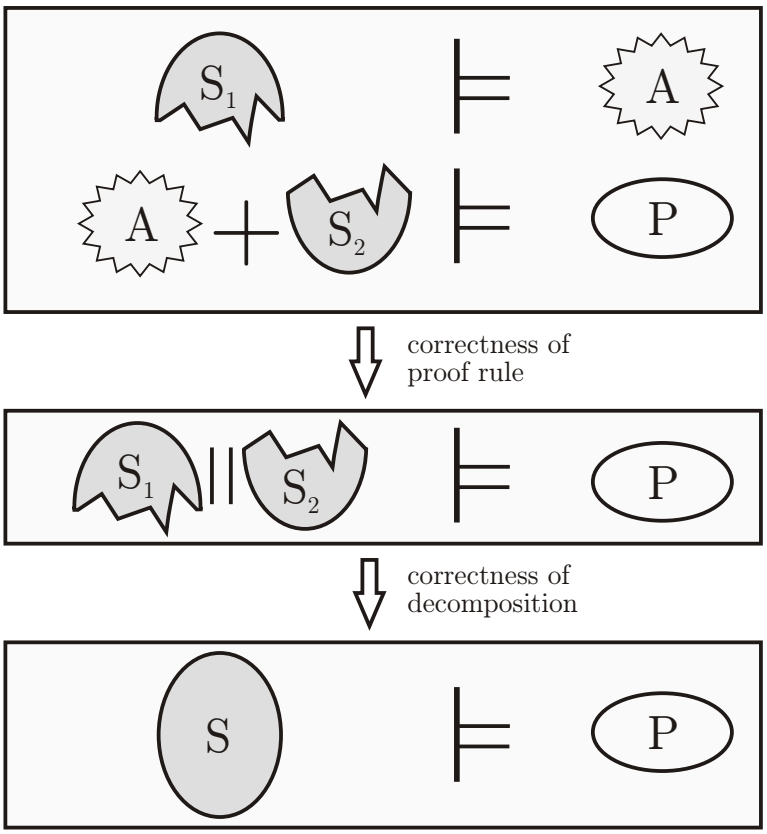


Figure 1.2: Illustration of the overall approach of this thesis

In this thesis, we *construct and evaluate* decompositions of integrated specifications. The starting point is a specification S for which we want to show a specific property P. We

define a set of correctness criteria, serving as the basis for the decomposition of S. Figure 1.1 illustrates the overall idea. The decomposition results in two specification parts, S_1 and S_2. These two parts represent the two parallel components of the decomposed system. An appropriate synchronisation between S_1 and S_2 ensures that the decomposition and the original system are behaviourally equivalent which is subsequently shown in the correctness proof.

S_1 and S_2 then serve as the input for assume-guarantee-based proof rules. The proof rule, as illustrated in Figure 1.2, states the following: if S_1 satisfies an assumption A (described by the symbol '$\models$') and if S_2 satisfies P under the assumption A, then the overall system composed of S_1 and S_2 satisfies P. Correctness of the decomposition yields that S satisfies P, if, and only if, the conclusion of the proof rule can be inferred.

The approach is based on several context-specific heuristics pointing the direction for reasonable decompositions. The technique thus allows for an efficient application of assume-guarantee reasoning. Within our implemented framework for CSP-OZ, we translate the obtained components to the input language of a model checker and ultimately apply the learning-based approach. We are able to evaluate different decompositions by comparing verification run-times with those for monolithic verification.

1.4 Contributions

Compositional verification for integrated formal methods has been researched in [ST04, But09]. These works perform the decomposition of a system by hand and rely on the fact that it can be carried out effectively.

Learning for compositional verification, especially to automate the verification process, was introduced in [CGP03] and further developed in [PGB+08]. The techniques are, however, not applied in the context of formal methods and rely on systems which are already composed of several components.

Alur and Nam [NA06, Nam07] use assume-guarantee-based reasoning in the context of symbolic model checking. They apply the learning framework to automatically generate assumptions and decompose a given system. In addition, they propose heuristics to improve the decomposition process. In their semantic domain of symbolic transition modules solely based on boolean variables, they do not deal with the aspects of integrated formalisms such as data flow, control flow and synchronisation. Furthermore, the developed heuristics only focus on aspects of the learning framework and they do not consider the (dependence structure of the) original system.

The key contribution of this thesis is an approach on how to combine all of these strategies, that is, how to effectively apply compositional verification for integrated formal methods: based on several correctness criteria and certain heuristics, we explicitly decompose the given system. The result of the decomposition serves as the input for the learning-based automated verification process.

Overall, the thesis' contributions are given as follows. We define an approach to decompose specifications written in CSP-OZ. The approach does not rely on systems which are already composed of several processes but, instead, leads to self-defined decompositions.

CATEGORY	CONTRIBUTIONS
Decomposition	✓ Decomposition for integrated specifications. ✓ Exploitation of specification's dependence structure. ✓ Heuristics-based approach to detect reasonable decompositions.
Soundness Proof	✓ Equivalence between original and decomposed system. ✓ Correctness in context of assume-guarantee framework.
Implementation	✓ Integration into graphical modelling framework. ✓ Integration into assume-guarantee-based framework. ✓ Evaluation-based on case studies.

Table 1.1: Contributions of this thesis

In order to achieve reasonable decompositions, we investigate heuristics, exploiting the dependence structure of the specification as well as algorithms for the assumption identification. We present a correctness proof, showing that our decomposition preserves the observable behaviour of the specification. Since the decomposition mandatorily modifies the specification's internal behaviour, the proof incorporates several techniques to link the original system to its decomposition. We integrate the approach along with the learning strategy into a graphical modelling framework for CSP-OZ [Sys06]. An evaluation of the approach is performed based on several case studies and two different learning strategies according to [CGP03] and [BGP03].

1.5 Thesis Structure

This thesis is structured as follows.

Chapter 2 provides an overview of (integrated) formal methods and introduces the employed formalism CSP-OZ [Fis97], a combination of the process algebra CSP [Hoa85], and the state-based formalism Object-Z [Smi00]. The semantics of CSP-OZ and necessary definitions are given. For an illustration of CSP-OZ, we present the running case study of this thesis. Along with this, we provide background on the dependence analysis for CSP-OZ, which serves as the basis for the decomposition approach. The dependence structure of a specification is defined by means of a dependence graph developed in [Brü08], reflecting the control flow of a specification's CSP part as well as data dependences with

respect to the Object-Z part. We present the definition and slightly modify it for our purpose.

Chapter 3 introduces compositional reasoning and starts with an overview on relevant techniques to cope with the state explosion problem in model checking. We survey compositional verification, particularly in the context of integrated formal methods. Afterwards, we present the specific method we deal with in this thesis: the assume-guarantee paradigm. The learning strategy to automatically generate assumptions is introduced next. In order to integrate assume-guarantee reasoning into our setting, we show the correctness of two compositional proof rules in the semantic domain of CSP-OZ. The chapter concludes with a discussion of related work.

Chapters 4-6 are the core chapters of this thesis. In Chapter 4, we introduce our definition for the decomposition of a CSP-OZ specification. We start by defining correctness criteria for a fragmentation of a specification's dependence graph into two parts. Subsequently, we define the decomposition of the specification itself resulting in two specification parts. These parts represent the two parallel components for the employed compositional proof rules. We motivate and describe the employed techniques to guarantee a semantics-preserving decomposition and illustrate the individual steps on the running case study. Finally, we discuss works closely related to our approach.

Chapter 5 presents the correctness proof of our approach. Ultimately, we show that the decomposition does not change the overall semantics of the specification. Several properties, relating the decomposed specification to the original system, are proven. We show that the original specification and the decomposed system, that is, the composition of the two parallel components, are behaviourally equivalent in our semantic domain. Achieving this is done through employing the compositional semantics of CSP-OZ along with the criteria on a correct decomposition.

Chapter 6 describes techniques and heuristics for finding reasonable decompositions. These are the ones for which model-checking-based on our approach will presumably outperform monolithic model checking. We motivate and discuss some context-specific heuristics for good decompositions. Furthermore, we introduce a second, bigger case study, on which we illustrate the application of the heuristics.

Chapter 7 introduces our implementation framework and experimental results. The graphical modelling framework Syspect [Sys06] for modelling CSP-OZ specifications serves as the platform. We describe our integration of the decomposition approach and the integration of the learning framework along with the heuristics-based identification for reasonable decompositions. Additionally, we evaluate our approach on three case studies and discuss the results.

Chapter 8 summarises this thesis, discusses the main results and points out possible topics for future work.

2 Background: Integrated Formal Methods

Contents

2.1 A Survey of (Integrated) Formal Methods 9
2.2 The Integrated Formalism CSP-OZ 11
2.2.1 Case Study: Candy Machine 11
2.2.2 Object-Z 16
2.2.3 CSP 20
2.2.4 Semantics of CSP-OZ 24
2.3 Dependence Analysis 27
2.3.1 Dependence Analysis for CSP-OZ: Motivation 27
2.3.2 Definition of the Control Flow Graph 29
2.3.3 Definition of the Data Dependence Graph 32
2.3.4 Definition of the Dependence Graph 36

The introduction gave a brief overview on the subject area and goals of this thesis. The following two chapters provide the necessary background for the main part of this work. In this chapter, (integrated) formal methods, and in particular CSP-OZ, will be introduced in Sections 2.1 and 2.2. The dependence analysis for CSP-OZ, which serves as the basis for the decomposition, is presented in Section 2.3.

2.1 A Survey of (Integrated) Formal Methods

Model-driven software development aims at the abstract description of a system by specifying a software model in some domain specific language. A model needs to precisely reflect the relevant aspects of the software product to be developed. After an accurate analysis, tools are used to automatically generate code from the model.

The Unified Modelling Language (UML) [BJR99] is undeniable *the* notation to model object-oriented systems in a graphical and intuitive way. The acceptance of the UML as a standard, not only in the academic but also in the industrial field, was not an overnight process. Over many years, researchers defined and evaluated different notations to finally end up with the UML 1.0 proposed in 1997.

Due to the lack of a common precise formal semantics, the UML is not adequate for a rigorous formal analysis. Even though there exist several tools supporting the automated verification of UML diagrams [BGH+05, DWQQ01, BBK+04], they are all restricted to part of the language.

In the perspective to define mathematically-based languages suitable for formal specification and verification, researchers all over the world investigate different techniques

and formalisms. Over the last three decades, a huge amount of formal methods has been developed. In [CW96], Clarke and Wing surveyed the current state of the art. More recently, Bowen [Bow09] set up a *Wiki* [Wik06] used by the formal methods community which gives a detailed overview over many individual formalisms and shows the broad spectrum of research in this area.

Formal methods can be classified into different categories. Mainly, these are behaviour oriented techniques concentrating on the dynamic aspects of a system such as communication, concurrence and control flow, state-based formalisms for the specification of the data and functional aspects and languages to describe hybrid systems which incorporate both, discrete and continuous behaviour.

Behaviour Oriented Formalisms: Among the formalisms to describe behavioural aspects of a system, Petri Nets [Rei85] are a graphical notation to illustrate distributed systems. Process algebras such as CCS [Mil89], CSP [Hoa85] and LOTOS [ISO89] describe concurrent systems by using an algebraic language. Milner also developed the strongly CCS related π-calculus [Mil99]. Another widely used formalism, particularly in the context of the UML, are State Charts [Har87].

State Based Formalisms: The most popular techniques concentrating on the data aspects of a system, that is, describing a system's state space, are Z [Spi92, ISO00], a set theory and first-order-predicate-logic-based formalism, and the Z related B method [Abr96], where B is slightly more low-level and focused on automatic code generation with great success in industrial application [Abr06]. Object-Z [Smi00] is an extension of Z to additionally integrate object-oriented concepts into Z. Event-B [AH06] extends the B method with guarded events. Abstract State Machines [BS03] describe a system's state space and its modifications by using transformation rules and functions.

Formalisms for Hybrid Systems: For the specification of hybrid systems, hybrid automata [ACHH92] combine the description of discrete and continuous behaviour of a system. For the description of continuous real time aspects, in 1994, Alur and Dill developed a real time extension for finite state automata, called timed automata [AD94].

Naturally, different description languages specify different viewpoints of a system. The analysis of large systems thus requires more than one dedicated formalism to reason about different aspects. Many researchers advocating formal methods agree on the statement that there exists no single notation covering all aspects of complex software systems. For this reason, they aim at combining existing, well researched languages, into one consistent new formalism, an *integrated* formal method.

These combinations range from the integration of two or more viewpoints into a single formalism. Combinations of a process calculus with a state-based technique are, for instance, CCS-Z [TA97] combining CCS and Z, the combination of CSP and Z into CSP-Z [MS98], along with CSP||B [TS99], a combination of CSP with the B-method. Fischer [Fis97] integrated CSP and Object-Z into the formalism CSP-OZ.

The integration of time aspects into existing formalisms is, for instance, researched in the context of Timed CSP [Sch99], an integration of real time into CSP. E-LOTOS [ISO01] supplements LOTOS to support time and incorporates a functional-language-based data typing part. In [Hoe06], Hoenicke extended CSP-OZ with the real time interval logic Duration Calculus [ZH04] into CSP-OZ-DC.

The differences between these combinations can also be found in how the new semantics is defined. As an example, Circus [WC02], a combination of CSP, Z and a refinement calculus [SWC02], introduces a new semantics *from scratch*, that is, the semantics of CSP and Z are redefined into a new model, using Hoare's approach of *Unifying Theories of Programs* [HJ98]. Other formalisms, such as CSP||B, keep the original semantics and are thus able to use existing tools.

The following section stepwise introduces the applied formalism, CSP-OZ, illustrated by an example. In order to familiarise this formalism for the core chapters, we introduce the syntax and semantics of CSP-OZ along with necessary definitions and characteristics.

2.2 The Integrated Formalism CSP-OZ

Ever since its introduction in 1978 by Sir Anthony Hoare [Hoa78], the process algebra Communicating Sequential Processes (CSP) draws a lot of attention and is widely used for the specification of concurrent systems. The basic underlying concept is a description of a system by *events* and *processes*: a process defines the communication and interaction aspects by using an underlying alphabet, its set of events.

The state-based Z notation was developed by Jean-Raymond Abrial and others in the late 1970s. By using the concept of *operation schemas*, a Z specification describes the state space and its modifications based on mathematical theory [Spi92]. Smith [Smi00] defined an object-oriented extension of Z, Object-Z.

The integrated formalism we will concentrate on in this thesis is CSP-OZ, a combination of CSP with the object-oriented specification language Object-Z, introduced in [Fis97] and further elaborated on in [Fis00]. In his PhD thesis, Fischer developed the formalism by preserving the original semantics of both, CSP and Object-Z, with the objective to reuse existing theories and tools for both, CSP and Object-Z. In comparison to [Smi00], he introduced a slightly modified notation for Object-Z to which we will refer in this thesis.

We introduce CSP-OZ by means of an example serving as the running case study for this thesis. Afterwards, we give an overview on the syntax and semantics along with required definitions for the incorporated formalisms CSP, Object-Z and CSP-OZ itself.

2.2.1 Case Study: Candy Machine

The following example of a CSP-OZ specification describes a *candy machine* allowing for the payment and collection of several goodies. At first, we define some basic types needed for the specification and start with a free type *Candies* denoting the set of possible candies a customer may order. These are either a chocolate, a cookie or crisps:

$Candies ::= CHOC \mid COOKIE \mid CRISPS$

For simplification, the candy machine only accepts coins with value 1 or 2:

$Coins == \{1, 2\}$

We define a constant identifying the maximal value of all inserted coins, which we set to 5:

$Max == 5$

Next, we give an axiomatic definition for a function determining the price of each of the candies:

$$price : Candies \to \mathbb{N}$$
$$price(CHOC) = 1 \land price(COOKIE) = 2 \land price(CRISPS) = 3$$

In general, a CSP-OZ specification consists of a set of a *classes* which can then be combined to define the overall system. In Chapter 4, we will consider a specification consisting of several classes. As of now, in our running example, we will sufficiently deal with a specification comprising one class only.

S

I	[interface definition]
`main`	[CSP part]
OZ	[Object-Z part]

Figure 2.1: Structure of a CSP-OZ specification

The general structure of a CSP-OZ class named *S* is depicted in Figure 2.1. A class consists of three parts, namely its *interface*, its *CSP part* and its *Object-Z part*. The Object-Z part is again divided into its state schema, initial state schema and its set of operation schemas as shown in Figure 2.2.

S.OZ

`State`	[state schema]
`Init`	[initial state schema]
`enable_`*op*	[`enable`-schemas]
`effect_`*op*	[`effect`-schemas]

Figure 2.2: Structure of the Object-Z part of a CSP-OZ specification

The fundamental concept of CSP-OZ is the connection between CSP part and Object-Z part by using the interface *I* as the common alphabet for both viewpoints of the system: one operation schema of the Object-Z part corresponds to a set of events of the CSP part.

For achieving this correspondence, the interface defines a set of typed channels. A channel declaration has the form

$$\texttt{chan}\ name[p_1 : t_1;\ \ldots p_n : t_n],$$

where *name* identifies the name of the channel and p_i is a decorated parameter of type t_i. We distinguish between three different parameter categories:

Input: Input parameters are decorated with '?' and controlled by the environment of the class. Neither the CSP part nor the Object-Z part can control input parameters. However, the guard of an operation can refer to input parameters, thus allowing the operation to be blocked for a subset of the values of the parameter's type.

Output: Output parameters are decorated with '!' and controlled by the class itself. Predicates of an operation schema can restrict output values. If the operation is executed, the value is determined non-deterministically.

Simple: In contrast to input and output parameters known from CSP and Object-Z, simple parameters are an extension in CSP-OZ and they are in general used for indexing purposes. Simple parameters are undecorated and controlled by the class *and* its environment. They can be restricted by both, the Object-Z part and the CSP part.

Figure 2.3 shows the actual CSP-OZ specification of the candy machine. Here, the interface comprises eight channels. For instance, channel *pay* has one input parameter of type *Coin* modelling the customer's payment. In contrast, channel *deliver* has one output parameter of type *Candies* modelling a goody the machine dispenses. Note that all parameters are inputs to the CSP part since neither of them is a simple parameter. Some channels such as *abort* do not use any parameters.

As already mentioned, the CSP part of the specification describes the *dynamic* behaviour of a system by means of the possible sequences of events and their orderings. This is achieved by a set of *process equations*. As a convention, the initial process of a class' CSP part is named `main`. The remaining set of process equations comprises four process names: *Payout* describes the behaviour of the system if the customer chooses to abort the procedure and collects his money. *Select* models the selection of an item and *Order* its actual ordering. Finally, *Deliver* describes the delivery of the ordered items.

The Object-Z part starts with the class' state schema, containing the set of state variables for the description of the class' state space and its modifications. These are two variables, *sum* and *credits*, of type $\mathbb{N}$ to denote the current sum of money paid by the customer and the remaining credits, respectively. A sequence of coins *paid* models the inserted coins, and a second sequence *items* the previously ordered candies before the actual delivery. Finally, the variable *selected* of type *Candies* describes the current item, selected by the customer.

The initial state schema of the class defines the set of valid initial configurations by using predicates, restricting the values of the state variables. In our example, both

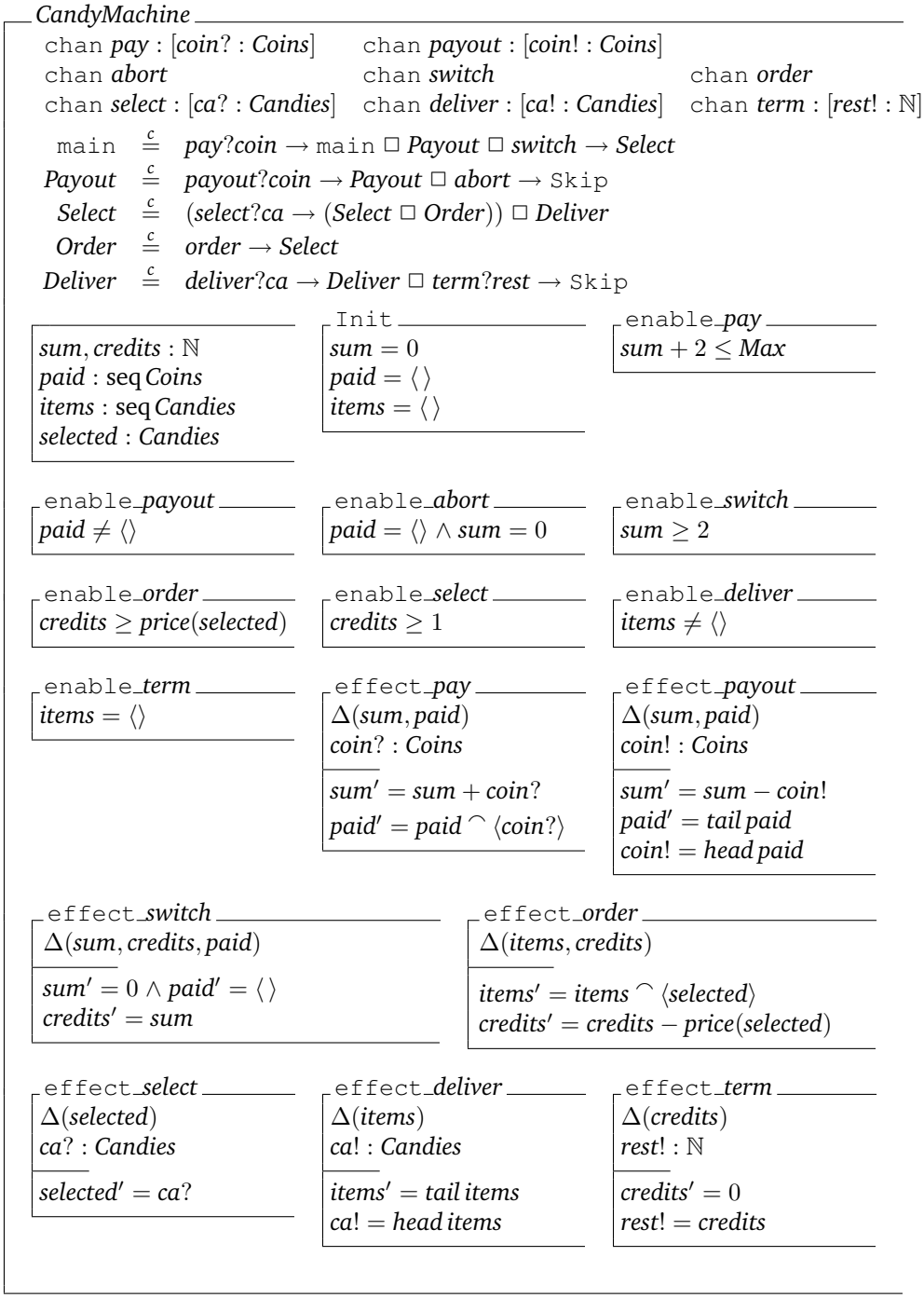


Figure 2.3: Candy machine specification

sequences need to be initially empty, and the initial sum of money is equal to zero. The remaining state variables are not restricted by the initial state schema.

The static behaviour of the class is described in terms of a set of `enable`- and `effect` schemas, conjointly defining the behaviour of an operation schema. An `enable` schema defines the precondition of an operation by again using predicates referring to state variables of the class. An operation can only be executed if its respective precondition is satisfied. Otherwise, the operation is blocked. For instance, operation *deliver* is blocked if there are no items to deliver, that is, $items = \langle\rangle$ holds. Besides, *order* can only be executed if there are enough credits left to pay the price of the selected items.

An `effect` schema defines an operation and how its execution modifies the state space. It starts with its Δ-list, comprising the set of state variables, modified by the operation. All variables not appearing in this list remain unchanged. As an example, the `effect` schema of *payout* modifies the variables *sum* and *paid*. Next, the schema can contain a set of parameter declarations, corresponding to the parameters in the operation's interface declaration. Finally, the predicate part of the schema defines the actual modifications of the state variables. For that purpose, a predicate can refer to the possible values of a state variable *after* execution of the operation; these post-state values are depicted in primed form. For instance, *payout* restricts the post value of *sum*, *sum*', to $sum - coin!$. Thus, the operation ensures that the only possible value of *sum* after execution of *payout* is exactly the original amount of money, reduced by the value of the dispensed coin. Note that the operation *abort* possesses an empty `effect` schema which leaves all variable values unchanged. In this thesis, we will leave out empty schemas.

We will now describe the dynamic behaviour of the class and its state space modifications by clarifying and illustrating its workflow. Figure 2.4 illustrates the CSP part of the specification as a state transition graph, according to the operational semantics of CSP as given in [Ros98].

A customer has three initial options, modelled by the CSP operator $\Box$ for external choice by the environment: first, if the amount of already inserted money increased by two is smaller than *Max*, a user can insert a coin into the machine (*pay*) followed by (using the prefix operator $\rightarrow$) a call of the initial process `main`. The coin and its value are stored in the variables *paid* and *sum*, respectively. Second, the customer can chose to cancel buying candies as described in the process *Payout*, where he repeatedly collects his coins (*payout*) by emptying the *paid* sequence. After a possibly empty sequence of *payout*s, the process is finally *abort*ed and terminates (denoted by `Skip`, the basic CSP process for termination). As a last option, if the user inserted at least coins of an overall value of 2, he can request to process to the ordering of candies (*switch*), for which the process *Select* is called. The customer may now *select* an item which she wants to order. If enough credits are left, the item is *order*ed by storing it in the sequence *items* and reducing the credits by the respective amount. Otherwise, the customer needs to reselect another item. If he ordered at least one item, he can proceed to get his candies *deliver*ed. In this case, the machine dispenses the items one by one in the correct order. The process *term*inates after the potential order and delivery of candies. Remaining spare money is returned.

Next, we clarify some syntactical aspects of Object-Z, CSP and CSP-OZ along with

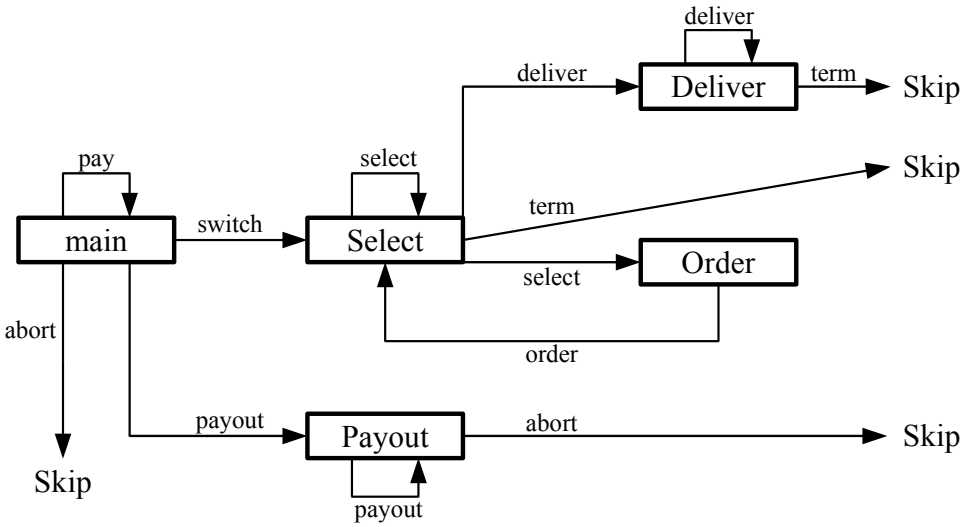


Figure 2.4: Illustration of the CSP part of the candy machine specification

defining their semantics. For more details, we refer to [Smi00, Spi92, ISO00], [Ros98, Sch99] and [Fis00] for comprehensive documentations on (Object-)Z, CSP and CSP-OZ, respectively.

2.2.2 Object-Z

As already explained, we use a sightly adapted version of the Object-Z language as introduced in [Fis00]. Therefore, we will continue to refer to the Object-Z part of a CSP-OZ class specification, denoted by *OZ*, instead of pure Object-Z class specifications.

OZ generally consists of a state schema, an initial state schema and a set of operation schemas, where elements of the latter comprise an `enable` schema and `effect` schema, as depicted in Figure 2.2. The keywords `State` and `Init` denote the state schema and initial state schema of a class, respectively. Thus, *OZ* can be denoted by a tuple:

$$OZ = (\texttt{State}, \texttt{Init}, (\texttt{enable\_}op)_{op \in Op}, (\texttt{effect\_}op)_{op \in Op})$$

In the remainder of this thesis, we denote the sets of all values for input parameters, output parameters and simple parameters of an operation schema *op* by $In(op)$, $Out(op)$ and $Simple(op)$, respectively. Elements of these sets are tuples adhering to the types of the operation parameters. The set *Events* is defined as the set of operation names of *OZ*, completed by values for their parameters:

$$Events = \{op.in.sim.out \mid op \in Op, in \in In(op), sim \in Simple(op), out \in Out(op)\}$$

The state schema `State` defines the state space of *OZ* and comprises the set of *state variables* the class uses along with their types. Additionally, the state schema contains a

(possibly empty) set of *state invariants* – a set of predicates, which have to be satisfied initially as well as for any reachable state of the class. The set of state variables will be referred to as V. A *state* of *OZ* is defined as a valuation of all state variables: for $V = \{x_1, \ldots, x_n\}$, a state s is denoted as the tuple $s = (v_1, \ldots, v_n)$ where v_i are values of x_i within the variable's domain. We write $\texttt{State}(s)$ or, equivalently, $s \in \texttt{State}$, to refer to states of the OZ state space, and we use $s.x_i$ to denote the value of x_i in the state s.

For the definition of our decomposition, we need to project a state $s \in \texttt{State}$ on a subset of the state variables $V' \subseteq V$:

Definition 2.2.1. *(State Projection)*
Let $V = \{x_1, \ldots, x_n\}$, and let $s = (v_1, \ldots, v_n)$ with $s \in \texttt{State}$. We use

$$(v_1, \ldots, v_n) = ((v_1, \ldots, v_{n-1}), v_n).$$

The projection *of s on the set of state variables $V' \subseteq V$, denoted by $s \restriction V'$, is inductively defined as*

$$((\ldots(v_1), \ldots v_{n-1}), v_n) \restriction V' := \begin{cases} (\ldots(v_1), \ldots v_{n-1}) \restriction V', & x_n \notin V', \\ ((\ldots(v_1), \ldots v_{n-1}) \restriction V', v_n), & x_n \in V'. \end{cases}$$

The initial state schema restricts the initial valuation of the state variables. The `enable` schema defines an operation's *guard*. It consists of a declaration part for possible input- and simple parameters (`enable`-schemas must not declare output variables) and a predicate part, containing predicates solely referring to unprimed state variables, that is, to the state *before* the operation took place. If the conjunction of these predicates is not satisfied, the operation is blocked. `enable_`*op* can be interpreted as a predicate, denoted by $\texttt{enable\_}op(s, in, sim)$ with $s \in \texttt{State}$, $in \in In(op)$ and $sim \in Simple(op)$.

An operation's `effect` schema declares the possible *post* states after the operation took place. It consists of a Δ-list, comprising all variables which are modified by the operation. The subsequent declaration part contains the schema's parameters and its predicate part defines the restriction on the post-state. For this, variables denoted in primed form refer to post state values. For any variable x not contained in the Δ-list, $x' = x$ implicitly holds. An `effect` schema can be denoted as the predicate $\texttt{effect\_}op(s, in, sim, out, s')$ with $s \in \texttt{State}$, $in \in In(op)$, $sim \in Simple(op)$, $out \in Out(op)$ and $s' \in \texttt{State}'$.

In the remainder of this thesis, we let $ref(op)$ denote the set of *referenced* variables of an operation (those occurring in unprimed form), whereas $mod(op)$ denotes its set of *modified* variables (those occurring in its Δ-list). In addition, we set $all(op) := ref(op) \cup mod(op)$.

The *precondition* of an `effect`-schema can be defined as

$$\begin{aligned} &\text{pre } \texttt{effect\_}op(s, in, sim) := \\ &\exists\, out \in Out(op), s' \in \texttt{State}' \bullet \texttt{effect\_}op(s, in, sim, out, s') \end{aligned}$$

In this thesis, we assume that $\texttt{enable\_}op(s, in, sim) \Rightarrow \text{pre } \texttt{effect\_}op(s, in, sim)$ holds. This corresponds to the *blocking* view of operations as described in [Fis00]: an operation can only be executed if its precondition is satisfied, otherwise it is blocked.

As an `enable`-schema can always be strengthened such that $\texttt{enable} \wedge \neg \text{pre } \texttt{effect}$ is impossible, this is not a restriction. For instance, consider the following operation *op*:

`enable_op`
$i? : \mathbb{N}$
$x > y$

`effect_op`
$\Delta(x)$
$i? : \mathbb{N}$
$x' = i? \wedge x' > y'$

For any value of $i?$ such that $i? \leq y$ holds, pre `effect_op` is not satisfied. Adding $i? > y$ to `enable_op` schema ensures the previous implication.

When referring to an operation *op*, comprising `enable_op` and `effect_op`, we denote its entire predicate part by *op.pred* whereas the declaration part will be denoted by *op.dec*. In case we need to refer to the delta list of an operation, we write *op.delta*.

Semantics of Object-Z

As we are interested in the sequences of events of a specification, our approach is based on an *operational* semantics for Object-Z and ultimately for CSP-OZ.

For the Object-Z part of a specification, we need to reason about events and states. The decomposition approach analyses a specification's dependence structure. A description of paths solely referring to events is insufficient, since we need to incorporate the state space and its modifications as well.

In order to be precise, execution of an event *op.in.sim.out* within the Object-Z part, changing the before state *s* into the after state s', refers to an operation's `enable`- and `effect`-schema:

$$s \overset{op.in.sim.out}{\longrightarrow} s' \Leftrightarrow (\texttt{enable\_}op(s, in, sim) \wedge \texttt{effect\_}op(s, in, sim, out, s'))$$

The notation we are using is closely related to the Object-Z semantics of [Brü08] which itself is based on the *history semantics* of Object-Z [Smi95]: sequences of state valuations and operation calls describe the possible behaviours.

As a semantic model, we use *labelled transitions systems* (LTS). In order to reason about states of the Object-Z part, a *path* of a labelled transition system is an alternating sequence of states and events.

Definition 2.2.2. *(Labelled Transition System)*
Let E be an alphabet of events. A labelled transition system (LTS) $M = (S, S_0, \rightarrow)$ over E consists of

- *a set of states S,*
- *a set of initial states $S_0 \subseteq S$ and*
- *a transition relation $\rightarrow \subseteq S \times E \times S$.*

A path of an LTS is a finite or infinite sequence $\langle s_0, e_0, s_1, e_1, \ldots \rangle$ alternating between states and events such that $(s_i, e_i, s_{i+1}) \in \rightarrow$ holds for all $i \geq 0$.

Note that paths of LTS *can* be infinite but do not *need* to be infinite. Next, we define the operational semantics of *OZ* in terms of labelled transitions systems.

Definition 2.2.3. *(Labelled Transition System for Object-Z)*
Let OZ be the Object-Z part of a CSP-OZ class specification. The LTS semantics of OZ is defined as the labelled transition system $M^{OZ} = (S, S_0, \rightarrow_{OZ})$*, defined over* $E := Events$*, with*

- $S = \{s \mid s \in \texttt{State}\}$,
- $S_0 = \{s \in S \mid \texttt{Init}(s)\}$,
- $\rightarrow_{OZ} = \{(s, op.in.sim.out, s') \mid \texttt{enable\_}op(s, in, sim) \wedge \texttt{effect\_}op(s, in, sim, out, s')\}$.

The set of all paths of M^{OZ} *is defined as* $Traces(OZ)$*. Moreover, let*

$$traces(OZ) := \{\pi \restriction Events \mid \pi \in Traces(OZ)\}$$

and for $tr \in traces(OZ)$*,*

$$tr \rhd Op := \begin{cases} \langle\rangle, & tr = \langle\rangle \\ \langle op \rangle \frown (tr' \rhd Op), & tr = \langle op.in.sim.out \rangle \frown tr' \end{cases}$$

Finally, for $\pi \in Traces(OZ)$*, let* $\pi[i]$ *denote the i-th state and* $\pi.i$ *the i-th event of* π*.*

For clarification, π denotes a trace within $Traces(OZ)$ distinguishing it from $tr \in traces(OZ)$ not comprising states. We exemplify the definition with our case study:

Example 2.2.4. *The following trace, named* π*, is a valid path of the LTS of the Object-Z part of the candy machine:*

$$\begin{array}{ll}
\langle & \\
(sum = 0, credits = 0, paid = \langle\ \rangle, items = \langle\ \rangle, selected = COOKIE), & pay.2, \\
(sum = 2, credits = 0, paid = \langle 2 \rangle, items = \langle\ \rangle, selected = COOKIE), & switch, \\
(sum = 0, credits = 2, paid = \langle\ \rangle, items = \langle\ \rangle, selected = COOKIE), & select.CHOC, \\
(sum = 0, credits = 2, paid = \langle\ \rangle, items = \langle\ \rangle, selected = CHOC), & order, \\
(sum = 0, credits = 1, paid = \langle\ \rangle, items = \langle CHOC \rangle, selected = CHOC), & deliver.CHOC, \\
(sum = 0, credits = 1, paid = \langle\ \rangle, items = \langle\ \rangle, selected = CHOC), & term.1, \\
(sum = 0, credits = 0, paid = \langle\ \rangle, items = \langle\ \rangle, selected = CHOC) & \\
\rangle &
\end{array}$$

Its projection on events is given by

$$tr = \pi \restriction Events = \langle pay.2, switch, select.CHOC, order, deliver.CHOC, term.1 \rangle.$$

The projection of tr on its set of operation names yields

$$tr \rhd Op = \langle pay, switch, select, order, deliver, term \rangle.$$

$$
\begin{array}{lcll}
P & ::= & \texttt{Skip} & \textit{(Termination)} \\
 & | & \texttt{Stop} & \textit{(Deadlock)} \\
 & | & a \rightarrow P_1 & \textit{(Prefixing)} \\
 & | & P_1 \,\Box\, P_2 & \textit{(External Choice)} \\
 & | & P_1 \sqcap P_2 & \textit{(Internal Choice)} \\
 & | & P_1 \fatsemi P_2 & \textit{(Sequential Composition)} \\
 & | & P_1 \parallel_A P_2 & \textit{(Interface Parallel)} \\
 & | & P_1 \,{}_{A_1}\!\parallel_{A_2} P_2 & \textit{(Alphabetised Parallel)} \\
 & | & P_1 \,|||\, P_2 & \textit{(Interleaving)} \\
 & | & P_1 \setminus A & \textit{(Hiding)} \\
 & | & X & \textit{(Process Call)} \\
 & | & P[\![\mathrm{R}]\!] & \textit{(Renaming)}
\end{array}
$$

Figure 2.5: Simplified grammar of CSP

2.2.3 CSP

In general, a CSP process P is defined over a set of communication events which the process can perform: its alphabet. For this, we need the notion of *channels*. A channel consists of a name and a finite, possibly empty, sequence of data types $T_1 \times \cdots \times T_k$, the *type* of the channel. An event is then composed of the channel name and possible data values, corresponding to the channel's type.

In our example specification of a candy machine, the channel *payout* is of type *Coins*. Thus, $payout.1$ denotes a possible event, communicated by the *CandyMachine* which is composed of the channel name *payout* and the value 1 according to its type.

In this thesis, we will, in general, refer to an alphabet *Events*, denoting a global set of all events which corresponds to the set of events for the Object-Z part. These are comprised of the operation names and values for their parameters. If we want to refer to the distinguished alphabet of a process P, we use the notation αP. Accordingly, we let *Op* denote the set of channel names, corresponding to the set of operation names for the Object-Z part. We use the terms operation and channel synonymously throughout this thesis.

The inductive definition of a CSP process, which we will refer to in this thesis, is summarised in the grammar, given in Figure 2.5. Here, $a \in Events$ denotes an event and $A, A_1, A_2 \subseteq Events$ sets of events.

`Skip` and `Stop` are basic CSP processes for termination and deadlock, respectively. `Stop` does not communicate at all whereas `Skip` solely communicates the reserved event $\checkmark$ to indicate successful termination. The prefix process $a \rightarrow P_1$ communicates the event a and subsequently behaves as P_1. $P_1 \,\Box\, P_2$ describes the external choice (resolved by the environment) between both processes P_1 and P_2 whereas $P_1 \sqcap P_2$ denotes the internal choice (resolved internally). $P_1 \fatsemi P_2$ describes sequential composition meaning that first, P_1 is executed and, if P_1 successfully terminates, P_2 is allowed to occur. $P_1 \parallel_A P_2$ defines the interface parallel composition of two processes, which need to synchronise on all events in A. Similarly, the alphabetised parallel composition $P_1 \,{}_{A_1}\!\parallel_{A_2} P_2$ needs to synchronously perform any events within $A_1 \cap A_2$. We will sometimes leave out the

synchronisation alphabet(s) and denote the parallel composition of two processes by $P_1 \parallel P_2$, if the alphabet is not considered. The interleave process $P_1 \mid\mid\mid P_2$ is a special case of parallel composition where the synchronisation alphabet is empty. $P_1 \setminus A$ behaves similar to P_1, except that events from A are hidden, that is, invisible to the environment. All events within A are renamed to a distinguished, internal event τ. Since CSP processes are defined via process *equations*, X denotes the body, that is, the right hand side, of a process equation. Finally, $P[\![\mathtt{R}]\!]$ depicts the process where all events e occurring in P are renamed to $R(e)$, according to a relation $R : \mathit{Events} \to \mathbb{P}(\mathit{Events})$.

From now on, we let L^{CSP} denote the set of all CSP terms. We introduce some additional generalisations and abbreviations, which we will use in the remainder of this thesis. First, binary operators can be indexed over some finite indexing set I. As an example, the indexed external choice, denoted by

$$\Box_{i \in I} P_i,$$

defines the external choice over all processes P_i with $i \in I$. Similarly, N-way indexed parallel composition can be denoted by

$$\Big|\Big|_{i=1}^{N} P_i.$$

Based on associativity laws, N-way indexed parallel composition can be transformed into a chain of binary parallel compositions. The same holds for the remaining binary operators. Therefore, in the following definitions and proofs, we do not need to deal separately with indexed operators.

The *prefix choice* process $a : A \to P_1$ initially offers any event of A and subsequently behaves as P_1. Prefix choice can be seen as generalisation of prefixing. For finite A, according to [Ros98], prefix choice can equivalently be transformed into indexed external choice based on the equivalence

$$a : A \to P_1 \Leftrightarrow \Box_{a:A}\, a \to P_1$$

The process $\mathtt{Run}_A$ defined as

$$\mathtt{Run}_A \stackrel{c}{=} a : A \to \mathtt{Run}_A$$

can always communicate any member of A. If no alphabet is specified, we assume $A = \mathit{Events}$ and set $\mathtt{Run} := \mathtt{Run}_{\mathit{Events}}$.

Sometimes, it is convenient to refer to the set of events extending a set of channel names with all possible parameter values. This motivates the following definition from [Ros98]:

Definition 2.2.5. *(Extension of channels)*
Let c be a channel of type $T_1 \times \cdots \times T_k$. The extension set of c is defined as

$$\{|\, c \,|\} := \{c.v_1.\ldots.v_k \mid v_i \in T_i\}.$$

The definition allows us to refer to a set of channel names as the synchronisation alphabet, meaning that the extension sets of their operations are synchronised.

A channel includes an *ordering* on its data types. *Partially* defined events fix a (possibly empty) subset of its type while the remaining data values (possibly none) are undetermined. Achieving this is done through using the underscore- (*don't care*-) symbol "_" in order to refer to positions within a channel's type and define:

Definition 2.2.6. *((Extensions of) partial events)*
Let c be a channel of type $T_1 \times \cdots \times T_k$. $c.v_1.\ldots.v_k$ is a partial *event if $v_j \in T_j \cup \{\_\}$. Its extension set is defined as*

$$\{|\, c.v_1.\ldots.v_k \,|\} := \{c.v'_1.\ldots.v'_k \mid \left\{ \begin{array}{ll} v'_j = v_j, & v_j \neq \_ \\ v'_j \in T_j, & \textit{otherwise} \end{array} \right\}.$$

Note that by definition, the set of partial events includes the set of (complete) events. We give an example for Definition 2.2.6:

Example 2.2.7. *Let c be a channel of type $\mathbb{N} \times \mathbb{B}$. Then,*

$\{|\, c.\_.\, true \,|\} = \{c.v_1.\, true \mid v_1 \in \mathbb{N}\}$ *and*
$\{|\, c.3.\_ \,|\} = \{c.3.\, true, c.3.\, false\}$.

Semantics of CSP

In order to analyse specifications and, in particular, CSP processes, we need to consider the formalism's *semantics*. The standard semantic model of CSP is the *failures-divergences* model. In addition, the less discriminating *stable failures* model and the least restrictive *traces* model can be chosen.

Traces of a CSP process describe its observable behaviour by means of sequences of events. The prefix closed set of all finite traces of a CSP process P is denoted by $traces(P) \subseteq \mathbb{P}(Events^*)$. Elements are described as sequences $\langle e_1, e_2, \ldots, e_n \rangle$ with $e_i \in Events$. Internal events (τ-events) do not appear in the traces of a process. For instance, $\langle pay.2, switch, select.CRISPS, order, deliver.COOKIE, term.2 \rangle$ describes a valid trace of the candy machine's CSP part.[1]

A *failure* of a CSP process P is expressed as a tuple $(tr, A) \in Events^* \times \mathbb{P}(Events)$ where tr denotes a trace and A a set of events which P is unable to accept after tr has been performed. For instance, $(\langle switch \rangle, \{pay\})$ is a failure of the CSP part of the candy machine.

Divergence within a CSP process P describes the ability of P to perform an infinite sequence of internal events. The set of divergences of a process P contains the set of traces after which P can diverge.

Our decomposition approach focusses on the verification of *safety* properties. As explained in [Weh00] and [OW05], this allows us to move to the semantic domain of the CSP traces model.

[1] Note that this is *not* a valid trace if we additionally consider the Object-Z part.

Next, we introduce some notations adopted from [Sch99] and [Ros98] which we will use in this thesis, and we start with the projection of a trace on a set of events:

Definition 2.2.8. *(Trace Projection)*
The projection of $tr \in traces(P)$ with respect to a set of events A is denoted by $tr \upharpoonright A$ and defined as:

$$\begin{aligned} \langle\rangle \upharpoonright A &= \langle\rangle, \\ (\langle a\rangle \frown tr) \upharpoonright A &= \begin{cases} tr \upharpoonright A, & a \notin A, \\ \langle a\rangle \frown (tr \upharpoonright A), & a \in A \end{cases} \end{aligned}$$

As an example:

$$\langle pay.2, switch, select.CRISPS, order, deliver.COOKIE, term.2\rangle \upharpoonright \{pay, term\} = \langle pay.2, term.2\rangle.$$

The set of *initial* events, a process is able to perform, is defined as follows:

Definition 2.2.9. *(Initials)*
Let P be a CSP process. Then,

$$initials(P) := \{a \mid \langle a\rangle \in traces(P)\}$$

For instance, $initials(\texttt{main}) = \{pay, switch, payout\}$.

In order to describe that a certain CSP process satisfies a given property, also described as a process, we need to be able to effectively *compare* processes. The general concept behind this is to show *refinement* of one process by another. If a specification Q refines another specification P, then Q is more restrictive and preserves the behaviour of P. In our semantic domain of traces, preservation means that Q offers fewer traces than P thus not allowing more behaviour. This gives rise to the following definition:

Definition 2.2.10. *(Trace Refinement)*
Let P, Q be CSP processes. Q is a trace refinement *of P, if $traces(Q) \subseteq traces(P)$. We write $P \sqsubseteq_T Q$. P is* trace equivalent *to Q, $P =_T Q$, if, and only if, $P \sqsubseteq_T Q$ and $Q \sqsubseteq_T P$.*

The traces of a CSP process can be obtained by defining its *transition graph*. A labelled transition system for a process can be deduced from the operational semantics of CSP. LTSs are the standard way for describing CSP processes in terms of transition graphs. For more details on the operational semantics of CSP, we refer to [Ros98].

Definition 2.2.11. *(Labelled Transition System for CSP)*
The LTS semantics of a CSP process `main` *over a set of events A is defined as the labelled transition system $M^{CSP} = (S, S_0, \rightarrow_{CSP})$ with*

- *$S = L^{CSP}$ the set of all CSP terms,*
- *$S_0 = \{\texttt{main}\}$,*
- *$\rightarrow_{CSP}$ according to the operational semantics of CSP.*

The labelled transition system definitions for Object-Z and CSP will be used to define the operational semantics of CSP-OZ.

Case Study Revisited

One particular property of the candy machine specification can informally be described as follows:

" The amount of money, paid by the customer, must be equal to the sum of the values of all delivered candies plus the potential spare money. "

For specifying properties of a specification, we will use CSP as the modelling language. This is reasonable since the semantics of CSP-OZ specifications can jointly be given in terms of CSP alone as we will see in Section 2.2.4.

Figure 2.6 defines a CSP process *Prop*, exactly describing the previously introduced property. Here, all three comprised processes have a parameter of type $\mathbb{N}$, counting the current amount of inserted money and credits, respectively. This yields three sets of *families* of process equations. $\mathit{Paying}(i)$ monitors the sum of inserted money whereas $\mathit{Collecting}(i)$ decreases the sum by the specific costs of the delivered candies. In order to identify the respective candy delivered by the machine, we explicitly denote the parameter for the event *deliver*. Finally, $\mathit{Terminate}(i)$ calls the event *term* with the remaining money of value i.

$$
\begin{array}{rcl}
\mathit{Prop} & = & \mathit{Paying}(0) \\
\mathit{Paying}(i) & = & \Box_{j \in \mathit{Coins}} (\mathit{pay}.j \rightarrow \mathit{Paying}(i+j)) \,\Box\, \mathit{Collecting}(i) \\
\mathit{Collecting}(i) & = & \mathit{deliver}.\mathit{CHOC} \rightarrow \mathit{Collecting}(i-1) \,\Box \\
 & & \mathit{deliver}.\mathit{COOKIE} \rightarrow \mathit{Collecting}(i-2) \,\Box \\
 & & \mathit{deliver}.\mathit{CRISPS} \rightarrow \mathit{Collecting}(i-3) \,\Box \\
 & & \mathit{Terminate}(i) \\
\mathit{Terminate}(i) & = & \mathit{term}.i \rightarrow \texttt{Skip}
\end{array}
$$

Figure 2.6: Correctness requirement for the candy machine specification

In order to show that *Prop* is valid for the specification of a candy machine, we need to prove

$$\mathit{Prop} \sqsubseteq_T \mathit{CandyMachine} \setminus \{\mid \mathit{payout}, \mathit{abort}, \mathit{startOrder}, \mathit{select}, \mathit{order} \mid\}.$$

As we are only interested in the behaviour reflected by *Prop*, all events not occurring in *Prop* are hidden.

2.2.4 Semantics of CSP-OZ

For the definition of the semantics of CSP-OZ, Fischer [Fis00] uses an extension of CSP, which he calls CSP_Z, and ultimately defines a CSP_Z process capturing the semantics

of a CSP-OZ class. Figure 2.7 shows a simplified version of his definition: a function *Semantics* inputs a CSP-OZ class and translates it into a CSP process. Here, the operator & represents *guarding* of an event defined as

$$b \;\&\; P :\Leftrightarrow (\textbf{if } b \textbf{ then } P \textbf{ else } \texttt{Stop}).$$

$$
\begin{array}{l}
\text{Semantics(S)} = \\
\quad \textbf{let} \\
\qquad \text{Z\_PART}(s) \quad = \\
\qquad\quad \Box_{op \in Op,\ in \in In(op),\ sim \in Simple(op)}\ \texttt{enable\_}op(s, in, sim)\ \& \\
\qquad\qquad \sqcap_{out \in Out(op),\ s' \in \texttt{State}' \bullet \texttt{effect\_}op(s,in,sim,out,s')} \\
\qquad\qquad\quad (op.in.sim.out \rightarrow \text{Z\_PART}(s')) \\
\\
\qquad \text{Z\_MAIN} \quad = \quad \sqcap_{s \in \texttt{State} \bullet \texttt{Init}(s)}\ \text{Z\_PART}(s) \\
\quad \textbf{within} \\
\qquad \text{Z\_MAIN} \ \|_{Events}\ \texttt{main}
\end{array}
$$

Figure 2.7: Translation of a CSP-OZ specification into a CSP process

The basic underlying idea for this definition is to define a CSP process Z_MAIN modelling the Object-Z part of the specification and putting it in parallel with the specification's original CSP part `main`. Both processes need to synchronise on *Events*. Z_MAIN non-deterministically chooses a valid initial state s and subsequently calls Z_PART(s). Z_PART(s) recursively executes operations of the Object-Z part in an arbitrary order as long as the operation's `enable`-schema is satisfied. Input parameters are deterministically chosen (using $\Box$) whereas output parameters and post states are determined non-deterministically (using $\sqcap$). This is motivated by the idea that output parameters and post states are internally chosen by a specification.

We aim at using the model checker `FDR2` [For05] for verifying CSP-OZ specifications against certain requirements. For this, we need to translate a CSP-OZ specification to the input language of `FDR2`, CSP_M, without changing its semantics. A tool-supported translation of CSP-Z to CSP_M has been accomplished in [MS01], [FMS01]. Bolton and Davies compare data refinement in Object-Z with failures-refinement in CSP based on the Object-Z semantics as given in [Smi95] and use a translation of Object-Z to CSP_M. In the context of refinement, Schneider [Sch05] introduced a more general translation from abstract data types (ADTs) [LZ74] to CSP, which can be applied for (part of) Object-Z as well.

In our context of CSP-OZ, Fischer [Fis97] derives a failures-divergences semantics for CSP-OZ based on the definition from Figure 2.7. This allows us to generally use a CSP model checker based on this transformation of CSP-OZ specifications. A transformation function using the above translation and resulting in a process defined in CSP_M is

introduced in [FW99]. In Chapter 7, we will apply this transformation to use FDR2 for checking trace refinements. We give more details on FDR2 and the tools we are using there.

Our aim is to give an *operational* semantics for CSP-OZ by using the definition from Fischer. In this thesis, we are interested in the *paths* a specification might execute. In this particular case, we do not need to deal separately with external choice and internal choice: based on the operational semantics of CSP, the trace semantics does not distinguish between external and internal choice. More precisely, for two CSP processes P_1 and P_2:

$$P_1 \,\Box\, P_2 =_T P_1 \sqcap P_2.$$

Next, we define the operational semantics of CSP-OZ by putting the labelled transition systems for the specification's CSP part and Object-Z part in parallel. The parallel composition of two labelled transition systems is defined as follows:

Definition 2.2.12. *(Parallel composition of labelled transition systems)*
Let $M^1 = (S^1, {S^1}_0, \rightarrow_1)$ *and* $M^2 = (S^2, {S^2}_0, \rightarrow_2)$ *be two labelled transition systems over the same set of events E. The* parallel composition *of* M^1 *and* M^2 *is defined as* $M^1 \parallel_E M^2 = (S, S_0, \rightarrow)$ *with*

- $S = S^1 \times S^2$, $S_0 = S^1_0 \times S^2_0$,
- $(s_1, s_2) \xrightarrow{e} (s'_1, s'_2)$ *if one of the three conditions*
  - *a)* $s_1 \xrightarrow{e}_1 s'_1 \wedge s_2 \xrightarrow{e}_2 s'_2$,
  - *b)* $s_1 \xrightarrow{\tau}_1 s'_1 \wedge s'_2 = s_2$,
  - *c)* $s'_1 = s_1 \wedge s_2 \xrightarrow{\tau}_2 s'_2$.

  holds.

The operational semantics of CSP-OZ is then defined as the parallel composition of M^{CSP} and M^{OZ}, synchronising on *Events*. Note that we assume the alphabets of operations of the CSP part and the Object-Z part to be equal. This is not a restriction, as any operation solely represented in the CSP part of a class can be added to its Object-Z part by using an empty predicate part not modifying the state space of the class. Conversely, operations exclusively appearing in the Object-Z part can be integrated into the CSP process by globally offering them based on an additional interleaving. Besides, based on the operational semantics of CSP, M^{CSP} can indeed perform τ-events whereas M^{OZ} cannot.

Table 2.1 gives an overview on the two semantics of CSP-OZ which we introduced in this section. When showing correctness of our approach, we will refer to the LTS semantics of CSP-OZ, incorporating state valuations and events in their paths. The more discriminating CSP_Z semantics maps a CSP-OZ class specification on a CSP process, preserving the original behaviour within the failures-divergences model.

Even though the translation of CSP-OZ to CSP_M uses the CSP_Z semantics and is thus semantics-preserving for any of the three models of CSP, our approach solely focusses on the traces model.

	LTS SEMANTICS	**CSP$_Z$ SEMANTICS**
Semantic model	traces model of CSP	failures-divergences model of CSP
Alphabet for paths	(`State` $\times$ *Events*)	*Events*
Application in thesis	correctness proof	model checking (trace refinement)

Table 2.1: Comparison between the different semantics for CSP-OZ

2.3 Dependence Analysis

The main aspects of this thesis are the construction and evaluation of decompositions for software models, specified in CSP-OZ. We require *correctness* of our approach, meaning that the decomposition must preserve the observable behaviour of the specification. This is achieved by a correctness proof, requiring a representation of the model on which the decomposition and the general proof can be carried out. This representation must reflect the structure of the specification as well as the interdependences between its elements.

In his PhD thesis [Brü08], Brückner introduced a dependence analysis for CSP-OZ based on the definition of a *(program) dependence graph*. In the context of program slicing [Wei81], he uses it to show correctness of his approach. Since a dependence graph precisely reflects all the specification's interdependences, we can take advantage of this construction and use his graph in a slightly modified version.

This section introduces the dependence analysis for CSP-OZ specifications mainly according to [Brü08]. We start with a small motivation, stepwise present the dependence graph and illustrate its definition by means of our case study. Instead of repeating all the details of the dependence analysis, we concentrate on the main aspects and an illustration of the concept. Along with that, we describe our context-specific modifications and introduce some necessary properties of the dependence graph.

2.3.1 Dependence Analysis for CSP-OZ: Motivation

The introduction already gave an overview on the overall goals of this thesis. In particular, Figure 1.1 illustrated the approach for decomposing a given specification S into two components S_1 and S_2, yielding a system $S_1 \parallel S_2$.

Decomposing a CSP-OZ specification S means that S is split-up into two smaller CSP-OZ specifications S_1 and S_2. For that purpose, the specification's elements, such as operation schemas and state variables, are distributed over S_1 and S_2. In order to define *correct* decompositions, we cannot simply assign these constituents to S_1 and S_2 *at random*: the specification's elements might *depend* on each other. The distribution of dependent elements over both components is not beneficial but generally possible. A definition of S_1 and S_2 needs to conform to the structure of the original model such that the

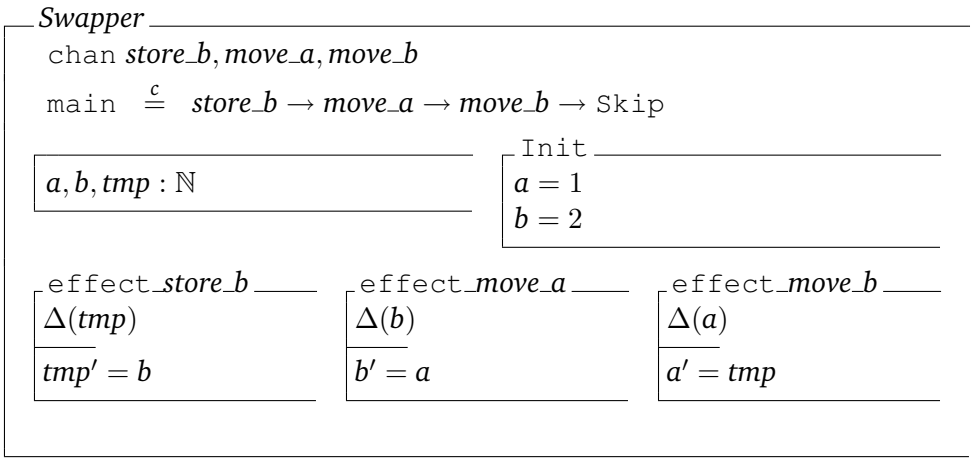


Figure 2.8: Simple CSP-OZ class specification for swapping two numbers

(observational) equivalence of S and $S_1 \parallel S_2$ can be deduced.

We illustrate the need for a precise specification's analysis with a small example. Consider the simple CSP-OZ specification *Swapper* as given in Figure 2.8. The specification swaps two natural numbers a and b with respective values 1 and 2 by using a temporary variable *tmp*.

A decomposition could, for instance, yield two specifications $Swapper_1$ and $Swapper_2$ such that *store_b* and *move_b* are distributed over different components. This definition bears some problems: first, the parallel composition of the resulting CSP parts needs to preserve the original ordering of events $\langle store\_b, move\_a, move\_b \rangle$ according to *Swapper*.`main`. In the parallel composition $Swapper_1 \parallel Swapper_2$, the operation *move_b* must not be performed prior to any other event. The dependence graph must therefore comprise edges reflecting the *control flow* of a specification's CSP part.

Second, consider the Object-Z part of the resulting specification part $Swapper_2$: the variable *tmp* is modified within *store_b*. We need to ensure that *move_b* refers to the correct value of *tmp*. The modified value somehow needs to be restored within $Swapper_2$. This interconnection needs to be reflected in the dependence graph as well. Here, we use edges representing the specification's *data dependences*.

In general, we need to preserve the dependence structure of both, a specification's CSP part and Object-Z part. Our dependence analysis for CSP-OZ specifications addresses this issue by using two graphs:

a) a *control flow graph (CFG)*, which represents the workflow of the specification's CSP part and

b) a *data dependence graph (DDG)*, representing the interdependences between state variables and parameters of the specification's Object-Z part.

The overall dependence structure is subsequently defined in the specification's *(program) dependence graph (DG)* combining the CFG and DDG. Our definition of the DG

mainly corresponds to the one by Brückner [Brü08]. However, in this thesis and contrary to Brueckner's definitions, the DG is defined with respect to operation nodes. We do not separately consider an operation's `enable`- and `effect`-schema and its contained predicates.

A decomposition defines a split-up of the dependence graph which then leads to the decomposition of the underlying specification. Preservation of the observable behaviour is defined in terms of correctness criteria on this fragmentation in Chapter 4.

2.3.2 Definition of the Control Flow Graph

In order to analyse a program in respect of its execution paths, control flow analysis [All70] is a standard practice. A *control flow graph* is a graph theoretical representation of a program.

The definition of [Brü08] yields a control flow graph representing a CSP process. Nodes of this graph mainly correspond to CSP events and CSP operators. We start with the general definition of the control flow graph. Ultimately, we are interested in a dependence graph for a CSP-OZ specification S. To this end, the following definitions refer to the CSP part `main` of a CSP-OZ specification and to the set of operation schemas Op of S.

Definition 2.3.1. *(Control Flow Graph (CFG) of S)*
The control flow graph (CFG) $\mathtt{CFG}_S = (N, \longrightarrow)$ *of a CSP-OZ specification S is defined over a set of nodes* $N = \mathtt{cf}(N) \cup \mathtt{op}(N)$ *and a set of edges* $\longrightarrow \subseteq N \times N$.

Nodes of the CFG either correspond to a CSP operator or to an operation schema of the underlying specification. Table 2.2 denotes all nodes along with the corresponding CSP operators, if existent.

We use a unique node start, representing the start of `main`. The set N comprises the set of nodes $\mathtt{op}(N)$ which is defined as

$$\mathtt{op}(N) = \{\mathsf{op}^i \mid op \in Op\} \cup \{\mathsf{init}\}.$$

Here, a special node init represents the initial state schema of a class, comprising all initial predicates. For the definition of the CFG, the init-node is conjoined with the start-node of the class. As an operation schema op may occur more than once in `main` and thus in its CFG, we denote the i-th occurrence of the respective operation node by op^i.

$\mathtt{cf}(N)$ is the set of CSP operator nodes plus a set of additional nodes representing entry and leaving of a process. The whole set complies with the elements of the CSP grammar as given in Figure 2.5. Some of these operators, namely the ones corresponding to external choice, internal choice, both parallel operators (which are not separately dealt with in the CFG) and interleaving, introduce *branching* into the CFG. Here, we introduce split nodes and corresponding join nodes, which are denoted by cfop and uncfop for $\mathsf{cfop} \in \{\mathsf{extch}, \mathsf{intch}, \mathsf{par}, \mathsf{interleave}\}$, respectively. According to operation nodes, the same notation for the i-th occurrence of a CSP operator node applies.

Note that we do not separately consider parallel composition of *classes* since, on graph level, parallel composition of classes and processes is equally dealt with [Brü08]. Thus, we equally treat specifications consisting of one and several classes.

Node	CSP operator	Name
start	-	(Start of `main`)
op^i	-	(Operation Node for $op \in Op$)
skip ^{i}	`Skip`	(Termination)
stop ^{i}	`Stop`	(Deadlock)
extch ^{i}	$\Box$	(Split External Choice)
unextch ^{i}	-	(Join External Choice)
intch^i	$\sqcap$	(Split Internal Choice)
unintch ^{i}	-	(Join Internal Choice)
seq ^{i}	$\overset{\circ}{9}$	(Sequential Composition)
par ^{i}	$\parallel$	(Split Parallel)
unpar ^{i}	-	(Join Parallel)
interleave ^{i}	$\|\|\|$	(Split Interleave)
uninterleave ^{i}	-	(Join Interleave)
start.X	-	(Process Entry)
term.X	-	(Process Termination)
call.X	-	(Process Call)
ret.X	-	(Process Return)

Table 2.2: Table of nodes of the control flow graph

The remaining four nodes are used for structuring of CSP process definitions: start of a process X, termination of X, call of X and returning from X. As an example, executing *switch* in the candy machine and subsequently calling the process *Select* corresponds to a CFG path $\langle \ldots, switch, \mathsf{call}.Select, \mathsf{start}.Select, \ldots \rangle$.

In general, a CFG node $\mathsf{n} \in N$ must always have zero, one or two successor nodes. We denote a single successor node by succ(n), in case of two successor nodes we separately denote each one with succ_one(n) and succ_two(n), respectively.

Paths of the CFG precisely reflect the control flow of `main`. For the correctness proof, we make one important observation: according to the definition of the CFG, the sole possibility of cycles within the CFG are process calls within the CSP part. This is reflected in [Brü08] where the definition of G_{CFG} introduces cycles into the CFG solely for the case of call-nodes.

Figure 2.9 shows a slightly simplified version of the CFG for the candy machine specification. We omit unextch nodes, term nodes and ret nodes to avoid a blow up in the illustration. Operation nodes are highlighted in grey.

The following notations for paths of the CFG are mainly corresponding to [Brü08]:

Definition 2.3.2. *(Paths of the Control Flow Graph)*
Let $\mathtt{CFG}_\mathsf{S} = (N, \longrightarrow)$ *be the CFG of* S*, and let* $\mathsf{n}, \mathsf{n'} \in N$*. We use the following notations for* paths *of the CFG, that is, sequences of nodes, visited, when walking along the edges of the graph:*

- $\mathtt{path}_{\mathtt{CFG}}$ *denotes the set of all paths of the CFG whereas* $\mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'})$ *denotes the set of paths starting in* n *and terminating in* n'.

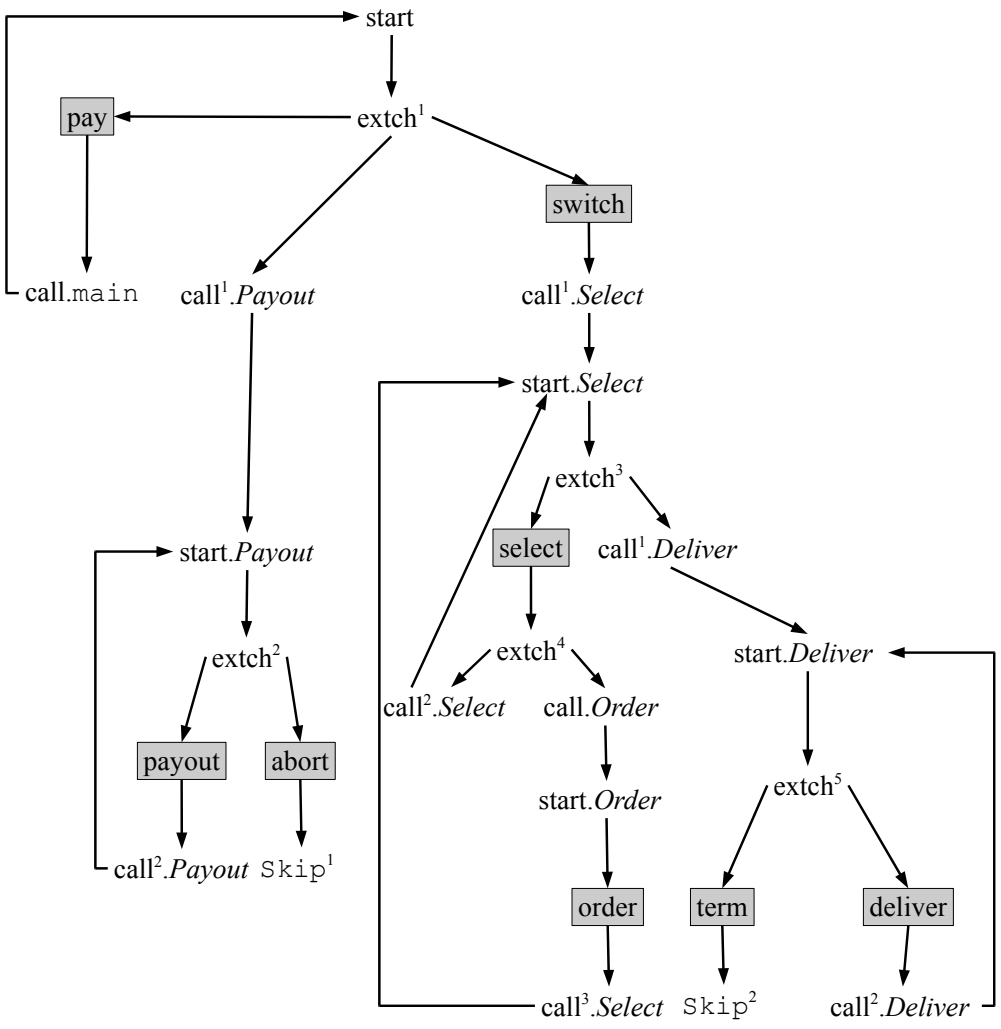


Figure 2.9: Control flow graph (CFG) for the candy machine specification

- $n \xrightarrow{\pi} n'$ *denotes that* $\pi \in \mathtt{path}_{\mathtt{CFG}}(n, n')$ *whereas* $n \xrightarrow{*} n'$ *denotes that there* exists *some path from* n *to* n'*, that is,* $\mathtt{path}_{\mathtt{CFG}}(n, n') \neq \varnothing$.
- *For* $n, n' \in \mathtt{op}(N)$*, we write*

$$n \xrightarrow{\bullet} n' \text{ if, and only if, } (\exists\, \pi \in \mathtt{path}_{\mathtt{CFG}}(n, n') \bullet \pi \cap \mathtt{op}(N) = \{n, n'\}).$$

We will sometimes need to refer to paths connecting two operation nodes n, n' without additional operation nodes in between. For this, we use the last definition. For $\pi \in \mathtt{path}_{\mathtt{CFG}}(n, n')$, we let $x \in \pi$ denote that $x \in N$ is an arbitrary node on the path π, including n and n' themselves.

We give a small illustrating example for the previous definition:

Example 2.3.3. *For the CFG of the candy machine from Figure 2.9, we get:*

- $\langle \mathsf{start}, \mathsf{extch}^1, \mathsf{switch}, \mathsf{call}^1.\mathit{Select}, \mathsf{start}.\mathit{Select}, \mathsf{extch}^3, \mathsf{select}\rangle \in \mathtt{path}_{\mathtt{CFG}}$,
- $\langle \mathsf{switch}, \mathsf{call}^1.\mathit{Select}, \mathsf{start}.\mathit{Select}, \mathsf{extch}^3, \mathsf{select}\rangle \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{switch}, \mathsf{select})$,
- $\mathsf{start}.\mathit{Deliver} \longrightarrow \mathsf{extch}^5$,
- $\mathsf{switch} \xrightarrow{*} \mathsf{order}$ *and*
- $\mathsf{select} \xrightarrow{\bullet} \mathsf{order}$.

Next, we define a mapping between the set of operation nodes of the CFG, $\mathtt{op}(N)$, and the set of operations *Op* of a specification:

Definition 2.3.4. *(Labelling of CFG nodes)*
Let $\mathtt{CFG}_S = (N, \longrightarrow)$ *be the CFG of* S*, and let Op be the set of all operation schemas of S. The* labelling function $l : \mathtt{op}(N) \to Op$ *maps an operation node of the CFG on its corresponding schema name:* $l(\mathsf{op}^i) := op$*. For* $O \subseteq Op$*, we define*

$$l^{-1}[O] := \{\mathsf{n} \in \mathtt{op}(N) \mid l(\mathsf{n}) \in O\}.$$

As multiple occurrences of an Object-Z operation within the CSP part of a specification's class are possible, the cardinality of $\mathtt{op}(N)$ is greater or equal than the cardinality of *Op*: for all $op \in Op$, there exists at least – but in many cases more than – one occurrence op^i within the DG. Thus, the mapping l is surjective but in general not injective. If $l^{-1}[\{op\}]$ only contains one element, we denote it by op, leaving out the index.

For a more precise definition and description of the CFG, we refer to [Brü08].

2.3.3 Definition of the Data Dependence Graph

The control flow of a program can be represented in a graph theoretical way, and the same applies to its data flow. Data flow analysis and data dependence graphs [Den74] aim at an evaluation and description of dependent program statements, incorporating data values. A data dependence is, for instance, given if one statement modifies a certain program variable, while another statement refers to it, and the variable is not overwritten in between.

The data dependence graph, which we consider, is solely defined over the set of nodes $\mathtt{op}(N)$, that is, the set of operation schemas of a specification plus its initial state schema. It supplements the CFG in the sense that its edges are related to paths of the CFG and that it is mainly derived from the Object-Z part of a specification.

As already mentioned, `enable`- and `effect`-schemas are comprised into one operation node. Dealing with operation nodes instead of its constituents is reasonable, since, as we will see in Chapter 4, our decomposition approach does not further decompose an operation but rather keeps operations as atoms.

Besides, we refer to a normalisation of the Object-Z part of S according to [Brü08]: as a state invariant needs to hold before and after execution of each method, it can safely be copied to the `effect`-schema of each method and eliminated from the state schema, without changing the behaviour of the specification.

The definition of the data dependence graph is as follows:

Definition 2.3.5. *(Data Dependence Graph (DDG) of S)*
The data dependence graph (DDG) $\mathtt{DDG}_S = (\mathtt{op}(N), \dashrightarrow)$ *of a specification S is defined over a set of nodes* $\mathtt{op}(N)$ *and a set of edges* $\dashrightarrow \subseteq \mathtt{op}(N) \times \mathtt{op}(N)$.

Edges of the DDG incorporate several dependences with all of them being introduced in [Brü08]. Table 2.3 denotes all comprised edges. Note that we do not consider control dependences, as we will explain in the next section.

Edge	Name
$\overset{\mathtt{dd}}{\dashrightarrow}$	(Direct Data Dependence)
$\overset{\mathtt{idd}}{\dashrightarrow}$	(Initial Data Dependence)
$\overset{\mathtt{ifdd}}{\dashrightarrow}$	(Interference Data Dependence)
$\overset{\mathtt{sd}}{\leftarrow\!\dashrightarrow}$	(Synchronisation Dependence)
$\overset{\mathtt{sdd}}{\dashrightarrow}$	(Synchronisation Data Dependence)

Table 2.3: Table of edges of the data dependence graph

The simplest example is a *(direct) data dependence*: assume a certain state variable $v \in V$ being modified in some operation schema op_1 and referenced in some other operation schema op_2, that is, $v \in (mod(op_1) \cap ref(op_2))$. For all operation nodes $\mathsf{n} \in l^{-1}(op_1)$, $\mathsf{n'} \in l^{-1}(op_2)$, such that there exists a CFG path from the first to the latter node and v is not further modified on this path, the DDG contains a data dependence edge $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$.

Initial data dependences are a special case of direct data dependences. Since the initial state schema poses restrictions on the set of state variables, an initial data dependence connects the representation of the initial state schema with an operation if some variable v is restricted in `Init` and referenced in op, without being overwritten in between. As initial data dependences will frequently be used in the following chapters, we introduce a separate notation: $\mathsf{init} \overset{\mathtt{idd}}{\dashrightarrow} \mathsf{n'}$, if, and only if, $\mathsf{init} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$ for $\mathsf{n'} \in l^{-1}(op)$.

An *interference data dependence* exists from one node to another if both nodes are located in different branches of an *interleaving* or *parallel composition* and, again, the source node modifies a variable that the target node references. Note that, in general, there is no CFG path connecting both nodes.

Synchronisation dependences model the fact that synchronised events within a parallel composition have a mutual dependence on each other. These edges can more likely be seen as a representation of the control flow. However, we integrate them in the DDG, since we want to keep the original definition of the CFG. Note that synchronisation dependences are always symmetric.

Finally, *synchronisation data dependences* complement synchronisation dependences. They connect two operation nodes if they are connected by a synchronisation dependence, and one of the corresponding operations declares an output variable which the other corresponding operation uses as an input.

Since we will need to refer to the precise conditions for some of these edges later on, we give their definitions next.

Definition 2.3.6. *((Direct-, Interference-) Data Dependence, Synchronisation Dependence)*

1.) A direct data dependence exists from $\mathsf{n} \in \mathtt{op}(N)$ *to* $\mathsf{n'} \in \mathtt{op}(N)$, $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$*, if, and only if,*

$$
\begin{array}{ll}
\exists\, op_1, op_2 \in Op \bullet \mathsf{n} \in l^{-1}(op_1), \mathsf{n'} \in l^{-1}(op_2) \wedge & \textit{(nodes corresp. to two operations)} \\
\exists\, v \in (mod(op_1) \cap ref(op_2)) \wedge & \textit{(v modified in } op_1\textit{, referenced in } op_2\textit{)} \\
\exists\, \pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'}) \bullet & \textit{(nodes are connected by CFG path)} \\
\forall\, \mathsf{m} \in \pi \bullet v \in mod(l(\mathsf{m})) \Rightarrow (\mathsf{m} = \mathsf{n}) \vee (\mathsf{m} = \mathsf{n'}) & \textit{(no further modification of v)}
\end{array}
$$

2.) An interference data dependence exists from $\mathsf{n} \in \mathtt{op}(N)$ *to* $\mathsf{n'} \in \mathtt{op}(N)$, $\mathsf{n} \overset{\mathtt{ifdd}}{\dashrightarrow} \mathsf{n'}$*, if, and only if,*

$$
\begin{array}{ll}
\exists\, op_1, op_2 \in Op \bullet \mathsf{n} \in l^{-1}(op_1), \mathsf{n'} \in l^{-1}(op_2) \wedge & \textit{(nodes corresp. to two operations)} \\
\exists\, v \in (mod(op_1) \cap ref(op_2)) \wedge & \textit{(v modified in } op_1\textit{, referenced in } op_2\textit{)} \\
\exists\, \mathsf{m} = (\mathsf{interleave} \vee \mathsf{par}_S \bullet op \in S) \wedge & \textit{(interleaving or parallel composition)} \\
\exists\, \pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{m}, \mathsf{n}) \wedge & \textit{(first node in one branch)} \\
\exists\, \pi' \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{m}, \mathsf{n'}) \bullet & \textit{(second node in the other branch)} \\
\pi \cap \pi' = \{\mathsf{m}\} & \textit{(no join of branches within paths)}
\end{array}
$$

3.) A synchronisation dependence exists between $\mathsf{n}, \mathsf{n'} \in \mathtt{op}(N)$, $\mathsf{n} \overset{\mathtt{sd}}{\dashleftarrow\dashrightarrow} \mathsf{n'}$*, if, and only if,*

$$
\begin{array}{ll}
\exists\, op \in Op \bullet \mathsf{n}, \mathsf{n'} \in l^{-1}(op) \wedge & \textit{(two nodes corresponding to same operation)} \\
\exists\, \mathsf{m} = \mathsf{par}_S \bullet op \in S \wedge & \textit{(parallel composition with operation synchronised)} \\
\exists\, \pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{m}, \mathsf{n}) \wedge & \textit{(first node in one branch of parallel composition)} \\
\exists\, \pi' \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{m}, \mathsf{n'}) \bullet & \textit{(second node in the other branch of par. composition)} \\
\pi \cap \pi' = \{\mathsf{m}\} & \textit{(no join of branches within paths)}
\end{array}
$$

Sometimes, we explicitly need to refer to the state variable responsible for a data dependence. This leads to the following notation:

Definition 2.3.7. *((Direct-, Interference-) Data Dependence by Reason)*
Let $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$*, and let the state variable v satisfy the criteria from Definition 2.3.6, 1.). In this case, we write* $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow}_{(v)} \mathsf{n'}$ *and say that* $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$ *holds* by reason of *v. Correspondingly, we define* $\mathsf{n} \overset{\mathtt{ifdd}}{\dashrightarrow}_{(v)} \mathsf{n'}$.

Note that $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$ and $\mathsf{n} \overset{\mathtt{ifdd}}{\dashrightarrow} \mathsf{n'}$ can hold by reason of more than one variable. The definitions for all kinds of dependences can be found in [Brü08]. We immediately deduce a small lemma which we will frequently use in the following chapters:

Lemma 2.3.8. *(Direct data dependence requires CFG path)*
Let $\mathsf{n}, \mathsf{n'} \in \mathtt{op}(N)$ *such that* $\mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow} \mathsf{n'}$*. Then,* $\mathsf{n} \overset{*}{\longrightarrow} \mathsf{n'}$.

Proof. Immediately follows from Definition 2.3.6, 1.). □

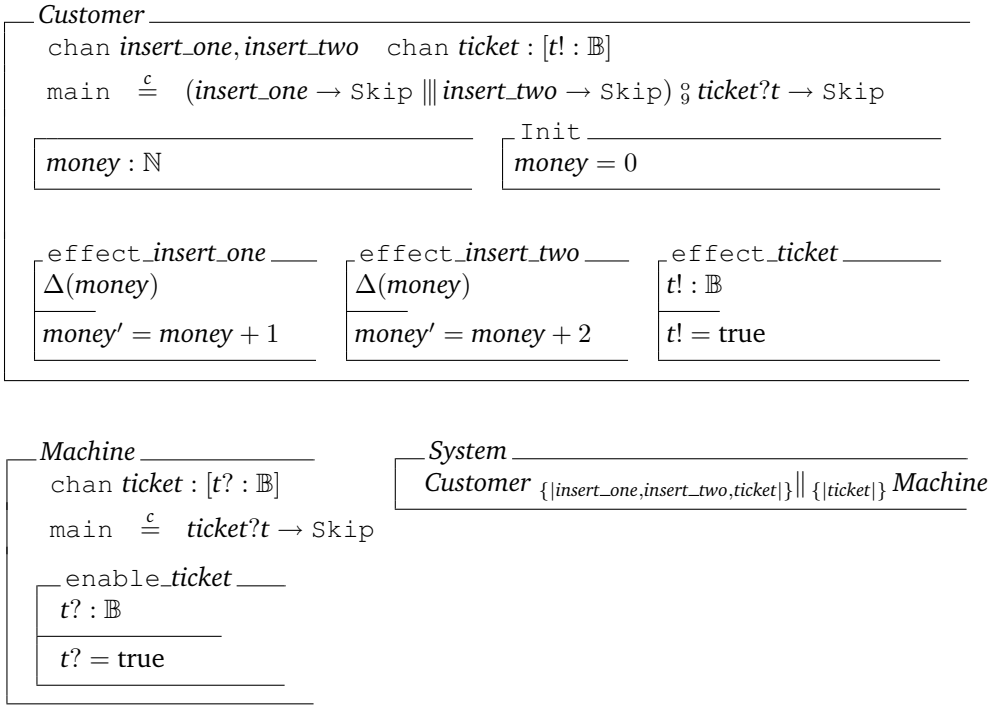


Figure 2.10: Simple CSP-OZ class specification for a ticket machine

Since our main case study does not incorporate all kinds of data dependences, we give a small example to illustrate them. Figure 2.10 shows a ticket machine specification consisting of two classes *Customer* and *Machine*. The overall system is defined as the parallel composition of both classes, synchronising on the set $\{|\ ticket\ |\}$. The customer can insert coins of value 1 and 2 in an arbitrary order and afterwards, the machine dispenses the ticket. The full DDG of this small specification is given in Figure 2.11. Edges are labelled according to Table 2.3.

The specification incorporates the following dependences:

Initial Data Dependence: Since the state variable *money* is restricted in the initial state schema of *Customer*, referenced within *insert_one*, *insert_two* and possibly not overwritten in between, there exist two initial data dependences (①, ②) from `Init` to the respective operations.

Synchronisation Dependence: The operation schema *ticket* is synchronised between both classes thus yielding a synchronisation dependence (③) between *Customer.ticket* and *Machine.ticket*.

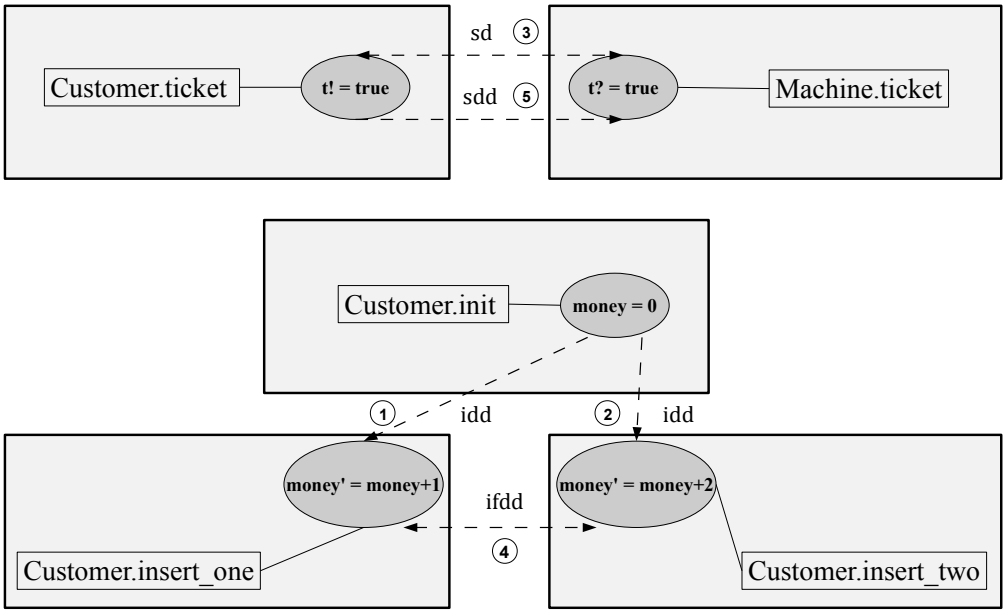


Figure 2.11: Data dependence graph (DDG) for the ticket machine specification

Interference Data Dependence: Based on $money \in (mod(insert\_one) \cap ref(insert\_two))$ and vice versa $money \in (mod(insert\_two) \cap ref(insert\_one))$, both nodes are connected via a (symmetric) interference data dependence (④).

Synchronisation Data Dependence: As *Customer.ticket* sends the value of the parameter *t* to *Machine.ticket*, a synchronisation data dependence (⑤), with *Customer.ticket* as the source node and *Machine.ticket* as the target node, connects both operation nodes.

The sole remaining data dependence, which we did not yet exemplify, is the *direct data dependence*. In the candy machine specification, one such edge is the link from *switch* to *select* due to $credits \in (mod(switch) \cap ref(select))$.

Figure 2.12 gives an extract of the DDG for the candy machine specification which solely comprises direct data dependences and initial data dependences. All edges of the DDG will be given in the next section as part of the specification's dependence graph.

2.3.4 Definition of the Dependence Graph

The idea of the definition of a *(program) dependence graph (PDG)*, as introduced in [FOW87], is the unification of all the dependences of a program into one determined graph which can then serve as the sole basis for the analysis of a program.

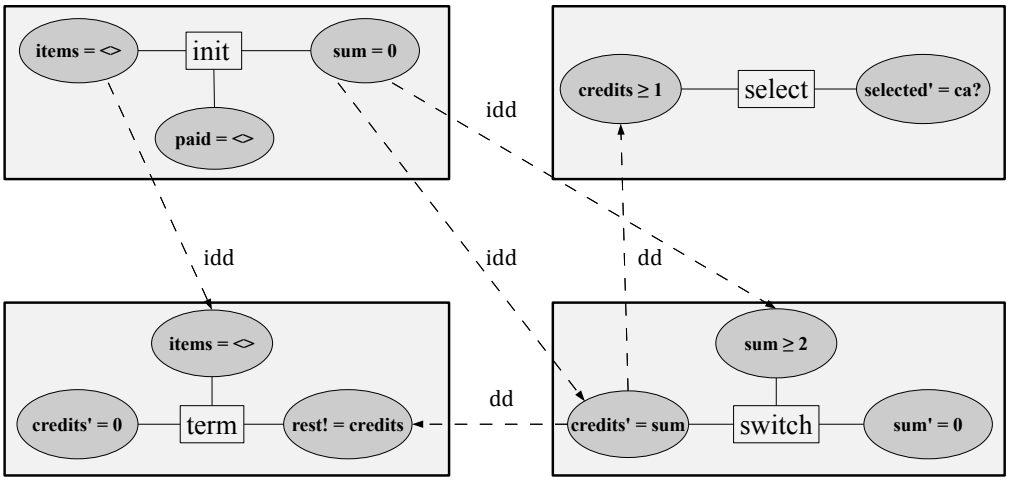


Figure 2.12: Extract of DDG for the candy machine specification

According to the structure of CSP-OZ specifications, the analysis of their dependence structure is two-folded: the construction of the overall dependence graph of the specification comprises the control flow graph for representing the control flow of a specification and the data dependence graph as a representation of its data flow. We will now consolidate both graphs into one. Again, we start with the general definition:

Definition 2.3.9. *(Dependence Graph (DG) of S)*
The dependence graph (DG) $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *of a CSP-OZ specification S is defined over a set of nodes N and a set of edges* $\longrightarrow_{\mathtt{DG}} \subseteq N \times N$*, where*

- $N = \mathtt{cf}(N) \cup \mathtt{op}(N)$ *and*
- $\longrightarrow_{\mathtt{DG}} = (\longrightarrow \cup \dashrightarrow)$*,*

according to Definition 2.3.1 and Definition 2.3.5 for the CFG and DDG, respectively.

The dependence graph is defined over the same set of nodes $N = \mathtt{cf}(N) \cup \mathtt{op}(N)$ as the CFG and comprises both, edges of the CFG and the DDG. Recall that edges of the DDG always connect operation nodes.

Our definition of the DG differs from the one defined in [Brü08] in several points:

Definition based on Operation Nodes: Our set of DG nodes comprises *operation* nodes instead of *predicate* nodes. Data dependences thus connect the respective operation nodes which contain the responsible predicates. This definition corresponds to Brückner's simplified graph representation, using super nodes. However, we additionally consolidate `enable`- and `effect` schemas of an operation into one node. The coarsening is motivated by the idea that in our decomposition, we will

keep operations atomic, that is, we will either assign all or none of the original predicates of an operation to the generated components.

Inclusion of the CFG: In our context, a decomposition completely needs to adhere to the CFG, since we must not destroy the overall dependence structure of a specification. Therefore, in contrast to Brückner, we integrate the full CFG into our dependence graph.

Neglect of Control Dependences: Based on the previous explanations, neither direct nor indirect control dependences as defined in [Brü08] are relevant in our context.

Neglect of Symmetric Data Dependences: Symmetric data dependences model sharing of modified variables between two predicates. These edges are only used for connecting two predicates within the same operation. Analogous to the previous explanations, we can safely omit them.

Paths of the DG are defined according to paths of the CFG, except that we use the notation $\texttt{path}_{\texttt{DG}}$. Finally, we present the dependence graph for our case study in Figure 2.13. We do not explicitly distinguish between the several types of edges of the DDG here. The `Init` schema of the specification, attached with outgoing initial data dependences, is linked to the start-node of the graph.

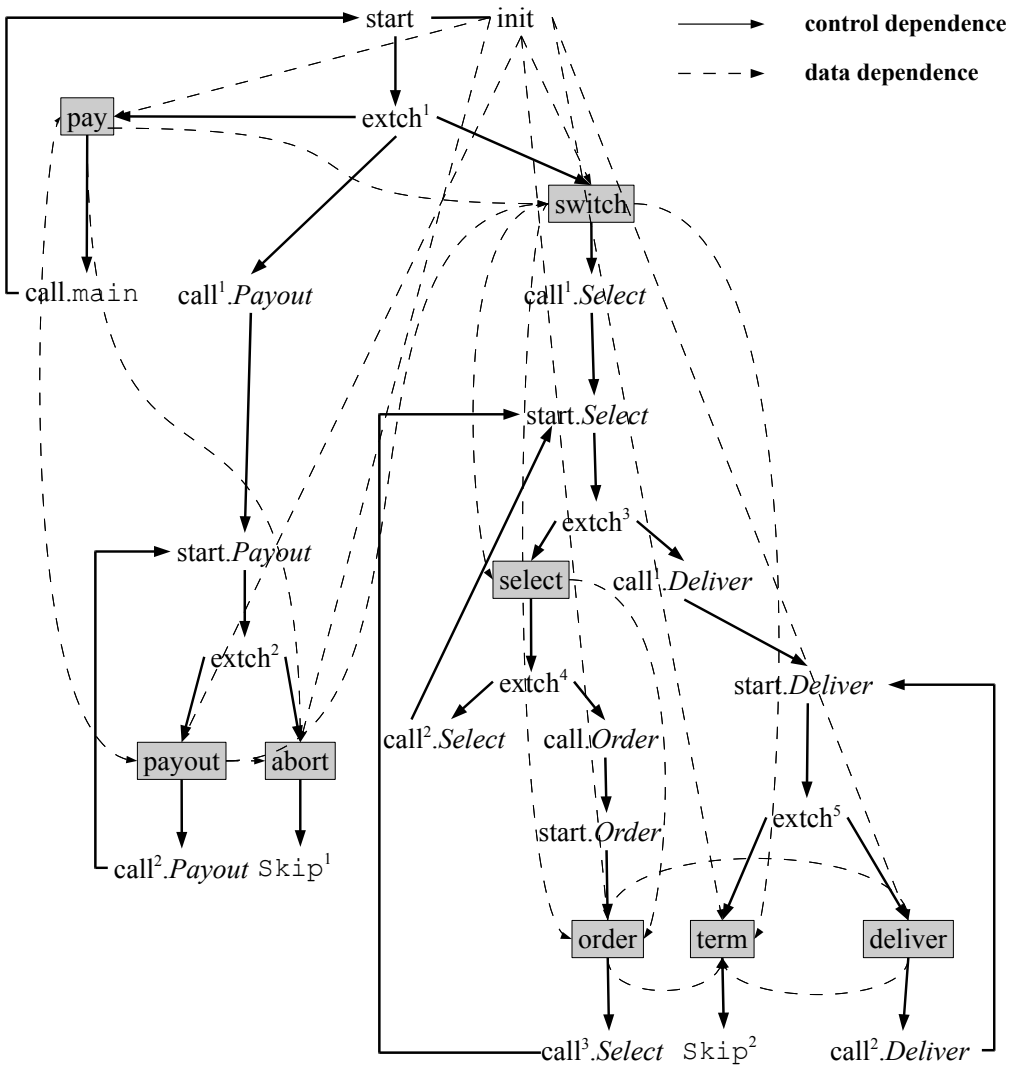


Figure 2.13: Dependence graph (DG) for the candy machine specification

3 Background: Compositional Reasoning

Contents

3.1 Approaches to the State Space Explosion 42

3.2 Compositional Reasoning 43

3.2.1 Assume Guarantee Proof Rules 43

3.2.2 Obstacles to the Application of Assume Guarantee Reasoning . 45

3.2.3 Learning for Compositional Verification 45

3.3 Assume-Guarantee Reasoning for CSP 47

3.3.1 Application Example: Elevator System 49

3.3.2 Soundness of Assume-Guarantee Proof Rules 50

3.4 Related Work 53

In the introduction, we discussed strategies to ensure the reliability of a software system. Our approach concentrates on the *verification* of a system model with respect to certain requirements. This is achieved by specifying the system in the integrated formalism CSP-OZ, as introduced in the previous chapter, and employ *model checking*.

Model checking [CGP99] is a technique to automatically verify a system model, represented as a finite state machine, against desired properties of the system, described in some logical formalism. It either shows the validity of the desired properties or produces counterexamples, giving some insight on why the model is invalid. The methodology is introduced in [EC80, CES86], and extensive research has been devoted to it over the last years.

Even though model checking algorithms generally have a linear or at worst polynomial complexity in the size of their underlying models [Sch02], they all need to compute the state space of the system, which exponentially grows in the number of its components. Therefore, the main focus of attention is to cope with this decisive task, know as the *state explosion problem*.

This chapter provides the necessary background on model checking techniques and particularly on compositional verification as our object of research. Section 3.1 elaborates on the most relevant techniques to tackle the state explosion problem. Subsequently, Section 3.2 gives an overview on compositional verification and introduces our employed proof method, assume-guarantee reasoning. Along with this, we describe a methodology on learning assumptions for an automation of assume-guarantee-based verification. Section 3.3 puts assume-guarantee reasoning into our semantic context. Finally, Section 3.4 discusses related work on compositional verification.

3.1 Approaches to the State Space Explosion

Verification of program correctness incorporates the analysis of any possible program execution and any reachable state. In order to achieve this, mathematical-based techniques aim to build a model, representing all possible program configurations. This structure is in general referred to as a program's *state space*. Model checking verifies the system model against certain requirements by analysing its state space.

Due to limited computing resources, automated verification of a software model can only construct models up to a certain extent. Thus, if the state space of a model becomes larger and larger, model checking becomes infeasible.

Model checking of specifications written in an integrated formal method are highly afflicted from the state explosion problem: as the data-oriented description of a system may cause an enormous state space due to large or even infinite data types, so does the behaviour-oriented description, owed to its concurrency. If two diverse formalisms are combined into one, automated formal verification suffers from both of these problems at the same time.

There are many strategies to tackle state explosion, with most of them having their specific advantages in certain domains. The most important techniques are described in the following.

Partial order reduction [KP88, God96] concentrates on the analysis of the *concurrency* of a system. More precisely, it aims at identifying independent and thus commutative transition paths in asynchronous systems. As a result, different orderings on these transitions can be conjoined. This technique clearly has its key benefits if applied to behaviour-oriented formalisms, incorporating asynchronous concurrency.

In order to apply model checking for infinite state systems, *abstraction techniques* need to be employed. In general, these techniques aim at either removing or simplifying parts of the system model.

One such technique is *data abstraction* [CGL94], which aims at handling large data domains. It is based on the idea of *abstract interpretations* [CC77]. Instead of evaluating a property with respect to all possible data values, an abstraction mapping identifies a set of concrete values for one abstract value. If the mapping satisfies certain correctness criteria, properties of the abstract system also hold for the concrete system. Data abstraction techniques for CSP-OZ were introduced in [Weh00].

However, too coarse abstractions can lead to wrong verification results. *Counterexample guided abstraction-refinement* [CGJ$^+$03] iteratively refines an initially minimal abstraction and is guided by the model checker's counterexamples. In case of a spurious counterexample, based on an over-approximation of the system, the model is refined, and the verification process is repeated. In the context of CSP and Z, this technique has been applied in [DW07].

Symmetry reduction [CJEF96] aims at finding behavioural symmetries to subsequently reduce the model. A similar approach is followed in [RW94] in the context of sequential composition.

Another abstraction technique is *cone-of-influence reduction* [Kur94]. Based on a certain property under interest, this technique aims at eliminating all specification elements

which do not influence the verification property. For that purpose, the dependence structure of the specification is computed and analysed. A similar technique is *program slicing* [Wei81] which was successfully applied in the context of CSP-OZ [Brü08], as already mentioned in Section 2.3.

Symbolic model checking [JEK+90, McM93] aims at representing the state space in a canonical and more efficient form by means of a boolean encoding (ordered binary decision diagrams, [Bry86]). Many existing model checkers work on a symbolic representation of the original state machine and by using symbolic model checking algorithms.

Bounded model checking,[BCCZ99] as one branch of symbolic model checking [CBRZ01], incrementally tries to find finite prefixes of counterexamples by examining paths up to a certain bound k. If no counterexample is found, the bound is incremented, and the algorithm continues. The bounded model checking problem can efficiently be reduced to the propositional satisfiability problem (SAT) [DP60].

3.2 Compositional Reasoning

The strategy to cope with the state explosion problem we focus on is *compositional verification*. Many systems are not defined as one single large component but more likely composed of smaller parts. Compositional verification [dRHH+01] uses this property by means of a "divide and conquer" approach: instead of verifying the system as a whole, the verification task is split up into smaller subtasks. The components of the system are verified independently, and the verification results are combined.

The benefits are evident: instead of computing the global state space of the overall system, compositional verification merely needs to deal with the individual state spaces of the system components and thus avoids the state explosion problem up to a certain extent.

There are a lot of different compositional proof strategies [BCC98] but the most popular ones are based on the *assume-guarantee paradigm* [FP78, Jon83, MC81]: since, in general, a system component S depends on its environment, it cannot be verified in isolation. However, if a certain *environment assumption* A is assumed for S, a *guarantee condition* G of S can be inferred. Typically, this is expressed by a logical triple $\langle A \rangle\ S\ \langle G \rangle$, stating that if S is part of an overall system satisfying A, then the system must guarantee G.

Assume guarantee reasoning uses the previously described paradigm in terms of *inference rules*. In our context and in the context of [BGP03], A, S and G represent labelled transition systems. Thus, we may let $\mathcal{L}(A)$ denote the *language* of the assumption A, that is, its set of traces over Σ^* on the underlying LTS, where Σ denotes the trace alphabet of A. Furthermore, let $\mathcal{L}(A)^C$ denote the complement of this language, that is, $\mathcal{L}(A)^C = \Sigma^* \setminus \mathcal{L}(A)$.

Next, we present the different proof rules, which we will deal with in this thesis.

3.2.1 Assume Guarantee Proof Rules

Proof rules adhering to the assume-guarantee paradigm can be classified into different categories. Suppose an overall system S to be composed of two components S_1 and

S_2 running in parallel: $S = S_1 \mid\mid S_2$. The simplest assume-guarantee proof rule can be described as follows: if component S_1 guarantees (satisfies) an assumption A, and if component S_2 satisfies a property *Prop* under the assumption A, then the overall system $S_1 \mid\mid S_2$ satisfies *Prop*. This can be denoted as an inference rule as given in Figure 3.1.

$$\frac{\langle \mathit{true} \rangle\ S_1\ \langle A \rangle \qquad \langle A \rangle\ S_2\ \langle Prop \rangle}{\langle \mathit{true} \rangle\ S_1 \mid\mid S_2\ \langle Prop \rangle}$$

Figure 3.1: Basic assume-guarantee proof rule **(B-AGR)**

$$\frac{\langle A_1 \rangle\ S_1\ \langle Prop \rangle \qquad \langle A_2 \rangle\ S_2\ \langle Prop \rangle \qquad \mathcal{L}(A_1)^C \cap \mathcal{L}(A_2)^C = \varnothing}{\langle \mathit{true} \rangle\ S_1 \mid\mid S_2\ \langle Prop \rangle}$$

Figure 3.2: Parallel assume-guarantee proof rule **(P-AGR)**

From now on, this rule will be denoted by **(B-AGR)** and it will be called the *basic* assume-guarantee proof rule. It can be classified as being *sequential* in the sense that the first premise, $\langle \mathit{true} \rangle\ S_1\ \langle A \rangle$, needs to be evaluated before the second premise, $\langle A \rangle\ S_2\ \langle Prop \rangle$, can be considered – A must already be determined before it can serve as an assumption for S_2.

Another proof rule is motivated by the need for a *symmetric* computation of assumptions for both components. One particular symmetric proof rule is given in [BGP03] and depicted in Figure 3.2. In contrast to the basic proof rule, we call this rule the *parallel* proof rule and refer to it as **(P-AGR)**.

The main difference to rule **(B-AGR)** is the usage of one additional premise and assumption. Moreover, the rule allows for a parallel computation of the first and second premise, since both assumptions do not appear on the right hand side of both logical triples.

The first premise states that under the assumption A_1, component S_1 satisfies *Prop*. The second premise states the corresponding for A_2 and S_2. In order to show that $S_1 \mid\mid S_2$ satisfies *Prop*, we need a third premise: the intersection of the complements of both assumption languages needs to be empty.

In [BGP03], the authors show that the third premise, which is equivalent to $\mathcal{L}(A_1) \cup \mathcal{L}(A_2) = \Sigma^*$, is indeed necessary. The intuitive reason can roughly be described as follows: A_1 restricts S_1 to *Prop* and A_2 restricts S_2 to *Prop*, whereas the conclusion states that $S_1 \mid\mid S_2$ satisfies *Prop* without any restriction. Thus, the unification of the languages of both assumptions must contain all possible words. This ensures that no possible behaviour is ruled out by both assumptions at the same time.

A third class of assume-guarantee proof rules are referred to as *circular* proof rules. These rules either involve circularity on the assumptions or, as in our case, on the components: one circular proof rule as introduced originally in [GL91] is depicted in Figure 3.3 which we will refer to as rule **(C-AGR)**. Here, two premises coincide on their component. In general, in comparison to non-circular ones, proving soundness and completeness of circular proof rules is rather difficult [Mai03].

In this thesis, we will focus on non-circular rules and on the the first two proof rules,

$$\frac{\begin{array}{c}\langle \mathit{true}\rangle\ S_1\ \langle A_1\rangle \\ \langle A_1\rangle\ S_2\ \langle A_2\rangle \\ \langle A_2\rangle\ S_1\ \langle Prop\rangle\end{array}}{\langle \mathit{true}\rangle\ S_1\ ||\ S_2\ \langle Prop\rangle}$$

Figure 3.3: Circular assume-guarantee rule **(C-AGR)**

rules **(B-AGR)** and **(P-AGR)**. For an application of these proof rules, one needs to identify appropriate assumptions.

3.2.2 Obstacles to the Application of Assume Guarantee Reasoning

Several issues complicate the application of assume-guarantee reasoning. First, the system needs to be composed of several components. If this is not the case, assume-guarantee reasoning is not applicable at all.

Furthermore, the identification of environment assumptions had to be done manually by the user. By the use of a new technique based on a learning approach and proposed in [CGP03], this process can now fully be automated. We will introduce the approach in the next section.

Even though that automated learning of an assumption removes one of the obstacles assume-guarantee reasoning has to deal with, its usefulness in comparison to monolithic verification is still questionable: the major aim of this technique is to explore smaller state spaces. However, an unadvantageous decomposition can still lead to large assumptions and thus large state spaces. In [CAC06], the authors investigated the effectiveness of assume-guarantee reasoning based on exploring different decompositions of a given system and comparing memory usage. The results show that only in very few cases, assume-guarantee reasoning indeed outperforms non-compositional verification. Even worse, in most cases, the explored state spaces are actually larger in compositional verification.

In particular in the context of compositional verification for formal methods, these considerations motivate the need for a technique to define decompositions which are advantageous for an application of assume-guarantee-based techniques. We address this problem in Chapter 6.

3.2.3 Learning for Compositional Verification

In order to apply an assume-guarantee-based proof rule, environment assumptions need to be identified. Consider the basic proof rule **(B-AGR)**. Unfortunately, it is a non-trivial process to find an assumption which, on the one hand, abstracts from S_1 by over-approximating it and which is, on the other hand, strong enough for S_2, such that *Prop* can be deduced. This applies to any of these proof rules.

Over many years, the development of an assumption had to be done manually by the user, not allowing assume-guarantee reasoning to be performed in an automatic manner. Recently, a new technique to fully automatically generate assumptions [CGP03] based on a *learning* algorithm [Ang87] has been developed. The core idea for this technique is to use a model checker to learn the assumption. This technique can be applied with respect to several assume-guarantee proof rules [PGB+08] and in a framework, freeing the user from manual interference.

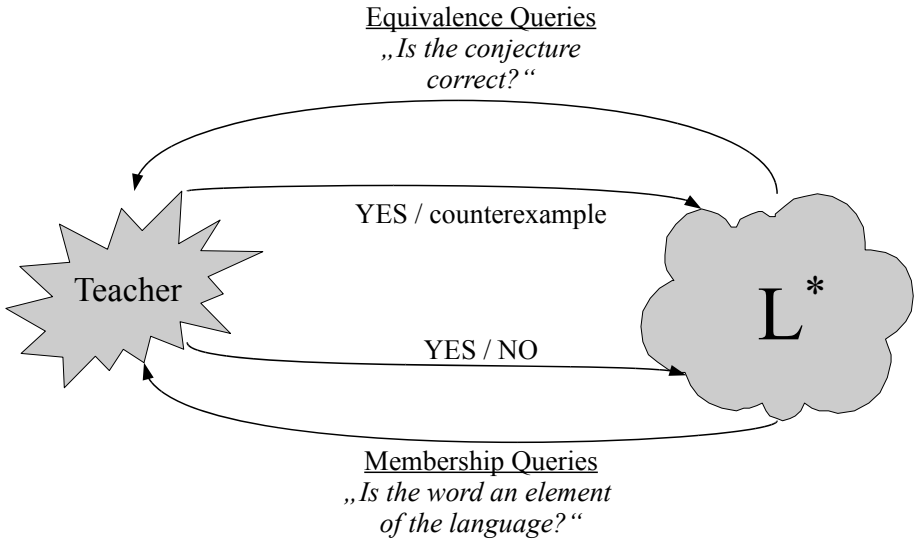


Figure 3.4: Illustration of the L* algorithm

The basis for this approach is an algorithm which learns an unknown regular language (in our case: the language of the assumption) and returns a deterministic finite automaton (DFA) accepting this language. The algorithm is called L*, and it was introduced in [Ang87]. We describe the basic idea of the algorithm: suppose that U is an unknown regular language over some alphabet Σ. For an effective learning of U, the algorithm requires an oracle which correctly answers two different questions:

Question (Membership Query):
Given a word w over the alphabet Σ, is w an element of U?
Answer:
Yes, if w is an element of U, *no* otherwise.

Question (Equivalence Query):
Does the DFA D accept the language U?
Answer:
Yes, if $\mathcal{L}(D) = U$ holds, a *counterexample* $w \in (\mathcal{L}(D) \setminus U) \cup (U \setminus \mathcal{L}(D))$ otherwise.

If the oracle (or *teacher*, as it is called in the context of L*) correctly answers this question, the algorithm always terminates and outputs a DFA D_U, such that $\mathcal{L}(D_U) = U$ holds.

Figure 3.4 illustrates this concept. The approach presented in [CGP03] incorporates the L* algorithm into an assume-guarantee-based framework for the automatic computation of the required assumptions. The technique can be applied to all three previously introduced proof rules, as shown in [PGB+08]. Here, a model checker serves as the teacher. The idea is to incrementally compute the assumption.

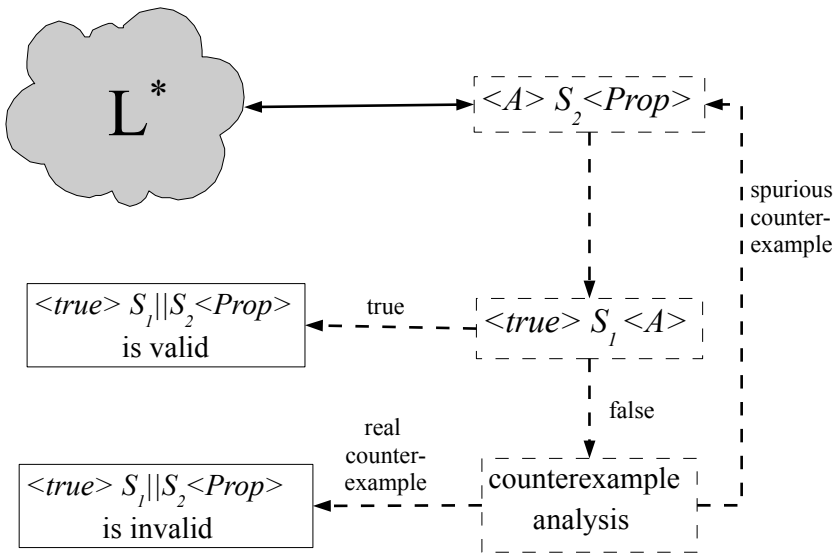


Figure 3.5: Illustration of the L* based learning framework

As an example, for the basic proof rule **(B-AGR)**, the framework starts by making use of L* to compute an assumption A such that $\langle A \rangle\ S_2\ \langle Prop \rangle$ holds. Afterwards, $\langle true \rangle\ S_1\ \langle A \rangle$ is checked.[1] If the result is true, correctness of the proof rule yields that $\langle true \rangle\ S_1 \| S_2\ \langle Prop \rangle$ holds. Otherwise, the counterexample is analysed. A spurious counterexample leads to a refinement of the verification process, a valid counterexample to the refutation of $\langle true \rangle\ S_1 \| S_2\ \langle Prop \rangle$. This is illustrated in Figure 3.5.

Next, we put assume-guarantee reasoning into our context by translating both rules, **(B-AGR)** and **(P-AGR)**, into the semantic domain of CSP-OZ. Subsequently, we show their soundness.

3.3 Assume-Guarantee Reasoning for CSP

Since our application of assume-guarantee reasoning lies in the domain of CSP-OZ specifications, we need to translate the previously identified proof rules into our context and show their correctness. Fortunately, as already explained in Chapter 2, CSP-OZ

[1] In terms of an LTS, *true* corresponds to the empty language.

specifications can be translated into semantic equivalent CSP processes. Therefore, it is sufficient to consider the semantic domain of CSP.

Verification properties can mainly be classified into two categories [OL82]: *safety* and *liveness* properties. Safety properties follow the principle of

" Nothing bad will ever happen! "

meaning that a violation of a safety property is given by a finite counterexample. In contrast, liveness properties can be described by

" Something good will eventually happen! "

describing that at some point, the property will be satisfied, not allowing to contradict a liveness property by a finite counterexample.

Our decomposition approach focuses on safety properties. This allows us to move to the domain of the CSP trace semantics instead of the more discriminating failures-divergences semantics: as explained in [Weh00] and [OW05], in contrast to liveness properties dealing with deadlock or livelock freedom, when dealing with safety properties, the CSP traces model is sufficient. An approach for verifying liveness properties in the context of compositional reasoning is, for instance, given in [CGK97]. According to this, the learning-based approach, as explained in the previous section, is also considering safety properties.

By translating assume-guarantee proof rules into the CSP traces model, a logical triple $\langle A \rangle\ S\ \langle Prop \rangle$ becomes a trace refinement condition $Prop \sqsubseteq_T A \| S$ which is by definition equivalent to $traces(A \| S) \subseteq traces(Prop)$.

We need to be more precise and consider the respective alphabets of A, S and $Prop$. Here, the alphabet of the assumption depends on the particular proof rule: for the basic rule, **(B-AGR)**, $\alpha A = (X_2 \cup Y) \cap X_1$ whereas for the parallel rule, **(P-AGR)**, $\alpha A = (X_1 \cap X_2) \cup Y$. Setting $\alpha S = X$, $\alpha Prop = Y$ and $\alpha A = \Sigma$, the condition becomes

$$Prop \sqsubseteq_T (A \ {}_{\Sigma}\|_{X}\ S) \setminus (Events \setminus Y),$$

where the right hand side processes need to be restricted to the alphabets of the left hand side processes by using hiding.

Figures 3.6 and 3.7 specify rules **(B-AGR)** and **(P-AGR)**, rephrased in terms of CSP trace refinement, where we additionally set $\alpha S_1 = X_1$ and $\alpha S_2 = X_2$.

We take a closer look at the third premise of rule **(P-AGR)**: in comparison to the work [CGP03], the authors move from the domain of labelled transitions system (LTS) to finite state machines (FSM) [BGP03] and construct the complement *co_M* of a FSM M to denote the third premise of rule **(P-AGR)** by

$$\mathcal{L}(co\_A_1 \| co\_A_2) = \varnothing$$

However, it is impossible to construct a CSP process *co_P* for some process P, accepting the complement of its language. This is based on the fact that the set of traces of a CSP process is always prefix-closed whereas its complement is not. Thus, *co_P* does not exist

$$\frac{A \sqsubseteq_T S_1 \setminus (\mathit{Events} \setminus \Sigma) \qquad \mathit{Prop} \sqsubseteq_T (A \;{}_\Sigma\|_{X_2}\; S_2) \setminus (\mathit{Events} \setminus Y)}{\mathit{Prop} \sqsubseteq_T (S_1 \;{}_{X_1}\|_{X_2}\; S_2) \setminus (\mathit{Events} \setminus Y)}$$

Figure 3.6: Rule **(B-AGR)** rephrased in terms of CSP trace refinement

$$\frac{\mathit{Prop} \sqsubseteq_T (A_1 \;{}_\Sigma\|_{X_1}\; S_1) \setminus (\mathit{Events} \setminus Y) \qquad \mathit{Prop} \sqsubseteq_T (A_2 \;{}_\Sigma\|_{X_2}\; S_2) \setminus (\mathit{Events} \setminus Y) \qquad (A_1 \,\Box\, A_2) \sqsubseteq_T \mathtt{Run}_\Sigma}{\mathit{Prop} \sqsubseteq_T (S_1 \;{}_{X_1}\|_{X_2}\; S_2) \setminus (\mathit{Events} \setminus Y)}$$

Figure 3.7: Rule **(P-AGR)** rephrased in terms of CSP trace refinement

and we use the equivalent[2] condition $\mathcal{L}(A_1)^C \cap \mathcal{L}(A_2)^C = \varnothing$. In our semantic domain of the CSP traces model, this means $\mathit{traces}(A_1)^C \cap \mathit{traces}(A_2)^C = \varnothing$. We will now show that $(A_1 \,\Box\, A_2) \sqsubseteq_T \mathtt{Run}_\Sigma$ and $\mathit{traces}(A_1)^C \cap \mathit{traces}(A_2)^C = \varnothing$ are equivalent, implying that rule **(P-AGR)** corresponds to rule 1 from [BGP03].

Lemma 3.3.1. *(Correspondence between rule **(P-AGR)** and rule 1 from [BGP03])*
Let A_1 and A_2 be two CSP processes over the alphabet Σ. Then,

$$(A_1 \,\Box\, A_2) \sqsubseteq_T \mathtt{Run}_\Sigma$$

holds, if, and only if,

$$\mathit{traces}(A_1)^C \cap \mathit{traces}(A_2)^C = \varnothing.$$

Proof.

$$\begin{array}{lll}
 & (A_1 \,\Box\, A_2) \sqsubseteq_T \mathtt{Run}_\Sigma & \\
\Leftrightarrow & \mathit{traces}(A_1 \,\Box\, A_2) = \mathit{traces}(\mathtt{Run}_\Sigma) & \text{(Definition of } \mathtt{Run}_\Sigma\text{)} \\
\Leftrightarrow & \mathit{traces}(A_1 \,\Box\, A_2) = \Sigma^* & \text{(Definition of } \mathit{traces}(\mathtt{Run}_\Sigma)\text{)} \\
\Leftrightarrow & \mathit{traces}(A_1 \,\Box\, A_2)^C = \varnothing & \\
\Leftrightarrow & (\mathit{traces}(A_1) \cup \mathit{traces}(A_2))^C = \varnothing & \text{(Definition of } \mathit{traces} \text{ for external choice)} \\
\Leftrightarrow & \mathit{traces}(A_1)^C \cap \mathit{traces}(A_2)^C = \varnothing &
\end{array}$$

□

Next, we give a small example illustrating the application of rule **(B-AGR)**.

3.3.1 Application Example: Elevator System

Figure 3.8 defines a CSP specification of a simple elevator system. It consists of two processes *Elevator* and *User*. The overall system is defined as the parallel composition of both processes, synchronising on the intersection of their alphabets

$$X_1 := \{\mathit{req\_floor}, \mathit{req\_close}, \mathit{move}, \mathit{stop}, \mathit{req\_open}\}$$

and

$$X_2 := \{\mathit{req\_floor}, \mathit{enter}, \mathit{req\_close}, \mathit{req\_open}, \mathit{leave}\}.$$

$$
\begin{aligned}
\mathit{Elevator} &\overset{c}{=} \mathit{req\_floor} \rightarrow \mathit{req\_close} \rightarrow \mathit{move} \rightarrow \mathit{stop} \rightarrow \mathit{req\_open} \rightarrow \mathit{Elevator} \\
\mathit{User} &\overset{c}{=} \mathit{req\_floor} \rightarrow \mathit{enter} \rightarrow \mathit{User} \;\Box \\
&\quad\; \mathit{req\_close} \rightarrow \mathit{User} \;\Box \\
&\quad\; \mathit{req\_open} \rightarrow \mathit{leave} \rightarrow \mathit{User} \\
\mathit{System} &\overset{c}{=} \mathit{Elevator} \;{}_{X_1}\|_{X_2}\; \mathit{User}
\end{aligned}
$$

Figure 3.8: CSP specification of a simple elevator system

The property, which we want to verify, is given as follows: a user entering the elevator (*enter*) will always lead to him leaving (*leave*) the elevator. As a CSP process, we write: $\mathit{Prop} \overset{c}{=} \mathit{enter} \rightarrow \mathit{leave} \rightarrow \mathit{Prop}$. Let $Y := \{\mathit{enter}, \mathit{leave}\}$ denote the alphabet of the property. Based on the definition of [CGP03], we get

$$\Sigma = (X_2 \cup Y) \cap X_1 = \{\mathit{req\_floor}, \mathit{req\_close}, \mathit{req\_open}\}.$$

In order to show that

$$\mathit{Prop} \sqsubseteq_T (\mathit{Elevator} \;{}_{X_1}\|_{X_2}\; \mathit{User}) \setminus (\mathit{Events} \setminus Y)$$

holds, we can apply rule **(B-AGR)** by defining

$$
\begin{aligned}
A &\overset{c}{=} \mathit{req\_close} \rightarrow A \;\Box\; \mathit{req\_floor} \rightarrow A' \\
A' &\overset{c}{=} \mathit{req\_close} \rightarrow A' \;\Box\; \mathit{req\_open} \rightarrow A
\end{aligned}
$$

Then, both premises of the rule are satisfied, that is, $\mathit{traces}(\mathit{Elevator}) \restriction \Sigma \subseteq \mathit{traces}(A)$ and $\mathit{traces}(A \;{}_{\Sigma}\|_{X_2}\; \mathit{User}) \restriction Y \subseteq \mathit{traces}(\mathit{Prop})$ hold.

3.3.2 Soundness of Assume-Guarantee Proof Rules

After translating both rules, **(B-AGR)** and **(P-AGR)**, into our setting of CSP, we need to show their soundness. In his bachelor's thesis, Wonisch [Won08] integrated the approach of [CGP03] into a framework for compositional reasoning about CSP processes, which he implemented by using the CSP model checker `FDR2` as the teacher. For that purpose, he showed the following soundness theorem for rule **(B-AGR)**:[3]

Theorem 3.3.2. *(Soundness of basic proof rule)*
Let S_1, S_2 and Prop be CSP processes. Let X_1, X_2, Y be alphabets, and let A be a CSP process

[2]This is based on $\mathcal{L}(A)^C = \mathcal{L}(\mathit{co\_A})$ and $\mathcal{L}(A \| B) = \mathcal{L}(A) \cap \mathcal{L}(B)$.
[3]We omit dealing with the technical aspect of $\checkmark$-freedom.

defined over the alphabet $\Sigma = (X_2 \cup Y) \cap X_1$*. Then, the following proof rule is sound:*

$$\frac{\begin{array}{c} A \sqsubseteq_T S_1 \setminus (\mathit{Events} \setminus \Sigma) \\ \mathit{Prop} \sqsubseteq_T (A \;{}_{\Sigma}\|_{X_2} S_2) \setminus (\mathit{Events} \setminus Y) \end{array}}{\mathit{Prop} \sqsubseteq_T (S_1 \;{}_{X_1}\|_{X_2} S_2) \setminus (\mathit{Events} \setminus Y)} \tag{3.1}$$

Proof. See [Won08], Theorem 1. □

We will now correspondingly show soundness of the parallel proof rule **(P-AGR)**.

Theorem 3.3.3. *(Soundness of parallel proof rule)*
Let S_1, S_2 *and Prop be CSP processes. Let* X_1, X_2, Y *be alphabets such that* $Y \subseteq X_1 \cup X_2$*, and let* A_1, A_2 *be CSP processes defined over the alphabet* $\Sigma = (X_1 \cap X_2) \cup Y$*. Then, the following proof rule is sound:*

$$\frac{\begin{array}{c} \mathit{Prop} \sqsubseteq_T (A_1 \;{}_{\Sigma}\|_{X_1} S_1) \setminus (\mathit{Events} \setminus Y) \\ \mathit{Prop} \sqsubseteq_T (A_2 \;{}_{\Sigma}\|_{X_2} S_2) \setminus (\mathit{Events} \setminus Y) \\ (A_1 \;\Box\; A_2) \sqsubseteq_T \mathtt{Run}_{\Sigma} \end{array}}{\mathit{Prop} \sqsubseteq_T (S_1 \;{}_{X_1}\|_{X_2} S_2) \setminus (\mathit{Events} \setminus Y)} \tag{3.2}$$

Proof. Let

$$t \in \mathit{traces}((S_1 \;{}_{X_1}\|_{X_2} S_2) \setminus (\mathit{Events} \setminus Y)).$$

We need to show $t \in \mathit{traces}(\mathit{Prop})$. The definition of *traces* for hiding ([Ros98]) yields the existence of

$$s \in \mathit{traces}(S_1 \;{}_{X_1}\|_{X_2} S_2),$$

such that $t = s \restriction Y$. Moreover, since s is defined over $X_1 \cup X_2$ and by applying the definition of $\mathit{traces}(P \;{}_X\|_Y Q)$ ([Ros98]), for $u_1 := s \restriction X_1$ and $u_2 := s \restriction X_2$, we get $u_i \in \mathit{traces}(S_i)$. Mainly corresponding to the correctness proof of the basic assume-guarantee rule, [Won08], we will now show:

(i) $s \restriction (\Sigma \cup X_1) \in \mathit{traces}(A_1 \;{}_{\Sigma}\|_{X_1} S_1)$ or (*)
$s \restriction (\Sigma \cup X_2) \in \mathit{traces}(A_2 \;{}_{\Sigma}\|_{X_2} S_2)$, (**)

(ii) $(*) \Rightarrow t_1' := s_1' \restriction Y \in \mathit{traces}(\mathit{Prop})$ and $t_1' = t$,

(iii) $(**) \Rightarrow t_2' := s_2' \restriction Y \in \mathit{traces}(\mathit{Prop})$ and $t_2' = t$,

where both, $(*)$ and $(**)$ lead to the conclusion $t \in \mathit{traces}(\mathit{Prop})$.
For property (i), let $s_i' := s \restriction (\Sigma \cup X_i)$. We first deduce

$$s_1' \restriction \Sigma = (s \restriction (\Sigma \cup X_1)) \restriction \Sigma = s \restriction \Sigma \text{ and}$$
$$s_2' \restriction \Sigma = (s \restriction (\Sigma \cup X_2)) \restriction \Sigma = s \restriction \Sigma.$$

Based on the third premise and the fact that $s \restriction \Sigma \in \mathit{traces}(\mathtt{Run}_\Sigma)$ by definition of $\mathtt{Run}_\Sigma$, we have $s \restriction \Sigma \in \mathit{traces}(A_1 \,\Box\, A_2)$. Thus, either $s \restriction \Sigma \in \mathit{traces}(A_1)$ or $s \restriction \Sigma \in \mathit{traces}(A_2)$ holds. Second, we get

$$s_1' \restriction X_1 = (s \restriction (\Sigma \cup X_1)) \restriction X_1 = s \restriction X_1 = u_1 \text{ and}$$
$$s_2' \restriction X_2 = (s \restriction (\Sigma \cup X_2)) \restriction X_2 = s \restriction X_2 = u_2,$$

with $u_1 \in \mathit{traces}(S_1)$. Both combined: if $s \restriction \Sigma = s_1' \restriction \Sigma \in \mathit{traces}(A_1)$, we use $s_1' \restriction X_1 \in \mathit{traces}(S_1)$ to deduce $s_1' \in \mathit{traces}(A_1 \;{}_\Sigma\|_{X_1}\; S_1)$. Otherwise, we get $s_2' \in \mathit{traces}(A_2 \;{}_\Sigma\|_{X_1}\; S_2)$. This concludes the proof of (i).

Next, we show (ii), (iii) is analogous. If (*) holds, $t_1' := s_1' \restriction Y \in \mathit{traces}(\mathit{Prop})$ follows immediately from the first premise. We are left to show $t_1' = t$:

$$\begin{array}{lll} & t_1' & \\ = & s_1' \restriction Y & \text{(Definition of } t_1'\text{)} \\ = & (s \restriction (\Sigma \cup X_1)) \restriction Y & \text{(Definition of } s_1'\text{)} \\ = & s \restriction ((\Sigma \cup X_1) \cap Y) & \text{(Definition of trace projection)} \\ = & s \restriction (((X_1 \cap X_2) \cup Y \cup X_1) \cap Y) & \text{(Definition of } \Sigma\text{)} \\ = & s \restriction Y & \\ = & t & \text{(Definition of } t\text{)} \end{array}$$

This concludes the proof. □

The following example [Sch09] shows that the restriction $\Sigma = (X_1 \cap X_2) \cup Y$ is indeed required.

Example 3.3.4. *Let* $S_1 = a \to a \to \mathtt{Stop}$, $S_2 = b \to b \to \mathtt{Stop}$ *and*

$$\mathit{Prop} = (a \to a \to \mathtt{Stop}) \,\Box\, (b \to b \to \mathtt{Stop}).$$

Thus, we get $X_1 = \{a\}$, $X_2 = \{b\}$ *and* $Y = \{a, b\}$. *Now assume* $\Sigma = \varnothing$ *and* $A_1 = A_2 = \mathtt{Stop}$. *Then, all three premises of the parallel rule are satisfied:*

- *We get* $(\mathtt{Stop} \;{}_\varnothing\|_{\{a\}}\; S_1) = a \to a \to \mathtt{Stop}$ *and thus* $\mathit{Prop} \sqsubseteq_T a \to a \to \mathtt{Stop}$.
- *Also,* $(\mathtt{Stop} \;{}_\varnothing\|_{\{b\}}\; S_2) = b \to b \to \mathtt{Stop}$ *and therefore* $\mathit{Prop} \sqsubseteq_T b \to b \to \mathtt{Stop}$.
- *Finally,* $\mathtt{Run}_\varnothing =_T \mathtt{Stop}$, *hence* $(\mathtt{Stop} \,\Box\, \mathtt{Stop}) \sqsubseteq_T \mathtt{Run}_\varnothing$.

However, $\mathit{Prop} \sqsubseteq_T (a \to a \to \mathtt{Stop} \;{}_{\{a\}}\|_{\{b\}}\; b \to b \to \mathtt{Stop})$ *does* not *hold as Prop does not allow the trace* $\langle a, b, a, b\rangle$ *which* $(a \to a \to \mathtt{Stop} \;{}_{\{a\}}\|_{\{b\}}\; b \to b \to \mathtt{Stop})$ *is able to conduct. In case of* $\Sigma = (X_1 \cap X_2) \cup Y = \{a, b\}$, *the third premise becomes*

$$(\mathtt{Stop} \,\Box\, \mathtt{Stop}) \sqsubseteq_T \mathtt{Run}_{\{a,b\}},$$

which is clearly not satisfied.

Summing up, we have shown the applicability of the proof rules **(B-AGR)** and **(P-AGR)** in the semantic domain of CSP. This allows us to apply compositional reasoning, based on the following decomposition approach in our context. Moreover, we can evaluate the efficiency of different decompositions by using the CSP model checker `FDR2` [For05].

3.4 Related Work

Model checking and (automated) compositional verification of specifications, written in (integrated) formal methods, is extensively researched. We give a brief overview on recent works, mainly in the context of our employed methods.

Model Checking for Formal Methods: In order to allow model checking of a software system, it needs to be specified in some formal language.

Leuschel examines LTL model checking [LMC01] for CSP by using the model checker `FDR2` [For05]. In [SW05], Smith and Wildman consider model checking of Z specifications by translating Z into the input language of the model checker SAL [BGL+00]. Derrick et al. also investigate Z model checking by using SAL in [DNS08]. Smith deals with model checking Object-Z specifications with respect to temporal logic formulae in [KS01].

CSP-Z model checking is researched in [MS01]. Model checking CSP-OZ specifications by again using `FDR2` is described in [FW99]. We use this approach in our implementation framework.

Compositional Verification for Formal Methods: Compositional verification has its early application within the scope of model checking in [EDK89] and later in [GL91, CGP99]. Proof rules for verifying real time system have been developed in [CMP94]. In the context of UML, compositional verification (and model checking) of embedded real time systems is, for instance, investigated in [SGT+03]. By defining a formal semantics for a domain specific subset of the UML, the authors allow themselves to reason about individual software components instead of the complete system.

In our context, Winter and Smith [WS03] deal with compositional verification for Object-Z. They analyse the class structure of an Object-Z specification and argument about restricted environments, allowing for the definition of a compositional proof rule. Modular reasoning of Object-Z is also investigated by Griffiths [Gri97, Gri98].

In [MG07], Moffat and Goldsmith examine compositional reasoning for CSP by identifying and showing several proof rules with respect to some CSP operators and certain structures of the overall system. Compositional reasoning for CSP is also analysed in [Moo90].

Compositional verification for *integrated* formal methods has extensively be researched in the context of CSP||B [ST02]. Amongst other works [ST04, ST05], Evans, Schneider and Treharne investigated how to decompose specifications into so-called *chunks* [STE05]. For Event-B, Butler [But09] described how to decompose specifications for independent refinement checks.

Assume-Guarantee Reasoning: Assume-guarantee reasoning was first introduced in [FP78, Jon83] and further developed in [Pnu84]. Several variants being applied in different domains, such as assumption-commitment for synchronous message

passing [MC81] and rely-guarantee for shared-variable concurrency [Jon83], exist. All of them can be subsumed under the roof of the assume-guarantee paradigm.

The book [dRHH+01] gives a profound overview on compositional reasoning and the assume-guarantee paradigm in particular.

Automated Compositional Reasoning: Ever since the introduction of compositional reasoning, one of the major goals is to fully automate this verification process. The idea to automatically generate assumptions in the context of assume-guarantee reasoning was first proposed in [GPB02].

Learning assumptions for compositional reasoning was introduced in [CGP03] and initially with respect to the basic proof rule **(B-AGR)**. The following paper [BGP03] extended the idea to symmetric proof rules, such as rule **(P-AGR)**. Apart from these authors, several other articles investigate this particular field of research: [GP08] contains a selection of articles on learning techniques for automated assume-guarantee reasoning. Nam and Alur [AMN05, NA06, Nam07] investigate L^*-based learning of assumption in the context of symbolic model checking. In the same context, the article [APR+01] presents a SAT-based technique for lazy learning of assumptions. Several articles concentrate on the optimisation of the L^* algorithm to more effectively compute the assumptions [GGP07, GMF07, CS07, CS08].

Besides the application of learning in the area of model checking, the L^* algorithm is used in several other software verification domains. For instance, in [CCST05], the authors use assumption learning in the context of simulations. Alur et al. [AČMN05] tackle synthesis of interface specifications based on learning. In the context of black box checking, that is, verifying a software system without a model, L^* is used to learn an unknown system [GPY02].

4 Decomposition of a Specification

Contents

4.1 Overview 56

4.2 Cut of a Dependence Graph 58

4.2.1 Fragmentation of the Control Flow Graph 58

4.2.2 Correctness Criteria for the Fragmentation 61

4.2.3 Definition of a Cut 66

4.2.4 Candy Machine Revisited: Cut of the Dependence Graph 70

4.3 Decomposing CSP-OZ Specifications 72

4.3.1 Intermediate Definition of the Decomposition 75

4.3.2 Preservation of the Data Dependences 81

4.3.3 Preservation of the Control Flow 86

4.3.4 Renaming for the Decomposition 98

4.3.5 Definition of the Decomposition 100

4.3.6 Candy Machine Revisited: Decomposition 101

4.3.7 Improvement of the Decomposition 103

4.4 Decomposition for the General Case: Number Swapper 106

4.5 Related Work 109

As previously stated, we focus on decomposing specifications, allowing for an efficient application of compositional reasoning. To this end, we analyse a specification's dependence structure by means of its dependence graph, as defined in Chapter 2.

The following core chapter presents the correctness criteria and the definitions for the decomposition of CSP-OZ specifications. Before going into the technical details, we start by outlining our approach in Section 4.1. Section 4.2 defines and illustrates the fragmentation of a dependence graph, denoted as *cut*. The fragmentation is based on certain correctness criteria, resulting in the decomposition of the specification itself, as introduced in Section 4.3. A special case of the definition will be illustrated by means of the case study from Chapter 2. Additionally, Section 4.4 introduces a second, smaller case study, exemplifying the *general* case of a decomposition. In the final section, we discuss related work.

In order to facilitate an illustrative and fluent description of the approach, we postpone most of the correctness proofs to the next chapter.

4.1 Overview

Compositional verification follows a "divide and conquer" approach: to cope with the state explosion problem, a local verification with respect to the components of a software model is applied.

However, as already stated in Section 3.2.2, two major obstacles complicate the application of compositional verification and particularly assume-guarantee reasoning.

First, the technique is only applicable if the overall model is composed of at least two components. If this is not the case, the model needs to be decomposed, without changing its observable behaviour.

Less evident, second, a decomposition itself does not always lead to an effective application of compositional verification. Disadvantageous decompositions may still cause large state spaces during model checking. We will deal with the aspect of classifying decompositions in Chapter 6.

In this chapter, we *construct* decompositions of specifications written in CSP-OZ, preserving the specification's semantics in the domain of the CSP traces model. As the dependence graph comprises the complete dependence structure of a specification S, our strategy primarily targets the distribution of the DG. Henceforward, S itself is decomposed such that the resulting specification parts S_1 and S_2 correspond to the generated segments of the DG.

A distribution of the DG is accomplished on the level of its *operation* nodes. Correctness criteria refer to the control flow and thus to CSP operator nodes as well. In order to fragment the DG into two subgraphs, we define a set $\mathbf{C} \subseteq \mathtt{op}(N)$, which serves as the link between them. We will call this set a *cut* motivated by the intuition that it identifies the line(s) of intersection of the graph. The set of cut nodes is common to both subgraphs and, consecutively, to both specification parts. From the specification point of view, the cut serves as the *interface*, that is, the synchronisation alphabet, between the specification parts S_1 and S_2.

Figure 4.1 illustrates the individual steps of our approach.

Computation of the DG, ①: Given a specification S, we first compute its dependence graph $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$, as introduced in Chapter 2. We mainly focus our considerations on its set of *operation* nodes.

Identification of the Cut, ②: Next, we identify a *cut* of the dependence graph: a set of operation nodes, yielding a correct fragmentation of the DG (represented by grey nodes in the figure). In Section 4.2, we present the definition of a cut along with the correctness criteria for the segmentation.

Fragmentation of the DG, ③: Determining the set of cut nodes and distributing the set of operation nodes results in two subgraphs. The cut itself is represented in both subgraphs.

Decomposition of the Specification, ④: The fragmentation of the DG leads to the definition of the two specification parts of S, S_1 and S_2. Section 4.3 precisely defines

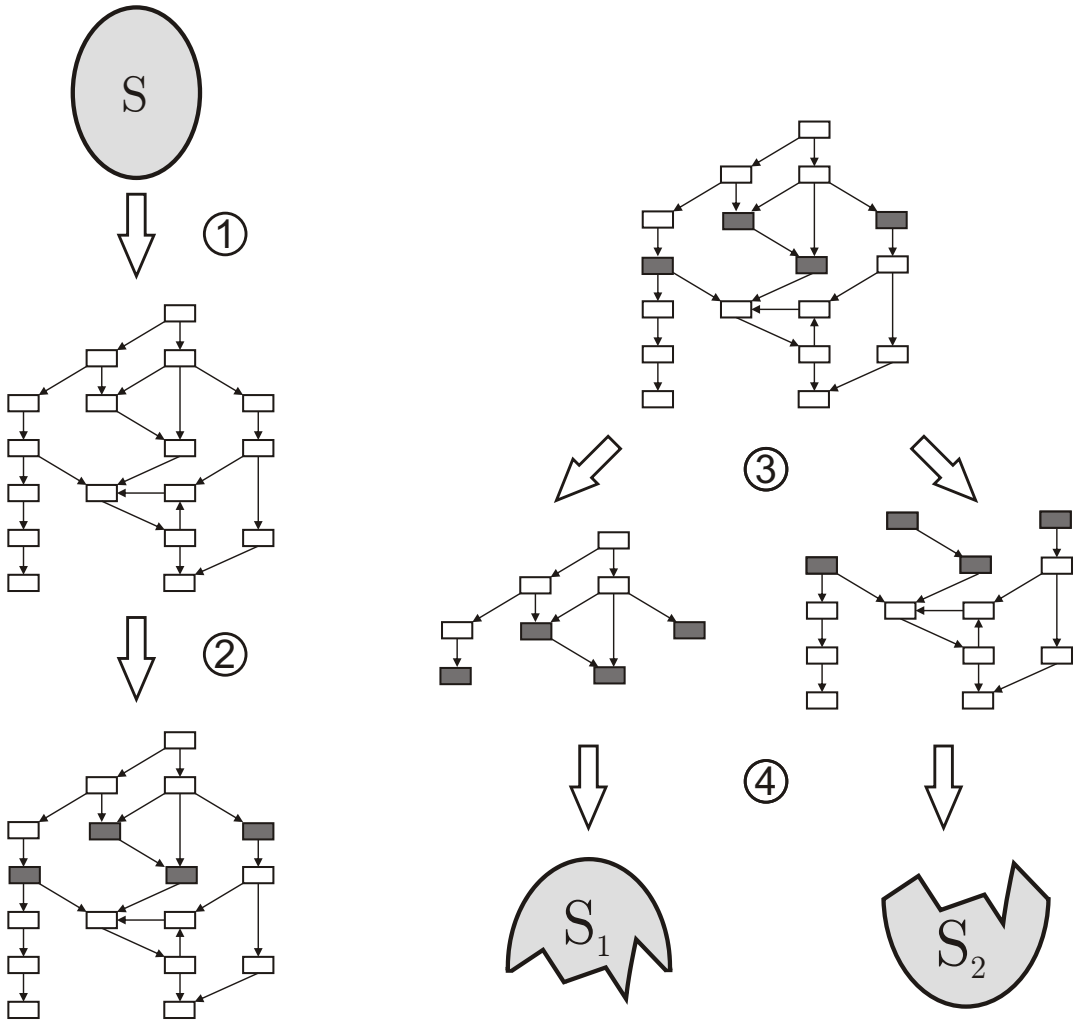


Figure 4.1: Cut identification, fragmentation of the dependence graph and decomposition of the specification

the decomposition and introduces the additional constructs required to ensure the (trace) equivalence of S and $S_1 \parallel S_2$.

Next, we introduce our definition of a cut along with the criteria which need to be satisfied such that the observable behaviour of the specification is preserved. We illustrate the definitions and criteria on several small examples and especially on our case study.

4.2 Cut of a Dependence Graph

Before we introduce the definition of a *valid cut* of a dependence graph, we start with identifying its *fragmentation* with respect to two sets of operation nodes. Correctness criteria on the fragmentation consecutively lead to the definition of the cut. Since most of our definitions are not restricted to the dependence graph, we introduce them for arbitrary graphs and subsequently apply them in our specific context.

4.2.1 Fragmentation of the Control Flow Graph

We are interested in identifying two different subgraphs of the DG. In particular, these subgraphs should not arbitrarily intersect. Thus, we need to define different *segments* of the graph which are disjoint.

The control flow graph comprises all nodes of the dependence graph and defines the workflow and the dynamic behaviour of a specification. Therefore, we will define a fragmentation of the control flow graph alone instead of considering the dependence graph. Subsequently, the data flow needs to be evaluated to verify that a corresponding fragmentation of the DG is *correct*.

In general, the technique needs to deal with all different kinds of nodes and edges. However, the subsequent distribution of nodes refers to *operation* nodes, which is sufficient in our context: we do not distribute the set of CSP operator nodes. This will be achieved in Section 4.3, where we define a *projection* of a CSP process with respect to a set of events.

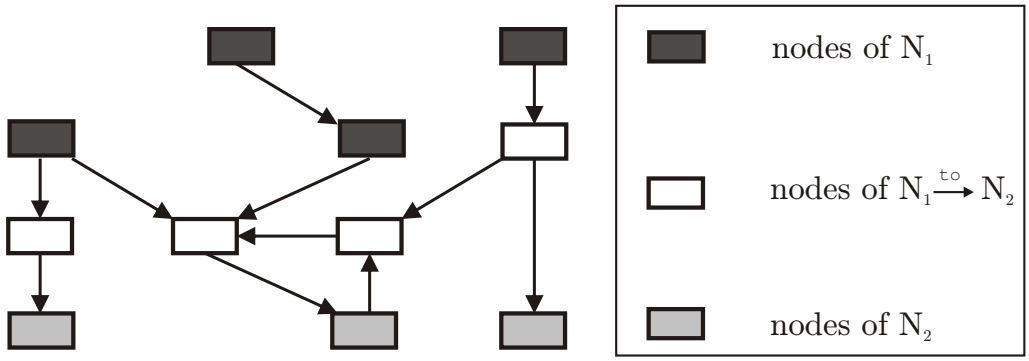


Figure 4.2: Illustration of Definition 4.2.1

First, the following definition determines all nodes reachable from one set of nodes N_1 not intersecting with another set of nodes N_2.

Definition 4.2.1. *(Interval from N_1 to N_2)*
Let $\mathtt{G} = (N, \longrightarrow)$ be a graph, and let $N_1, N_2 \subseteq N$. Then,

$$N_1 \xrightarrow{\mathtt{to}} N_2 := \{\mathsf{n'} \in N \mid \exists\, \mathsf{n} \in N_1, \pi \in \mathtt{path_G}(\mathsf{n}, \mathsf{n'}) \bullet \pi \cap N_2 = \varnothing\} \setminus N_1.$$

The interval *excludes* both, N_1 and N_2, as illustrated in Figure 4.2. Intuitively, it can be regarded as the set of nodes "between" N_1 and N_2. Note that both, N_1 and N_2, are allowed to be empty.

The previous definition allows us to divide the set of nodes of a graph into several subsets (or *phases*, as we call them). Next, we introduce the *fragmentation* of the CFG, which is defined with respect to two sets of operation nodes, $\mathbf{C}_1$ and $\mathbf{C}_2$:

Definition 4.2.2. *(Fragmentation of the control flow graph)*
Let $\mathtt{CFG}_S = (N, \longrightarrow)$ *be the control flow graph of a specification S, and let* $\mathbf{C}_1, \mathbf{C}_2 \subseteq \mathtt{op}(N)$. *Moreover, let*

$$\mathsf{StartNodes} := \{\mathsf{start}.P \mid P \in L^{CSP}\}$$

and

$$\mathsf{start}_1 := (\{\mathsf{start}\} \xrightarrow{\mathtt{to}} \mathbf{C}_1) \cap \mathsf{StartNodes}.$$

A fragmentation *of (the set of operation nodes of)* $\mathtt{CFG}_S$ *with respect to a tuple* $(\mathbf{C}_1, \mathbf{C}_2)$ *is a set of three phases* $\mathbf{Ph}_1$, $\mathbf{Ph}_2$ *and* $\mathbf{Ph}_3$ *defined as*

1.) $\mathbf{Ph}_1 := ((\{\mathsf{start}\} \xrightarrow{\mathtt{to}} \mathbf{C}_1) \cap \mathtt{op}(N)) \cup \{\mathsf{init}\}$, *(Phase 1)*

2.) $\mathbf{Ph}_2 := (\mathbf{C}_1 \xrightarrow{\mathtt{to}} \mathbf{C}_2) \cap \mathtt{op}(N)$, *(Phase 2)*

3.) $\mathbf{Ph}_3 := (\mathbf{C}_2 \xrightarrow{\mathtt{to}} \mathsf{start}_1) \cap \mathtt{op}(N)$. *(Phase 3)*

$\mathbf{C}_1$ and $\mathbf{C}_2$ serve as the two lines of intersection for the graph. The first phase $\mathbf{Ph}_1$ contains all operation nodes *before* the first line of intersection. We add the unique init-node of the specification to $\mathbf{Ph}_1$, comprising the set of initial predicates.

The second phase includes the set of operation nodes *between* both lines of intersection. Finally, the third phase comprises the set of operation nodes *behind* the second line of intersection. A first correctness criterion will exclude that any two of the five sets have a common element.

Intuitively, one would expect $\mathbf{Ph}_3 := (\mathbf{C}_2 \xrightarrow{\mathtt{to}} \varnothing) \cap \mathtt{op}(N)$. However, we need to "stop" adding nodes to $\mathbf{Ph}_3$ after reaching a *recursive call* back to $\mathbf{Ph}_1$. Otherwise, our subsequently defined correctness criteria on a fragmentation would rule out allowed recursive calls. Therefore, we define a set start_1, comprising all nodes $\mathsf{start}.X$ occurring before the first line of intersection. This specific point will become clearer in the next section.

In the general case of a *cut*, as introduced in Section 4.2.3, we use the previous definition as follows: we determine two sets of operation nodes, namely $\mathbf{C}_1$ and $\mathbf{C}_2$, which will from now be called the *first cut* and the *second cut*. The definition results in five disjoint sets of operation nodes $\mathbf{Ph}_1$, $\mathbf{C}_1$, $\mathbf{Ph}_2$, $\mathbf{C}_2$ and $\mathbf{Ph}_3$. Henceforth, we will refer to $\mathbf{Ph}_1$, $\mathbf{Ph}_2$ and $\mathbf{Ph}_3$ as the *phases* of a fragmentation, whereas $\mathbf{C}_1$ and $\mathbf{C}_2$ will be referred to as its *cut sets*.

The following lemma states that a fragmentation of the CFG is always complete in the sense that no nodes are left out:

Lemma 4.2.3. *(Completeness of Fragmentation)*
Let $\mathtt{CFG_S} = (N, \longrightarrow)$ *be the control flow graph of a specification S, and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a fragmentation. Then,*

$$\mathbf{Ph}_1 \cup \mathbf{C}_1 \cup \mathbf{Ph}_2 \cup \mathbf{C}_2 \cup \mathbf{Ph}_3 = \mathtt{op}(N).$$

Proof. The left-to-right inclusion is obvious. For the opposite inclusion, let $\mathsf{n} \in \mathtt{op}(N)$. Based on the definition, the special init-node is an element of $\mathbf{Ph}_1$. Moreover, as any CFG node is reachable from the unique start-node, there exists $\pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{start}, \mathsf{n})$. Without loss of generality let $\pi = \langle \mathsf{start}, \mathsf{n}_1, \ldots, \mathsf{n}_k \rangle$ and $\mathsf{n}_k = \mathsf{n}$.

If $\mathsf{n} \in \mathbf{Ph}_1$, we immediately deduce the right-to-left inclusion. Otherwise, $\pi \cap \mathbf{C}_1 \neq \varnothing$ holds. $\mathsf{n} \in \mathbf{C}_1$ would again conclude the proof. If $\mathsf{n} \notin \mathbf{C}_1$, there exists an index $1 \leq l_1 < k$ such that $\mathsf{n}_{l_1} \in \mathbf{C}_1$. Since n is reachable from n_{l_1}, either $\mathsf{n} \in \mathbf{Ph}_2$ or, otherwise, there exists $\mathsf{n}_{l_2} \in \mathbf{C}_2$ for some $l_1 < l_2 \leq k$. If $l_2 = k$, we have shown $\mathsf{n} \in \mathbf{C}_2$. In the opposite case, we deduce that n is reachable from $\mathbf{C}_2$ which either leads to $\mathsf{n} \in \mathbf{Ph}_3$ or to $\pi \cap \mathsf{start}_1 \neq \varnothing$ based on the definition of $\mathbf{Ph}_3$. In this case, we infer that there exists some $l_3 > l_2$ and $\mathsf{n}_{l_3} \in \mathsf{start}_1$. Here, $l_3 \neq l_2$ since $l_2 \in \mathtt{op}(N)$ and $l_3 \in \mathsf{StartNodes}$. Hence, the path $\langle \mathsf{start}, \ldots, \mathsf{n}_{l_3} \rangle$ contains at least three different nodes n_{l_1}, n_{l_2} and n_{l_3}.

Reapplication of the previous ideas now starting in n_{l_3} yields a sequence of nodes which continuously traverses the CFG through its five fragments. As the length of the sequence increases with every cycle, but never leaves the set $\mathbf{Ph}_1 \cup \mathbf{C}_1 \cup \mathbf{Ph}_2 \cup \mathbf{C}_2 \cup \mathbf{Ph}_3$, it eventually reaches n, yielding the right-to-left inclusion. □

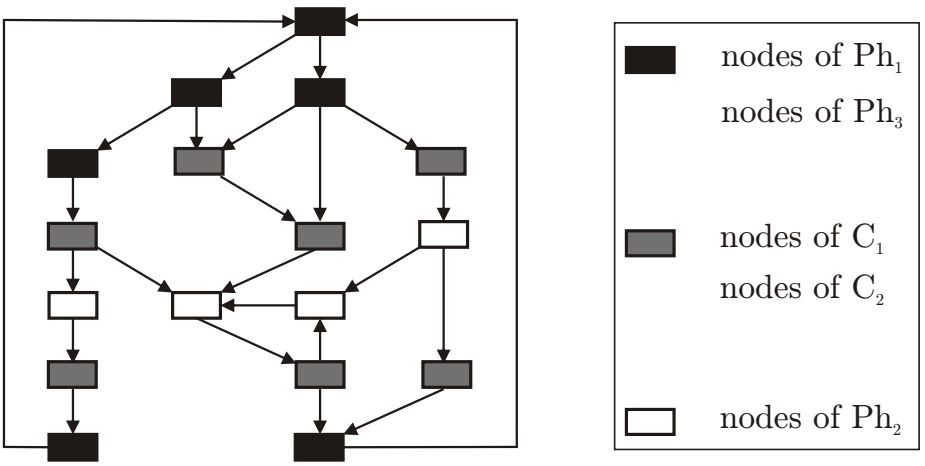


Figure 4.3: Fragmentation of the DG

Figure 4.3 illustrates the fragmentation. As already mentioned, we do not deal with nodes of $\mathtt{cf}(N)$ here. Thus, all boxes denote operation nodes. Besides, nodes of $\mathbf{Ph}_1$ and $\mathbf{Ph}_3$ have the same colour, since both segments will be assigned to the same component in Section 4.3. Hence, we will mostly not distinguish between $\mathbf{Ph}_1$ and $\mathbf{Ph}_3$.

Since the definition of the fragmentation cannot be arbitrary, we need to specify additional correctness constraints. These criteria coarsely describe the following aspects:

Criterion 1 – disjointness: All fragments are disjoint.

Criterion 2 – no crossing: The lines of intersection (cut sets) are not circumvented by data dependence edges.

Criterion 3 – no reaching back: Paths of the CFG have to comply to the ordering of the fragments.

Criterion 4 – all-or-none: The set of operation nodes corresponding to the same schema must not be distributed over different fragments.

We give a detailed definition of the correctness criteria next.

4.2.2 Correctness Criteria for the Fragmentation

In order to define a *correct* fragmentation of the DG and ultimately a correct decomposition of the specification, several correctness criteria need to be satisfied. If possible, a criterion will again be defined for arbitrary graphs.

Most of the criteria will rule out specific edges of the DG with respect to the fragmentation. We illustrate these edges by means of a recurrent figure. Recall that nodes of the cut sets and phases are always operation nodes, that is, elements of $\mathtt{op}(N)$.

Criterion 1: disjointness

As a first and straightforward correctness criterion, we require that all segments resulting from the graph fragmentation are *pairwise disjoint*. Intuitively, this is motivated by the fact that we aim at a *partitioning* of the dependence graph. We recall the set theoretical definition for disjointness:

Definition 4.2.4. *(***disjointness***)*
Let $\mathtt{G} = (N, \longrightarrow)$ *be a graph, and let* $N_1, N_2 \subseteq N$*. Then,* N_1 *and* N_2 *satisfy* **disjointness***, if, and only if,* N_1 *and* N_2 *are disjoint, that is,* $N_1 \cap N_2 = \varnothing$*.*

The definition of a cut will comprise the condition that $\mathbf{Ph}_1$, $\mathbf{C}_1$, $\mathbf{Ph}_2$, $\mathbf{C}_2$ and $\mathbf{Ph}_3$ are pairwise disjoint. Based on the construction of the different phases, this constraint is particularly related to CFG paths, as it *excludes* several edges between the different segments. For instance, as $\mathbf{Ph}_1$ and $\mathbf{Ph}_2$ have to be disjoint, a direct edge from a node of $\mathbf{Ph}_2$ to a node of $\mathbf{Ph}_1$ is impossible: the definition of $\mathbf{Ph}_2$ yields that the target node would be an element of $(\mathbf{Ph}_1 \cap \mathbf{Ph}_2)$.

Figure 4.4 illustrates that CFG edges with the source node in $\mathbf{Ph}_3$ ($\mathbf{Ph}_2$) and the target node in $\mathbf{Ph}_2$ ($\mathbf{Ph}_1$) are not allowed. Note that edges in the opposite direction are already ruled out by definition of the fragmentation. Further note that we *intentionally* allow edges connecting nodes of $\mathbf{Ph}_3$ and $\mathbf{Ph}_1$. This substantiates the definition of $\mathbf{Ph}_3$.

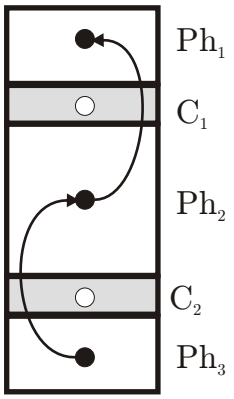


Figure 4.4: Disallowed control flow edges based on disjointness

Criterion 2: no crossing

The second correctness criterion tackles the previously described aspect of a cut identifying the *lines of intersection* of the dependence graph. Since it is generally impossible to decompose the graph into two completely independent (that is, unconnected) subgraphs, the cut needs to serve as the link between them. Intuitively, this link should not be evaded when switching from one subgraph to the other. Therefore, paths of the DG

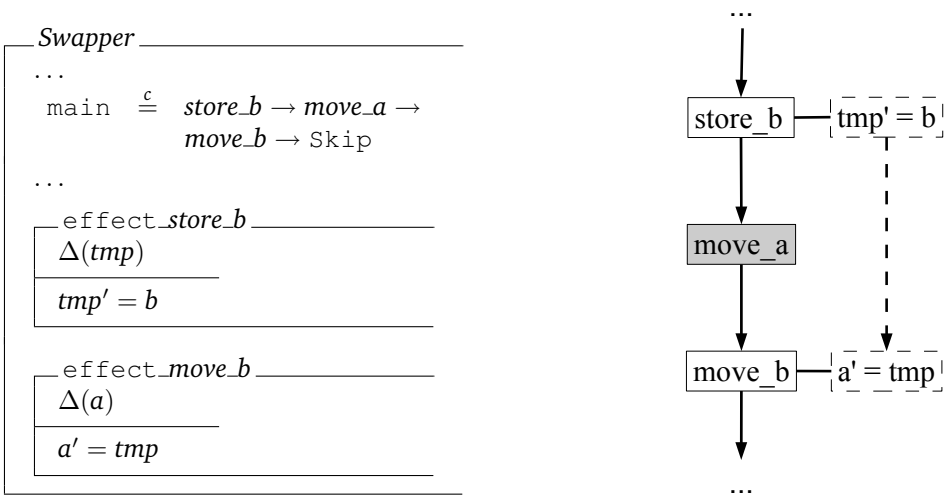


Figure 4.5: Motivation for the correctness criterion **no crossing**

must not circumvent the cut. Based on our fragmentation of the CFG and the criterion **disjointness**, we implicitly ensure this for control flow edges. However, we also need to guarantee that *data dependence edges* do not evade the cut as well: on the level of

the underlying specification, the set of operation schemas of the cut defines the *interface* between both resulting specification parts. If the behaviour of the specification parts depends on each other, these shared operations are responsible for preserving the mutual influence. This will be achieved by using them as *transmitters* for the correct values of modified state variables. If a data dependence circumvents the cut, it would be impossible to transmit the influence of one component on the other.

As an example, recall the small specification for a number swapper from Chapter 2, Figure 2.8. The modification of *tmp* within *store_b* and the reference to *tmp* within *move_b* yields a direct data dependence from the first to the latter operation node. Choosing the set $\{move\_a\}$ as the set of cut nodes is not reasonable: the modified value of *tmp* cannot be transmitted. In this case, the data dependence edge circumvents the cut as illustrated in Figure 4.5.

In general, we have to disallow data dependence edges connecting the different fragments of the dependence graph if neither of the involved nodes is an element of the cut. These edges *cross* the cut in the sense that there exists a direct link between different sides of the cut. This motivates the following definition of a predicate called **no crossing**, which we will subsequently use with respect to $(\mathbf{Ph}_1 \cup \mathbf{Ph}_3)$ and $\mathbf{Ph}_2$:

Definition 4.2.5. *(**no crossing**)*
Let $\mathtt{G} = (N, \longrightarrow)$ *be a graph, and let* $N_1, N_2 \subseteq N$. *Then,* $\mathtt{noCr}(N_1, N_2, \mathtt{G})$*, if, and only if,*

$$\nexists \mathsf{n}_1 \in N_1 \; \nexists \mathsf{n}_2 \in N_2 \bullet \mathsf{n}_1 \longrightarrow \mathsf{n}_2 \vee \mathsf{n}_2 \longrightarrow \mathsf{n}_1$$

This condition will be called **no crossing** between N_1 and N_2.

For the definition of the cut, we require $\mathtt{noCr}((\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \mathbf{Ph}_2, \mathtt{DDG_S})$. The disallowed data dependence edges are illustrated in Figure 4.6.

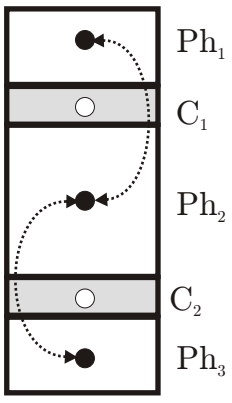


Figure 4.6: Disallowed data dependences based on **no crossing**

Criterion 3: no reaching back

The next constraint needs to be defined with respect to the DG of a specification since here, we explicitly need to refer to operation nodes and CSP operator nodes.

First, we consider the control flow graph and its fragmentation: the two lines of intersection, namely $\mathbf{C}_1$ and $\mathbf{C}_2$, dissect the graph into several fragments. We require that paths of the control flow graph need to comply to the *ordering* of the segments as follows: any path of the CFG starts in start and either remains in $\mathbf{Ph}_1$ or subsequently reaches $\mathbf{C}_1$. Consecutively, the path remains in the respective segment or advances to either $\mathbf{Ph}_2$ or directly to $\mathbf{C}_2$. Next, the path may reach $\mathbf{Ph}_3 \cup \mathbf{Ph}_1$ or immediately $\mathbf{C}_1$. Following up on this, all paths need to comply with the ordering $\mathbf{Ph}_1, \mathbf{C}_1, \mathbf{Ph}_2, \mathbf{C}_2, \mathbf{Ph}_3$, possibly repeated. Phases are potentially skipped in between.

Thus, we generally allow the control flow to advance with respect to the ordering of the segments or to remain in a segment. However, a path must not directly return to a *previous fragment*.

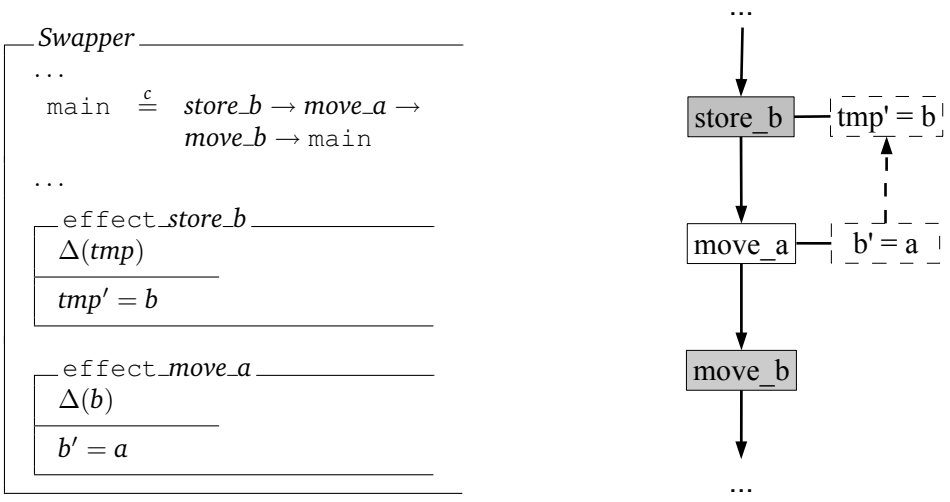


Figure 4.7: Motivation for the correctness criterion **no reaching back**

The application of the following criterion is two-folded: besides the fact that paths of the control flow graph should comply with the ordering of its segments, we also consider data dependences. For them, the motivation for this constraint is similar to **no crossing**, which already excludes a skipping of the *cut sets*. In addition, we need to exclude data dependences, returning to a previous segment.

Recall the example from Figure 2.8 with a small modification: we replace `Skip` with a recursive call of `main`. The modification of *b* within *move_a* and the reference to *b* within *store_b* yields a direct data dependence from the first to the latter operation node. Choosing the sets $\{store\_b\}$ and $\{move\_b\}$ as the sets of cut nodes is not reasonable: in this case, the data dependence edge reaches back to the first cut as illustrated in Figure

4.7. The modified value of *b* cannot be transmitted in between.

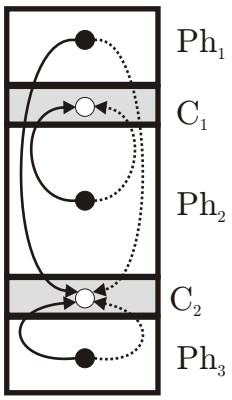


Figure 4.8: Disallowed edges based on **no reaching back**

It is sufficient to disallow edges reaching back to a *cut* segment: control flow edges reaching back from the cut to the previous phase are already excluded by definition of the fragmentation and the criterion **disjointness**. Moreover, corresponding data dependences do *not* need to be excluded. Figure 4.8 shows the additionally disallowed edges of the DG.

In order to formally express that a CFG path or a data dependence edge must not return to a previous segment, we define a predicate **no reaching back** which inputs two sets of operation nodes: the first set denotes the source nodes, the second set the target nodes. Data dependence edges must not connect the first to the latter set of nodes, the same needs to hold for control flow edges. As CFG paths from one operation node to another possibly comprise CSP operator nodes in between, we need to rule out those paths from the first to the latter set of nodes without operation nodes in between. Recall that

$$\mathsf{n} \xrightarrow{\bullet} \mathsf{n'}, \text{ if, and only if, } (\exists\, \pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'}) \bullet \pi \cap \mathtt{op}(N) = \{\mathsf{n}, \mathsf{n'}\}).$$

Definition 4.2.6. *(***no reaching back***)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph of a specification S, and let* $N_1, N_2 \subseteq \mathtt{op}(N)$. *Then,* $\mathtt{noRB}(N_1, N_2, \mathtt{DG}_S)$*, if, and only if,*

$$\forall\, \mathsf{n}_1 \in N_1 \bullet (\nexists \mathsf{n}_2 \in N_2 \bullet \mathsf{n}_1 \xrightarrow{\bullet} \mathsf{n}_2 \wedge \nexists \mathsf{n'}_2 \in N_2 \bullet \mathsf{n}_1 \dashrightarrow \mathsf{n'}_2)$$

This condition will be called **no reaching back** from N_1 to N_2.

The definition will be instantiated as

$$\mathtt{noRB}(\mathbf{Ph}_2, \mathbf{C}_1, \mathtt{DG}_S) \wedge \mathtt{noRB}((\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \mathbf{C}_2, \mathtt{DG}_S).$$

Criterion 4: all-or-none

The last correctness criterion restricts the distribution of the set of operation nodes of the DG. Definition 2.3.4 introduced a labelling function l, mapping an operation node on its schema name. In our decomposition, we have to require that for any operation schema $op \in Op$, *all* corresponding nodes op^i are assigned to the same graph fragment.

Intuitively, this condition is necessary, since schemas corresponding to operation nodes occurring in the cut are generally modified. For the different cut sets $\mathbf{C}_1$ and $\mathbf{C}_2$, this modification can differ. Moreover, schemas occurring *outside* of the cut remain unchanged. A distribution of $\{\mathsf{op}^i \in \mathtt{op}(N) \mid l(\mathsf{op}^i) = op\}$ over at least two different segments would require a *duplication* of the schema which is undesirable and technically infeasible.

The following predicate defines this **all-or-none** law – it will subsequently be used with respect to the cut sets $\mathbf{C}_1, \mathbf{C}_2$ and the complement of $\mathbf{C}_1 \cup \mathbf{C}_2$:

Definition 4.2.7. *(**all-or-none**)*
Let $\mathtt{G} = (N, \longrightarrow)$ *be a graph, and let* $N_1, N_2 \subseteq N$. *Then,* $\mathtt{AoN}(N_1, N_2, \mathtt{G})$*, if, and only if,*

$$N_1 \subseteq N_2 \vee N_1 \subseteq (N \setminus N_2)$$

This condition will be called **all-or-none** law for N_1 relative to N_2.

This completes the definition of the correctness criteria. They will consecutively be used to define a *cut*, that is, a *correct* fragmentation of the DG, and subsequently the decomposition of a specification.

4.2.3 Definition of a Cut

The previously introduced correctness criteria along with Definition 4.2.2 immediately lead to the first of two core definitions of this thesis, the definition of a *cut*:[1]

Definition 4.2.8. *([General] Cut of the DG)*
Let $\mathtt{DG_S} = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph and* $\mathtt{CFG_S} = (N, \longrightarrow)$ *the control flow graph of a specification S, respectively. A fragmentation* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *of the CFG according to Definition 4.2.2 is called a (valid) cut of the DG, if, and only if, the following correctness criteria are satisfied:*

Criterion 1 (*disjointness*): *The following five sets are pairwise disjoint:*

- $\mathbf{Ph}_1$, $\mathbf{Ph}_2$, $\mathbf{Ph}_3$, *(phases)*
- $\mathbf{C}_1$, $\mathbf{C}_2$. *(cut sets)*

Criterion 2 (*no crossing*):

$\mathtt{noCr}((\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \mathbf{Ph}_2, \mathtt{DDG_S})$, *(**no crossing** between different components)*

[1] In the following definition, allowing a cut set to be empty does not pose a problem: if $\mathbf{C}_1 = \varnothing$, the fragmentation either yields a trivial decomposition or a contradiction to the criterion **disjointness**. $\mathbf{C}_2 = \varnothing$ will subsequently be identified as a special case of the cut definition.

Criterion 3 (*no reaching back*):

- $\texttt{noRB}(\mathbf{Ph}_2, \mathbf{C}_1, \texttt{DG}_S)$ *and* *(***no reaching back** *to* first *cut set)*
- $\texttt{noRB}((\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \mathbf{C}_2, \texttt{DG}_S)$, *(***no reaching back** *to* second *cut set)*

Criterion 4 (*all-or-none*): *For all operation nodes $op \in Op$:*

- $\texttt{AoN}(l^{-1}[\{op\}], \mathbf{C}_1, \texttt{DG}_S)$ *and* *(no cut-distribution of nodes*
- $\texttt{AoN}(l^{-1}[\{op\}], \mathbf{C}_2, \texttt{DG}_S)$. *associated to one operation)*

We ultimately aim at the definition of two specification parts S_1 and S_2, resulting from the decomposition of the dependence graph of S. The previous definition of a cut identifies a fragmentation of the set of operation nodes of the dependence graph in the following way: the unification of $\mathbf{C}_1$ and $\mathbf{C}_2$ together with $\mathbf{Ph}_3$ and $\mathbf{Ph}_1$ yields the set of operations of the first component S_1. Accordingly, $\mathbf{C}_1$ and $\mathbf{C}_2$ together with $\mathbf{Ph}_2$ constitute the second component S_2. This is illustrated in Figure 4.9. Operations corresponding to the first cut set identify the link from S_1 to S_2, whereas the second cut set determines the opposite link. The precise definition of S_1 and S_2 will be given in Section 4.3.

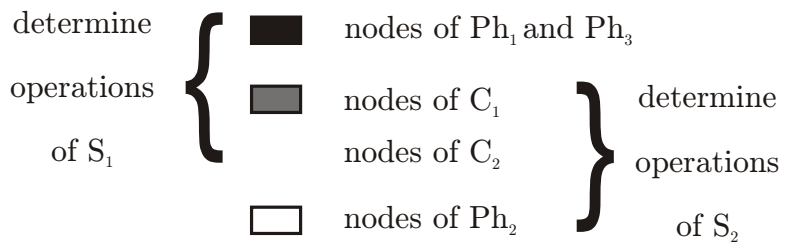


Figure 4.9: Fragmentation of the set of operation nodes in general case

In order to establish a well-defined fragmentation of the original dependence graph and thus well-defined specification components, CSP operator nodes need to be considered as well. In Section 4.3, we will determine the CSP parts of the components S_1 and S_2, resulting from a *projection* of the CSP part of S onto the specific sets of operation schemas. This definition will provide a correct distribution of the CSP operators and thus, operator nodes of the DG.

Figure 4.10 shows all allowed edges of the DG. Dotted edges depict data dependences, whereas solid edges represent a unification of both, control flow edges and data dependences.

As we introduced $\mathbf{C}_1$ as the first line of intersection and $\mathbf{C}_2$ as the second, we need to substantiate that $\mathbf{C}_2$ is located *behind* $\mathbf{C}_1$. The following lemma shows that our definition indeed matches with the intuition. It states that there are no direct CFG paths from the second cut to the first cut – any such path needs to proceed over $\mathbf{Ph}_1$ via a recursive call. Recall that

$$\mathsf{start}_1 := (\{\mathsf{start}\} \xrightarrow{\texttt{to}} \mathbf{C}_1) \cap \mathsf{StartNodes}.$$

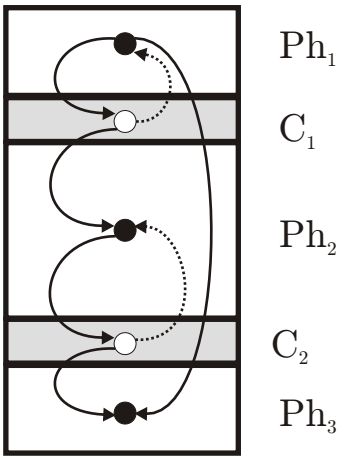


Figure 4.10: Assignment of DG edges to the subgraphs

Lemma 4.2.9. *(No direct CFG paths from second to first cut)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the DG of a specification S and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut of the DG. Then, the following holds:*

$$\forall\, c_1 \in \mathbf{C}_1, c_2 \in \mathbf{C}_2 \bullet (\pi \in \mathtt{path}_{\mathtt{CFG}}(c_2, c_1) \Rightarrow \pi \cap \mathsf{start}_1 \neq \varnothing).$$

Proof. Assume the opposite: let $\pi \in \mathtt{path}_{\mathtt{CFG}}(c_2, c_1)$ with $c_2 \xrightarrow{\pi} c_1$ and $\pi \cap \mathsf{start}_1 = \varnothing$. In this case, by definition of $\mathbf{Ph}_3$, the node $c_1 \in (\mathbf{Ph}_3 \cap \mathbf{C}_1)$ violates Definition 4.2.8, correctness criterion **disjointness**. □

Since the CFG of a specification may include recursive calls, yielding paths from $\mathbf{Ph}_2$ back to $\mathbf{Ph}_1$, we generally need to identify *two* lines of intersection. The first subgraph thus contains nodes located *before the first cut* ($\mathbf{Ph}_1$) as well as nodes located *behind the second cut* ($\mathbf{Ph}_3$). We will now additionally consider a special case of the segmentation, which corresponds to the definitions of [MWW08].

Assume that the dependence graph of a specification can be fragmented in such a way that there are no paths from $\mathbf{Ph}_2$ back to $\mathbf{Ph}_1$. Intuitively, this means that recursion can only occur within the same phase. In particular, such a DG does not incorporate "outer" recursive calls in the sense that a path reaching $\mathbf{Ph}_2$ never returns to the start-node.

In this case, the dependence graph can reasonably be segmented into two subgraphs without the need for a second line of intersection: the first subgraph contains the nodes before the sole line of intersection and the second subgraph the nodes behind it, whereas both subgraphs include the cut set.

In this specific case, we call the dependence graph *sequential* based on the possibility to fragment it without outer recursion. The now simplified fragmentation is illustrated in Figure 4.11. This leads to the following definition:

Definition 4.2.10. *(Single Cut)*
Let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. We call* $\mathbf{C}$ *a* single *cut, if, and only if,* $\mathbf{C}_2 = \varnothing$.

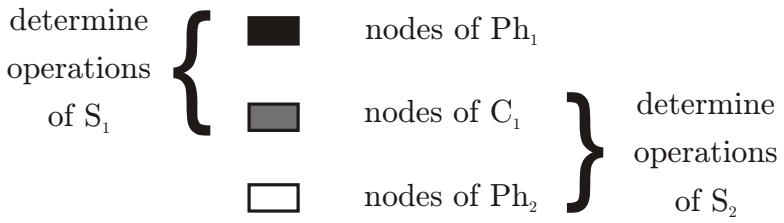


Figure 4.11: Fragmentation of the set of operation nodes in the special case

In the case of a single cut, we synonymously write $\mathbf{C}$ and $\mathbf{C}_1$. The restriction $\mathbf{C}_2 = \varnothing$ incorporates several repercussions. First of all, the fragmentation yields $\mathbf{Ph}_2 = (\mathbf{C}_1 \overset{\mathtt{to}}{\rightarrow} \varnothing) \cap \mathtt{op}(N)$ from which we can deduce that no CFG paths from $\mathbf{Ph}_2$ back to $\mathbf{C}_1$ are allowed *at all*. Moreover, no paths from $\mathbf{Ph}_2$ to $\mathbf{Ph}_1$ can exist. Finally, $\mathbf{Ph}_3 = \varnothing$ holds. We will summarise and proof these claims in the following lemma:

Lemma 4.2.11. *(Properties of single cut)*
Let $\mathbf{C}$ *be a single cut. Then, the following holds:*

1. $\mathbf{Ph}_2 = (\mathbf{C}_1 \xrightarrow{\mathtt{to}} \varnothing) \cap \mathtt{op}(N)$,
2. $\forall\, \mathsf{n} \in \mathbf{Ph}_2, \mathsf{n'} \in \mathbf{C}_1 \bullet \mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'}) = \varnothing$,
3. $\forall\, \mathsf{n} \in \mathbf{Ph}_2, \mathsf{n'} \in \mathbf{Ph}_1 \bullet \mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'}) = \varnothing$,
4. $\mathbf{Ph}_3 = \varnothing$,

Proof:

1. Obvious. ✓
2. Assume that there exist $\mathsf{n} \in \mathbf{Ph}_2, \mathsf{n'} \in \mathbf{C}_1$ such that $\pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'})$. We distinguish two cases for π: if $\pi \cap \mathbf{Ph}_1 = \varnothing$, there exist some nodes $\mathsf{l} \in \mathbf{Ph}_2, \mathsf{m} \in \mathbf{C}_1$ of π such that $\mathsf{l} \xrightarrow{\bullet} \mathsf{m}$. This yields a contradiction to the correctness criterion **no reaching back**. Otherwise, let m be the *first* node of π which is an element of $\mathbf{Ph}_1$. Then, π either reaches m via some direct edge from $\mathbf{Ph}_2$, violating the correctness criterion **disjointness** ($\mathsf{m} \in \mathbf{Ph}_2 \cap \mathbf{Ph}_1$). Otherwise, there is an indirect connection via $\mathbf{C}_1$, which again violates **no reaching back** at some point within π. ✓
3. Now assume there exist some $\mathsf{n} \in \mathbf{Ph}_2, \mathsf{n'} \in \mathbf{Ph}_1$ such that $\pi \in \mathtt{path}_{\mathtt{CFG}}(\mathsf{n}, \mathsf{n'})$. According to the previous case, second part, this path violates one of the correctness criteria **disjointness** and **no reaching back**. ✓
4. Since, in particular, $\varnothing \overset{\mathtt{to}}{\rightarrow} M = \varnothing$ for any set M, we immediately deduce the equation. ✓ □

Table 4.1 summarises the differences between a general cut and a single cut. Based on our case study from Chapter 2, we consecutively illustrate the definition for the special case of a single cut.

	General Cut	**Single Cut**
Number of Cut Sets	two	one
disjointness	$\mathbf{Ph}_1, \mathbf{C}_1, \mathbf{Ph}_2, \mathbf{C}_2, \mathbf{Ph}_3$ are pairwise disjoint	$\mathbf{Ph}_1, \mathbf{C}_1, \mathbf{Ph}_2$ are pairwise disjoint
First Subgraph	comprises $\mathbf{Ph}_1$, $\mathbf{C}_1$, $\mathbf{C}_2$ and $\mathbf{Ph}_3$	comprises $\mathbf{Ph}_1$ and $\mathbf{C}_1$
Second Subgraph	comprises $\mathbf{C}_1$, $\mathbf{Ph}_2$ and $\mathbf{C}_2$	comprises $\mathbf{C}_1$ and $\mathbf{Ph}_2$
Allowed Recursion	within one segment, between $\mathbf{Ph}_3$ and $\mathbf{Ph}_1$	within one segment

Table 4.1: Comparison between the general cut and the single cut

4.2.4 Candy Machine Revisited: Cut of the Dependence Graph

Chapter 2 introduced the specification *CandyMachine*. We illustrate the previous definitions of a fragmentation and a cut by means of this particular example. The example complies to the general restrictions for a *single* cut and thus allows a demonstration of the special case. Section 4.4 additionally illustrates the general case.

Here, we will neglect three specific data dependences, namely the three *initial* data dependences originating from the `Init` predicate $items = \langle\rangle$ to the respective operation nodes order, term and deliver. The reason why we are allowed to do this will precisely be given in Section 4.3.7, where we will deal with the neglect of specific initial data dependences. Intuitively, these dependences originate from a predicate restricting a variable which is never modified or referenced in any of the schemas *pay*, *payout*, *abort* and *switch*. We will show that the source of this dependence can safely be moved to the second subgraph.

We start the illustration of the cut definition with the fragmentation of the CFG, according to Definition 4.2.2. Figure 2.9 from Section 2.3.2 depicts the control flow graph of the candy machine. Let $\mathbf{C}_1 := \{\mathsf{switch}\}$ and, according to the definition of a single cut, $\mathbf{C}_2 := \varnothing$. This leads to the following fragmentation:

$$\begin{aligned} \mathbf{Ph}_1 &= \{\mathsf{pay}, \mathsf{payout}, \mathsf{abort}\}, \\ \mathbf{C}_1 &= \{\mathsf{switch}\} \text{ and} \\ \mathbf{Ph}_2 &= \{\mathsf{select}, \mathsf{order}, \mathsf{term}, \mathsf{deliver}\}. \end{aligned}$$

For showing that this fragmentation satisfies the constraints of Definition 4.2.8, recall

the DG of the candy machine specification as given in Figure 2.13, and consider the four correctness criteria for the decomposition:

Criterion 1 (disjointness): $\mathbf{Ph}_1$, $\mathbf{C}_1$ and $\mathbf{Ph}_2$ are disjoint. In particular, this is due to the non-existent recursive calls from $\mathbf{Ph}_2$ to $\mathbf{Ph}_1$.

Criterion 2 (no crossing): In case we neglect the previously identified initial data dependences, $\mathtt{noCr}(\mathbf{Ph}_1, \mathbf{Ph}_2, \mathtt{DDG_S})$ holds. No data dependences connect a node of $\mathbf{Ph}_1$ and $\mathbf{Ph}_2$.

Criterion 3 (no reaching back): $\mathtt{noRB}(\mathbf{Ph}_2, \mathbf{C}_1, \mathtt{DG_S})$ holds as well. There are no CFG paths or data dependences originating from $\mathbf{Ph}_2$ targeting $\{\mathsf{switch}\}$.

Criterion 4 (all-or-none): Obvious, since there are no multiple occurrences of an operation within the CSP part of *CandyMachine*.

The fragmentation based on $\mathbf{C}_1 = \{\mathsf{switch}\}$ thus yields a valid (single) cut. This is illustrated in Figure 4.12. The left hand side depicts the first subgraph, and the right hand side displays the second subgraph. Note that for the reduced parts of the graphs, we applied a simplification on the sets of CSP operators and control flow edges. The precise definition of the modified CSP part is given in the next section.

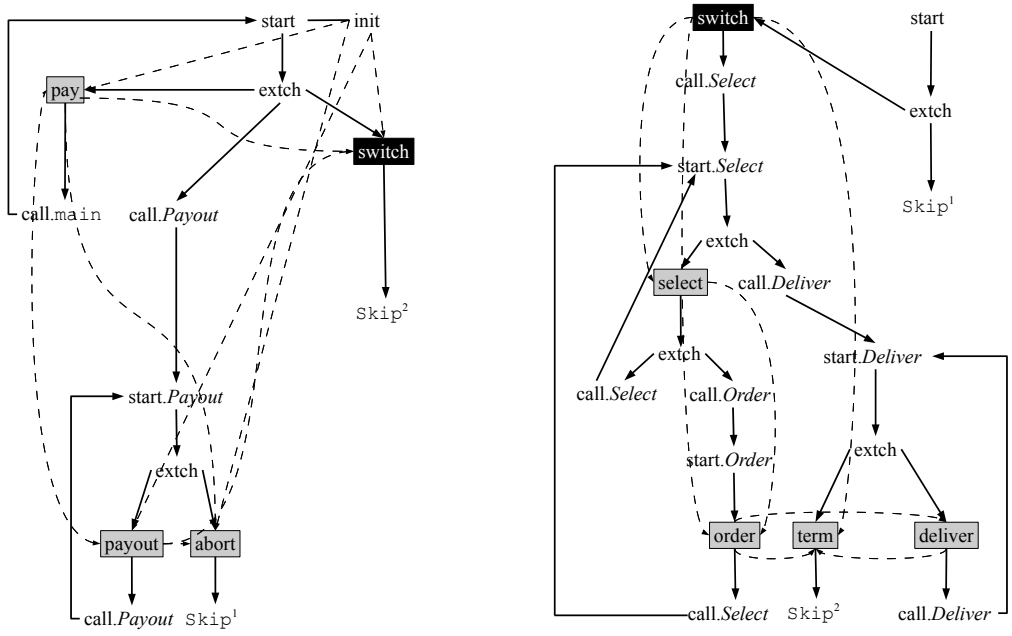


Figure 4.12: Cut of the dependence graph for the candy machine

This concludes the illustration of a (single) cut. So far, we considered the dependence graph of a specification which represents its dependence structure. We defined the cut of

the DG, separating it into two parts. Next, we need to transfer the fragmentation of the *graph* $\mathtt{DG}_S$ to the decomposition of the *specification* S.

4.3 Decomposing CSP-OZ Specifications

A *cut* of the dependence graph of a specification S as defined in the previous section determines a fragmentation of the DG, resulting in several clusters of nodes. This segmentation serves as the cornerstone for the identification of two specifications S_1 and S_2, representing a correspondent decomposition of S.

S	
I	[interface definition]
`main`	[CSP part]
`State`	[Object-Z part: state schema]
`Init`	[Object-Z part: initial state schema]
`enable_`*op*	[Object-Z part: `enable`-schemas]
`effect_`*op*	[Object-Z part: `effect`-schemas]

Figure 4.13: Constituents of a CSP-OZ class specification

In this section, we transfer the previous definitions from the graph level to the specification level. Again, we do not distinguish between specifications consisting of one and several classes. The decomposition of a specification is defined with respect to the fragmentation of the DG and is thus independent of the class structure. Therefore, throughout this thesis, we will synonymously refer to *class* and *specification*.

Recall the structure of a CSP-OZ class specification as given in Figure 4.13. At first, we have to identify the different constituents of S_1 and S_2, namely its interface definition, its CSP part and its Object-Z part. Subsequently, we assemble both specifications by identifying a synchronisation alphabet A, employed for the definition of $S_1 \parallel_A S_2$. The construction has to make sure that S and $S_1 \parallel_A S_2$ have the same observable behaviour.

A first fingerpost for the definition of S_1 and S_2 is directly given by the fragmentation of the DG: the sets of operation nodes corresponding to $\mathbf{C}_1$ and $\mathbf{C}_2$ take the role of connecting the different specification parts where $\mathbf{C}_1$ is responsible for preserving the influence of S_1 on S_2 whereas $\mathbf{C}_2$ identifies the opposite link. Additionally, the cut is the basis for the definition of the synchronisation alphabet A. Moreover, nodes of $\mathbf{Ph}_1, \mathbf{Ph}_2$ and $\mathbf{Ph}_3$ represent the operations local to S_1 ($\mathbf{Ph}_1, \mathbf{Ph}_3$) and S_2 ($\mathbf{Ph}_2$). This is illustrated in Figure 4.14.

In order to construct two well-defined specifications S_i, $i \in \{1, 2\}$, we start with a first, intermediate definition of a decomposition in Section 4.3.1, where we need to deal with the following subtasks:

Definition of the Interfaces of S_i: Identifying the set of operations of a component, the

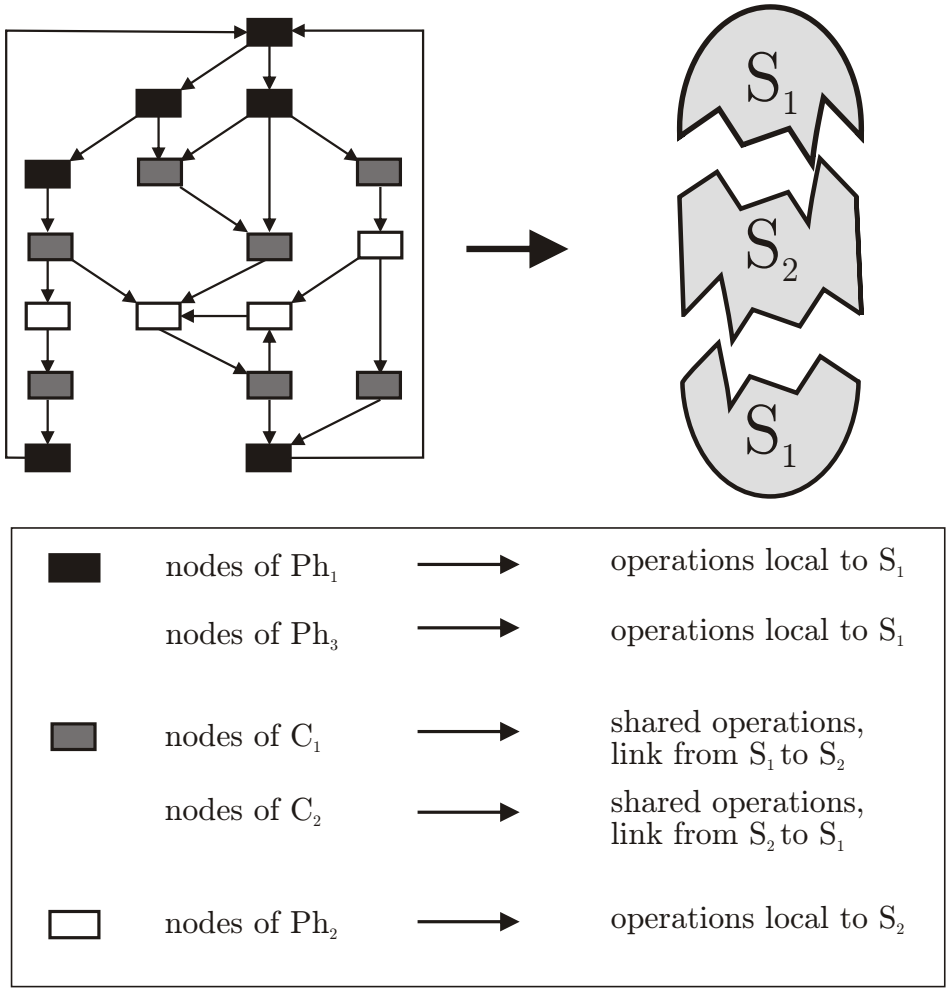


Figure 4.14: Correspondence between graph nodes and specification operations

fragmentation of the dependence graph immediately yields the set of channel declarations of S_i.

Definition of the CSP Parts S_i.`main`: According to its interface, the CSP part of S_i needs to be restricted to the component's set of channels. For that purpose, we define a *projection* of the original CSP part on the remaining operations of a component, according to [Brü08].

Definition of the State Schemas of S_i: One of the decisive aspects for an effective application of compositional reasoning is the size of the *state space* of the involved components. As the set V of state variables of a specification's Object-Z part determines the size of the Object-Z state space, the sets $S_1.V$ and $S_2.V$ necessarily need

to be smaller than $S.V$. Hence, we need to identify two subsets of $S.V$, forming the sets of required state variables for the specification parts. Additionally, we need to deal with the state invariants of the state schema.

Definition of the Initial State Schemas S_i.Init: Following up on the restriction of the sets of state variables, we accordingly need to restrict the original initial state schema. Moreover, an optimisation for this definition, as already indicated in the last section, will be given in Section 4.3.7.

Definition of the Operation Schemas for S_i: According to the definition of the set of channels, we use the fragmentation of the dependence graph in order to identify the sets of operation schemas of a component. The determination of their respective declaration parts and predicate parts is straightforward.

Definition of the Synchronisation Alphabet: The definition of both specification parts leads to the overall system $S_1 \parallel_A S_2$. The assembly requires a definition of the synchronisation alphabet A.

Carrying out the previous considerations will result in two well-defined specifications S_1 and S_2 and an assembly of S_1 and S_2 into $S_1 \parallel S_2$. However, the pure definition of two specification parts, resulting from a cut, is insufficient. Additionally, we need to preserve the behaviour of the specification. To this end, we have to *modify* part of the generated components, mainly by adding *parameters* to some operations:

Preservation of Data Dependences: Even though we do not allow data dependences to circumvent the cut based on the correctness criterion **no crossing**, we still have to transmit the *allowed* influence of one on the other specification part. Data dependences may indeed target the set of cut operation nodes as well as originate from them. From a specification level, this means that modifications of state variables within one component influence state variables of the other component. In order to preserve these dependences, we introduce additional *transmission* parameters, passing the relevant state variable modifications of one to the other component. Section 4.3.2 deals with this aspect.

Preservation of CSP Part: The definition of the CSP parts for S_1 and S_2 based on a projection does not automatically yield an equivalence of the original CSP part and the CSP part of $S_1 \parallel S_2$. In particular, the synchronisation of both CSP processes may introduce additional sequences of events which are infeasible for the original specification. For ensuring the equivalence of both, the CSP parts of S and $S_1 \parallel S_2$, we introduce additional *address* parameters, ensuring a correct synchronisation of both resulting CSP processes in Section 4.3.3.

Renaming of Events: Based on the introduction of additional parameters to some of the specification's channels, S and $S_1 \parallel S_2$ are solely equivalent modulo different channel types. In Section 4.3.4, we introduce an *event renaming* relation, linking the modified to the original channels.

These are the crucial aspects which we will deal with in the upcoming sections. We proceed in two steps: first, in Section 4.3.1, we introduce a decomposition of S with respect to a cut into two well-defined specification parts S_1 and S_2. Subsequently, we modify the decomposition to achieve a *thorough* decomposition by modifying part of the components elements. The complete definition of the thorough decomposition of S into S_1 and S_2 will be given in Section 4.3.5, incorporating all the definitions and considerations of the previous sections. After illustrating the approach on our candy machine specification in Section 4.3.6, Section 4.3.7 gives an improvement for the decomposition by pointing out an optimisation for dealing with initial state predicates.

4.3.1 Intermediate Definition of the Decomposition

The current section stepwise introduces the different constituents of two specifications S_1 and S_2, resulting from a valid cut of $\mathtt{DG}_S$. As of now, we are interested in developing a *well-defined* decomposition. Some of the subsequent definitions are marked as *intermediate*, as the corresponding specification elements will later by modified to ensure a *semantics-preserving* decomposition.

We start the definition of the components S_1 and S_2 by identifying their respective interfaces and CSP parts. In order to bridge the gap between the set of operation nodes, resulting from a cut and the corresponding set of operations, we use Definition 2.3.4:

Definition 4.3.1. *(Sets of operations of components)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. The sets of operation schemas for the decomposition of S are defined as*

- $Op_1 := l[(\mathbf{Ph}_1 \cup \mathbf{Ph}_3) \setminus \{\mathsf{init}\}]$,
- $Op_2 := l[\mathbf{Ph}_2]$,
- $Op_{\mathbf{C}_1} := l[\mathbf{C}_1]$ *and*
- $Op_{\mathbf{C}_2} := l[\mathbf{C}_2]$.

We let $Op_{\mathbf{C}} := Op_{\mathbf{C}_1} \cup Op_{\mathbf{C}_2}$.

We exclude init from the definition since we will separately deal with the initial state schema. It is important to note that in general, Op_1 and Op_2 are *not* disjoint, as a multiple occurrence of an operation may lead to one occurrence being assigned to $\mathbf{Ph}_1 \cup \mathbf{Ph}_3$ and another to $\mathbf{Ph}_2$. However, the three sets $Op_{\mathbf{C}_1}$, $Op_{\mathbf{C}_2}$ and $Op_1 \cup Op_2$ are indeed disjoint based on the correctness criterion **all-or-none**.

Next, we deduce the interfaces of the components S_1 and S_2 from the previous definition, where $I|_O$ denotes the restriction of the interface I on the operations of O:

Definition 4.3.2. *(Interfaces of components, intermediate definition)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. The interfaces for the decomposition of S into* S_1 *and* S_2 *are defined as*

- $S_1.I := I|_{(Op_1 \cup Op_{\mathbf{C}})}$ *and* *(Interface for* S_1*)*

- $S_2.I := I|_{(Op_2 \cup Op_{\mathbf{C}})}$. *(Interface for S_2)*

This definition will slightly be adapted in Section 4.3.3, based on the introduction of additional parameters to the channels.

For determining the CSP parts of S_i, the process $S.\texttt{main}$ is restricted on the sets of events corresponding to the component's sets of operations. To this end, we define the *projection* of a CSP process on a subset of its events according to [Brü08]. The definition also applies, if the specification is composed of several classes:

Definition 4.3.3. *(Projection of CSP processes, [Brü08])*
Let P be the right-hand side of a CSP process definition and $E \subseteq$ Events. The projection of P on E, *denoted by $P|_E$, is inductively defined:*

1. $\texttt{Skip}|_E := \texttt{Skip}$ *and* $\texttt{Stop}|_E := \texttt{Stop}$,
2. $(e \rightarrow P)|_E := \begin{cases} P|_E, & e \notin E \\ e \rightarrow P|_E, & \textit{otherwise}, \end{cases}$
3. $(P \circ Q)|_E := (P|_E) \circ (Q|_E)$ *for* $\circ \in \{;\ , |||, \Box, \sqcap\}$,
4. $(P\ ||_A\ Q)|_E := (P|_E)\ ||_{A \cap E}\ (Q|_E)$.

According to [Brü08], we can apply several simplifications to the resulting CSP processes. Such a modification is, for instance, given by replacing a process equation $P \stackrel{c}{=} P$ by $P \stackrel{c}{=} \texttt{Stop}$ or $P \stackrel{c}{=} (P \circ Q)$ with $P \stackrel{c}{=} Q$ for $\circ \in \{\Box, \sqcap\}$. Note that an equation $P \stackrel{c}{=} P$ introduces *divergence* [Ros98] into the overall process, that is, an infinite loop without an execution of an external event. In the semantic model of traces, replacing it with $P \stackrel{c}{=} \texttt{Stop}$ does not influence the behaviour of the process. For more details, see [Brü08].

This definition of the projection allows us to inductively define the processes $S_1.\texttt{main}$ and $S_2.\texttt{main}$. As the definition is applied with respect to a set of *events*, we use the extension sets of the respective sets of operations.

Definition 4.3.4. *(CSP parts of components, intermediate definition)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ be the dependence graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. The CSP parts for the decomposition of S into S_1 and S_2 are defined as

- $S_1.\texttt{main} := S.\texttt{main}|_{\{|Op_1|\} \cup \{|Op_{\mathbf{C}}|\}}$ *and* *(CSP part for S_1)*
- $S_2.\texttt{main} := S.\texttt{main}|_{\{|Op_2|\} \cup \{|Op_{\mathbf{C}}|\}}$. *(CSP part for S_2)*

Again, due to the additional parameters, the CSP parts of the components will slightly be modified in Section 4.3.4 by introducing a renaming function.

Next, we define the Object-Z parts of S_1 and S_2. We have to identify their state schemas, initial state schemas and operation schemas.

The state schema of S comprises a set of state variables $S.V$ with their respective types, along with a possibly empty set of state invariants. In order to define the state schemas of S_1 and S_2, we first identify two subsets of $S.V$. By setting

- $S_1.V := all(Op_1 \cup Op_{\mathbf{C}_1})$ and
- $S_2.V := all(Op_2 \cup Op_{\mathbf{C}_2})$,

we restrict both state schemas to those variables which are referenced or modified in at least one of the component's local operations or operations of one specific cut set. Not adding $Op_{\mathbf{C}_1}$ and $Op_{\mathbf{C}_2}$ to *both* sets will become clearer when we define the predicate parts of the operations and when we deal with transmitting the state space modification between the components in Section 4.3.2. Note that we do not additionally refer to variables occurring in $S.\texttt{Init}$.

As a consequence of invariants influencing the execution of any operation, according to the previous definition, variables occurring in some invariant need to be represented in both, $S_1.V$ and $S_2.V$. This is implicitly guaranteed by the normalisation as introduced in Section 2.3.3, attaching all state invariants to any `effect`-schema.

For the complete definition of $S_i.\texttt{State}$, we will use Definition 2.2.1:

Definition 4.3.5. *(`State` schemas of components)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ be the dependence graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. The state schemas for the decomposition of S into S_1 and S_2 are defined over

- $S_1.V := all(Op_1 \cup Op_{\mathbf{C}_1})$ *and*
- $S_2.V := all(Op_2 \cup Op_{\mathbf{C}_2})$,

as

- $S_1.\texttt{State} := \{s \upharpoonright (S_1.V) \mid s \in S.\texttt{State}\}$ *and*
- $S_2.\texttt{State} := \{s \upharpoonright (S_2.V) \mid s \in S.\texttt{State}\}$.

Next, we are concerned with the initial state schema of a class, that is, the decomposition of $S.\texttt{Init}$ into $S_1.\texttt{Init}$ and $S_2.\texttt{Init}$. The question arises of how to deal with predicates referring to elements of both, $S_1.V$ and $S_2.V$.

Consider some initial state predicate $p(x,y)$ with x being assigned to $S_1.V \setminus S_2.V$ and y being assigned to $S_2.V \setminus S_1.V$. The predicate can neither be assigned to $S_1.\texttt{Init}$ nor to $S_2.\texttt{Init}$, since one of the specific variables is not an element of the respective component. However, an elimination of the predicate is infeasible, since the relation between x and y would get lost.

Therefore, a simple restriction of $S.\texttt{Init}$ onto predicates dealing with $S_i.V$ is insufficient. The general definition of the initial state schemas of S_1 and S_2 will refer to $S.\texttt{Init}$ and use an *existential quantification* for a subset of $S.V$. This leads to the following definition:

Definition 4.3.6. *(`Init` schemas of components)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ be the dependence graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. Furthermore, let $(S.V \setminus S_1.V) = \{v_1, \ldots, v_n\}$ and let $S_1.V = \{w_1, \ldots, w_m\}$. The initial state schemas for the decomposition of S into S_1 and S_2 are defined as

- $S_1.\texttt{Init} := \exists v_1, \ldots, v_n \bullet S.\texttt{Init}$ *and*
- $S_2.\texttt{Init} := \exists w_1, \ldots, w_n \bullet S.\texttt{Init}$.

Both `Init`-predicates are well-defined, that is, all free variables occurring in $S_i.\texttt{Init}$ are elements of its respective sets of state variables $S_i.V$. Note that for the initial state schema of S_2, shared variables, that is, elements of $S_1.V \cap S_2.V$, are also quantified: these variables are already restricted in the first specification part.

We use the following abbreviation: variables not occurring in the initial state schema will not be quantified. Precisely, if p is a predicate referring to variables $x_1, \ldots, x_k$,

$$\exists y_1, \ldots, y_m \bullet p(x_1, \ldots, x_k)$$

is abbreviated by

$$\exists z_1, \ldots, z_n \bullet p(x_1, \ldots, x_k),$$

where $\{z_1, \ldots, z_n\} = \{x_1, \ldots, x_k\} \cap \{y_1, \ldots, y_m\}$. Moreover, we omit trivially satisfied predicates as, for instance, $\exists v \bullet v = n$ with $n \in t_v$.

Recall the abstract example from before: the initial state predicate $p(x,y)$ will be changed to $\exists y \bullet p(x,y)$ for $S_1.\texttt{Init}$ and to $\exists x \bullet p(x,y)$ for $S_2.\texttt{Init}$. A proof of the adequateness of this definition will be given in Chapter 5. In addition, Section 4.3.7 indicates that a subset of a specification's initial data dependences does not need to be considered when it comes to validating the correctness of a cut.

We remain to define the declaration parts and predicate parts of the component's operations. For local operations to S_i, we simply keep the original definition *as-is*. For the set of cut operations, we solely keep the predicate parts in one of the specifications parts. In order to ensure corresponding types, we always need to preserve the original declaration parts. Precisely:

Definition 4.3.7. *(Operation schemas of components, intermediate definition)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ *be the dependence graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. The operation schemas for the decomposition of S into* S_1 *and* S_2 *are defined as*

$$S_1.op := \begin{cases} S.op, & op \in (Op_1 \cup Op_{\mathbf{C}_1}), \\ [S.op.dec \mid true], & op \in Op_{\mathbf{C}_2}. \end{cases}$$

$$S_2.op := \begin{cases} S.op, & op \in (Op_2 \cup Op_{\mathbf{C}_2}), \\ [S.op.dec \mid true], & op \in Op_{\mathbf{C}_1}. \end{cases}$$

Again, this definition needs to be modified, when we are dealing with data dependences between both components. Finally, we unify all the previous definitions into one, the *intermediate* decomposition of S into two components S_1 and S_2:

Definition 4.3.8. *(Decomposition with respect to a cut, intermediate definition)* *Let* $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. Let*

$$Op_1, Op_2, Op_{\mathbf{C}_1}, Op_{\mathbf{C}_2}, Op_{\mathbf{C}}$$

be defined according to Definition 4.3.1. The (intermediate) decomposition of S with respect to $(\mathbf{C}_1, \mathbf{C}_2)$ into S_1 and S_2 *is defined as*

S_1	
$S_1.I$	*[*according to Definition 4.3.2*]*
$S_1.\mathtt{main}$	*[*according to Definition 4.3.4*]*
$S_1.\mathtt{State}$	*[*according to Definition 4.3.5*]*
$S_1.\mathtt{Init}$	*[*according to Definition 4.3.6*]*
$S_1.op$	*[*according to Definition 4.3.7*]*

S_2	
$S_2.I$	*[*according to Definition 4.3.2*]*
$S_2.\mathtt{main}$	*[*according to Definition 4.3.4*]*
$S_2.\mathtt{State}$	*[*according to Definition 4.3.5*]*
$S_2.\mathtt{Init}$	*[*according to Definition 4.3.6*]*
$S_2.op$	*[*according to Definition 4.3.7*]*

The *system* generated from the components is defined as the parallel composition of both classes, synchronising on the set of cut events, that is,

$$S_1 \parallel_{\{|Op_{\mathbf{C}}|\}} S_2.$$

In Section 4.3.6, we carry out the decomposition for the candy machine. For a stepwise illustration of the decomposition on a simpler example, we consider a trivial CSP-OZ specification for subsequently increasing three natural numbers l, m and n as given in Figure 4.15. The set $\mathbf{C} = \{\mathsf{change\_m}\}$ defines a valid single cut. We get

- $Op_1 := \{change\_l\}$,
- $Op_2 := \{change\_n\}$ and
- $Op_{\mathbf{C}} := \{change\_m\}$.

The intermediate definition of the components $Increaser_1$ and $Increaser_2$ is given in Figure 4.16. The overall system is defined as

$$Increaser_1 \parallel_{\{|change\_m|\}} Increaser_2.$$

Note that currently, $Increaser_2.change\_m$ is empty. Moreover, the generated initial state predicates can be simplified:

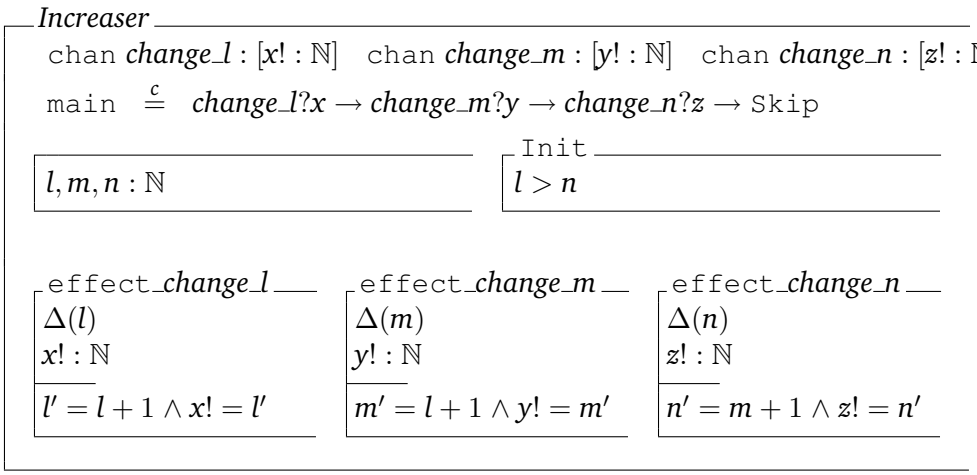


Figure 4.15: Simple CSP-OZ specification for increasing two natural numbers

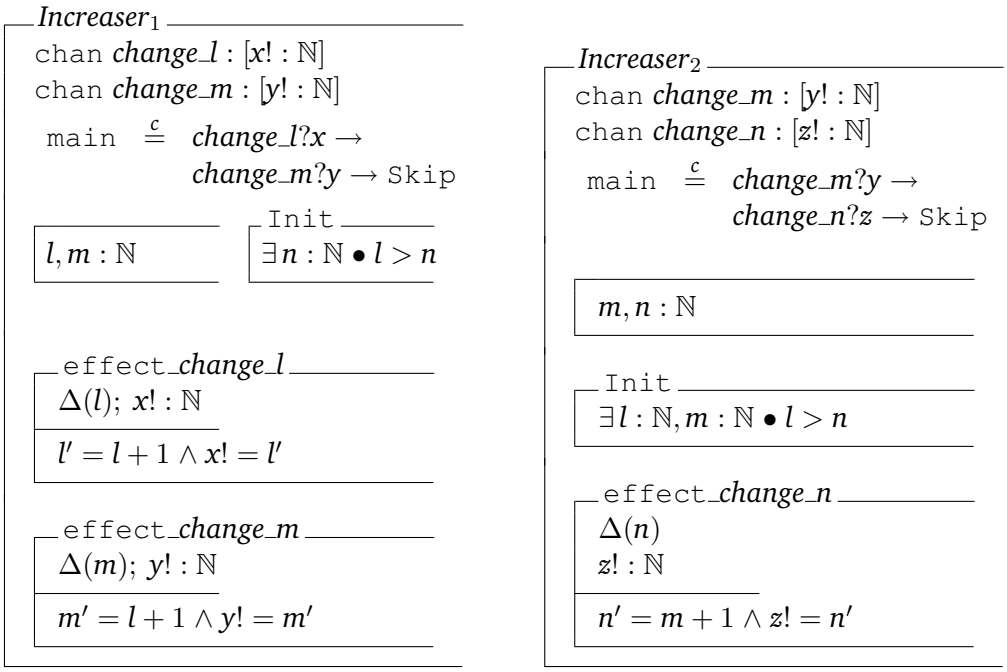


Figure 4.16: Intermediate decomposition of *Increaser*

- $\exists n : \mathbb{N} \bullet l > n \Leftrightarrow l > 0$ and
- $\exists l : \mathbb{N}, m : \mathbb{N} \bullet l > n \equiv$ true, respectively.

This completes the intermediate definition of the different constituents of the components, resulting in a well-defined decomposition of S. In an optimum way, the de-

composition results in two completely independent specification parts S_1 and S_2. In this case, the previously given intermediate decomposition is *final* in the sense that no further modification is required. In the context of assume-guarantee reasoning, this is preferable, as no supplemental constructs need to be added, ensuring that the size of the components remains rather small.

However, two completely independent specification parts are far from realistic. This would, for instance, require the cut to split a graph into two unrelated pieces, not sharing any ingoing and outgoing data dependences. Along with that, any branching within the control flow graph would have to be assigned to one component.

In order to ensure a universally *valid* decomposition in our context, the introduction of additional parameters and an event renaming is required. These extensions are given next, yielding a modification of the previously as intermediate marked definitions.

4.3.2 Preservation of the Data Dependences

As a first step, we are interested in preserving the original data flow, that is, the state space modifications. In particular, both components sharing the same state variables requires that a modification within one component is visible to the other component. Even though it is impossible that data dependences circumvent the set of cut operations based on the criterion **no crossing**, they can indeed target the cut and originate from it, thus causing mutual influence between both components, based on the data flow. Figure 4.17 again illustrates the fragmentation of a specification's dependence graph.

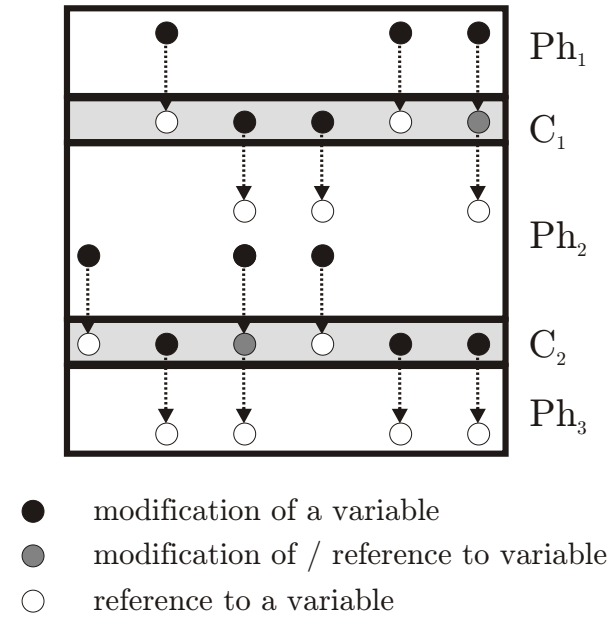


Figure 4.17: Possible data dependences targeting the cut and originating from the cut

Dotted edges denote data dependences between two operation nodes, where the schema corresponding to the source node *modifies* a certain state variable, and the target schema *references* a variable. Nodes highlighted in grey depict schemas which modify one and reference another state variable.

The crucial edges are the ones *originating* from a cut operation and targeting an operation in the subsequent phase or the other cut: they represent variables modified in one specification part (within a cut schema and possibly before as well) and referenced in the other. These modifications must be *preserved* to not refer to inconsistent values.

In the example *Increaser*, a particular sequence of two data dependences conforms to this specific problem: the schema *change_l* modifies the variable *l*, the schema *change_m* references l and modifies *m* and *change_n* references *m*. This sequence of state modifications is not reflected in the decomposition of *Increaser* as given in the last section. For an illustration, assume that initially, $l = 3$, $m = 2$ and $n = 1$ holds. Table 4.2 denotes the state valuations of *Increaser* during the processing of the event trace $\langle \mathit{change\_l}.4, \mathit{change\_m}.5, \mathit{change\_n}.6 \rangle$. Additionally, assuming the same initial state, the corresponding traces of the components are given.

Trace of *Increaser*	Trace of $\mathit{Increaser}_1$	Trace of $\mathit{Increaser}_2$
$\langle (l = 3, m = 2, n = 1),$	$\langle (l = 3, m = 2),$	
change_l.4,	*change_l*.4,	
$(l = 4, m = 2, n = 1),$	$(l = 4, m = 2),$	$\langle (m = 2, n = 1),$
change_m.5,	*change_m*.5,	*change_m*.**2**,
$(l = 4, m = 5, n = 1),$	$(l = 4, m = 5), \rangle$	$(\mathbf{m{=}2}, n = 1),$
change_n.6,		*change_n*.**3**,
$(l = 4, m = 5, n = 6) \rangle$		$(\mathbf{m} = \mathbf{2}, \mathbf{n} = \mathbf{3}) \rangle$

Table 4.2: Comparison of two traces for *Increaser* and its components

As the modification of *m* depends on l and is no longer represented in $\mathit{Increaser}_2$, the value of *m* is *inconsistent* after the operation *change_m* took place. This inconsistency is in particular visible to the outside, as the parameter value of the event *change_m* has changed from 5 to 2. Even worse, this inconsistency is propagated to the value of *n* as well. The inconsistency changes the behaviour of the original specification as the trace $\langle \mathit{change\_l}.4, \mathit{change\_m}.5, \mathit{change\_n}.6 \rangle$ cannot be restored within $\mathit{Increaser}_1 \parallel_{\{|\mathit{change\_m}|\}} \mathit{Increaser}_2$. Since we are interested in the equivalence of traces of events the specification and its decomposition may perform, this inconsistency must be prohibited.

The set of cut operations serves as the (sole) link between both specification parts, and any influence of one component on the other must be transmitted via the cut. A correspondence of the values of shared variables between both components is achieved by the introduction of additional parameters. The type of a cut operation is possibly extended based on this set of *transmission* parameters, each representing one specific shared state variable modified in one and referenced in the other specification part.

Precisely, these parameters are outputs to the *modifying* specification part and inputs to the *referencing* component, while transmitting the values of the respective state variables.

First, we have to clarify which variables actually need to be transmitted, that is, which variables exert influence from one on the other component. The following definition identifies two sets of state variables, namely the ones which need to be transmitted via the first cut set and the second cut set:

Definition 4.3.9. *(Cut variables)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. The modifications of* $\mathsf{n} \in \mathtt{op}(N)$ *influencing* $X \subseteq \mathtt{op}(N)$ *are defined as*

$$V_X^{\mathsf{n}} = \{v \in S.V \mid \exists\, \mathsf{n'} \in X \bullet \mathsf{n} \overset{\mathtt{dd}}{\dashrightarrow}_{(v)} \mathsf{n'} \vee \mathsf{n} \overset{\mathtt{ifdd}}{\dashrightarrow}_{(v)} \mathsf{n'}\}.$$

The sets of cut variables *for the decomposition of S into* S_1 *and* S_2 *are given by*

- $CV(\mathbf{C}_1) := \bigcup_{\mathsf{n} \in \mathbf{C}_1} V^{\mathsf{n}}_{(\mathbf{Ph}_2 \cup \mathbf{C}_2)}$ *and*
- $CV(\mathbf{C}_2) := \bigcup_{\mathsf{n} \in \mathbf{C}_2} V^{\mathsf{n}}_{(\mathbf{Ph}_1 \cup \mathbf{Ph}_3 \cup \mathbf{C}_1)}$.

$v \in CV(\mathbf{C}_1)$ holds, if there exists a (direct- or interference-) data dependence by reason of v originating from the first set of cut operations and targeting an operation from $\mathbf{Ph}_2$ or $\mathbf{C}_2$. $CV(\mathbf{C}_2)$ is analogously defined. The definition is complete in the sense that *all* variables exerting influence from one on the other component are included: the correctness criterion **no crossing** ensures that data dependences must not circumvent the set of cut nodes.

As we need to refer to operation *schemas* instead of operation *nodes* when adding transmission parameters to an operation, we set

$$V_X^{op} = \bigcup_{\mathsf{n} \in l^{-1}(op)} V_X^{\mathsf{n}}$$

and let

$$CV_1 := V^{op}_{(\mathbf{Ph}_2 \cup \mathbf{C}_2)} \text{ and } CV_2 := V^{op}_{(\mathbf{Ph}_1 \cup \mathbf{Ph}_3 \cup \mathbf{C}_1)}.$$

Even though we might have different sets of cut variables for $\mathsf{n}, \mathsf{n'} \in l^{-1}(op)$, the definition is reasonable: the correctness criterion **all-or-none** ensures that two different operation nodes corresponding to one operation schema must not be distributed over a cut set and its complement.

The previous considerations lead to the following, final definition for the operation schemas of S_1 and S_2:

Definition 4.3.10. *(Operation schemas of components, final definition)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the dependence graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. For* $CV_1 = \{v_1, \ldots, v_n\}$ *and* $CV_2 = \{w_1, \ldots, w_m\}$*, let*

$$\begin{array}{lcllcl} op.tr\_in^1 & = & tr_{v_1}? : t_{v_1};\ \ldots;\ tr_{v_n}? : t_{v_n}, & op.tr\_in^2 & = & tr_{w_1}? : t_{w_1};\ \ldots;\ tr_{w_m}? : t_{w_m}, \\ op.tr\_out^1 & = & tr_{v_1}! : t_{v_1};\ \ldots;\ tr_{v_n}! : t_{v_n}, & op.tr\_out^2 & = & tr_{w_1}! : t_{w_1};\ \ldots;\ tr_{w_m}! : t_{w_m}. \end{array}$$

The operation schemas for the decomposition of S into S_1 and S_2 are defined as

$$S_1.op := \begin{cases} S.op, & op \in Op_1, \\ [S.op.delta;\ S.op.dec;\ op.tr\_out^1 \mid op.pred \wedge \bigwedge_{v \in CV_1} tr_v! = v'], & op \in Op_{\mathbf{C}_1}, \\ [\Delta(w_1, \ldots, w_m);\ S.op.dec;\ op.tr\_in^2 \mid \bigwedge_{w \in CV_2} w' = tr_w?], & op \in Op_{\mathbf{C}_2}. \end{cases}$$

$$S_2.op := \begin{cases} S.op, & op \in Op_2, \\ [\Delta(v_1, \ldots, v_n);\ S.op.dec;\ op.tr\_in^1 \mid \bigwedge_{v \in CV_1} v' = tr_v?], & op \in Op_{\mathbf{C}_1}, \\ [S.op.delta;\ S.op.dec;\ op.tr\_out^2 \mid op.pred \wedge \bigwedge_{w \in CV_2} tr_w! = w'], & op \in Op_{\mathbf{C}_2}. \end{cases}$$

The declaration parts of all cut operations are extended by additional transmission parameters. For the influence of S_1 on S_2, we add predicates $tr_v! = v'$ for each cut variable $v \in CV_1$ to the first cut set and corresponding predicates $v' = tr_v?$ to the second. We proceed accordingly for variables of S_2 influencing S_1. The delta lists of the *receiving* operations need to comprise all modified cut variables. Figure 4.18 illustrates the concept of these parameters. In Chapter 5, we will show that this technique is sufficient to restore the data flow of a specification in its decomposition.

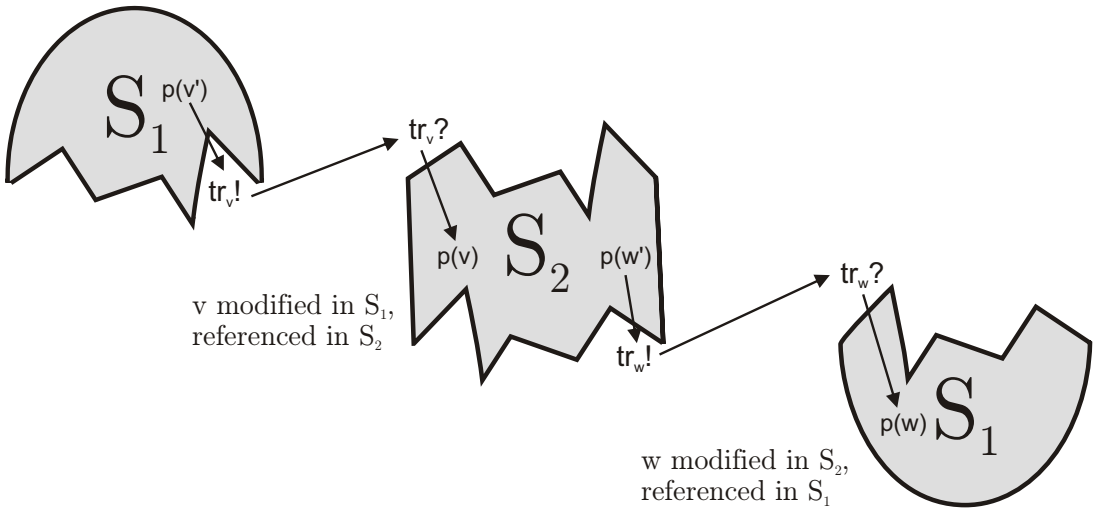


Figure 4.18: Illustration of the transmission parameters

In our example, due to the data dependence change_m $\overset{dd}{\dashrightarrow}_{(m)}$ change_n, the state variable m is a cut variable of *change_m*. Thus, we add one transmission parameter tr_m to *change_m*, serving as an output to $Increaser_1.change\_m$ and an input to

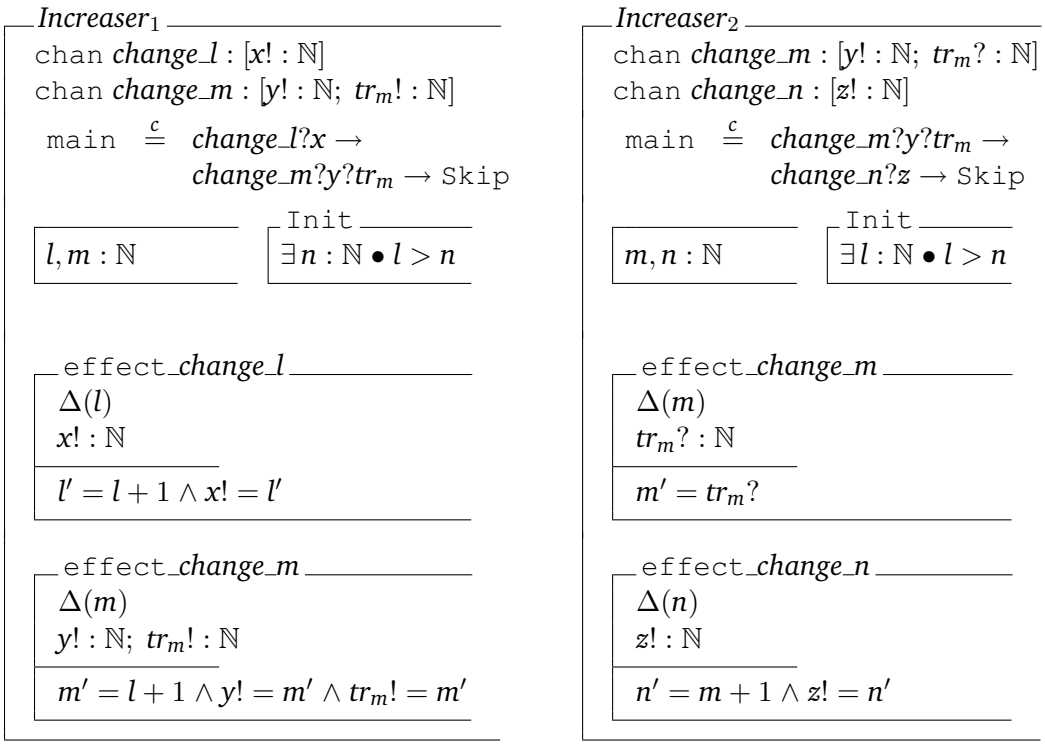


Figure 4.19: Decomposition of *Increaser*, modified according to Definition 4.3.10

Increaser$_2$.*change_m*. The modified decomposition is shown in Figure 4.19. Note that we have to modify the specification's interfaces and CSP parts as well. We deal with this aspect in Section 4.3.4.

This modification fixes the previously identified inconsistency as shown in Table 4.3. Next, we deal with the reconstitution of the *control flow* of the original specification within its decomposition. The underlying concept similarly uses additional parameters.

Trace of *Increaser*	Trace of *Increaser*$_1$	Trace of *Increaser*$_2$
$\langle(l=3,m=2,n=1),$ *change_l*.4, $(l=4,m=2,n=1),$ *change_m*.5, $(l=4,m=5,n=1),$ *change_n*.6, $(l=4,m=5,n=6)\rangle$	$\langle(l=3,m=2),$ *change_l*.4, $(l=4,m=2),$ *change_m*.5, $(l=4,m=5),\rangle$	$\langle(m=2,n=1),$ *change_m*.**5**, $(\mathbf{m{=}5},n=1),$ *change_n*.**6**, $(\mathbf{m=5},\mathbf{n=6})\rangle$

Table 4.3: Comparison of two traces of *Increaser* and its components after modification

4.3.3 Preservation of the Control Flow

The fact that one specification part influences the other one due to its *data* flow is quite intuitive. Additional to that and less obvious, the intermediate decomposition and reassembly can also cause a modification of the original *control* flow of a specification. For instance, it is possible that the CSP part of $S_1 \parallel S_2$ allows for additional traces, thus causing a violation of the trace equivalence between S and $S_1 \parallel S_2$.

As the problem of preserving the control flow of a specification is solely related to the CSP part of a specification, we entirely omit dealing with the Object-Z part in this section.

Restoring the Original Synchronisation

First, we will deal with ensuring a correct synchronisation between S_1 and S_2. In order to illustrate the general problem, we give a small example.

Example 4.3.11. *Let S be a specification over a set of events $\{a, b, c, d, e\}$, and let*

$$S.\texttt{main} := (a \rightarrow c \rightarrow d \rightarrow \texttt{Skip}) \,\Box\, (b \rightarrow c \rightarrow e \rightarrow \texttt{Skip}).$$

Let $\mathbf{C} = \{c\}$ be a valid single cut yielding

- $S_1.\texttt{main} := (a \rightarrow c \rightarrow \texttt{Skip}) \,\Box\, (b \rightarrow c \rightarrow \texttt{Skip})$ *and*
- $S_2.\texttt{main} := (c \rightarrow d \rightarrow \texttt{Skip}) \,\Box\, (c \rightarrow e \rightarrow \texttt{Skip})$.

Let $tr := \langle a, c, e \rangle$. Then, $tr \in traces(S_1.\texttt{main} \parallel_{\{c\}} S_2.\texttt{main})$ but $tr \notin traces(S.\texttt{main})$.

The example points out the following: Definition 4.2.8 allows the cut sets to contain several nodes with the same operation name - for $\mathsf{n}_1, \mathsf{n}_2 \in \mathbf{C}_i$, the equation $op = l(\mathsf{n}_1) = l(\mathsf{n}_2)$ is possible. Let us denote two different occurrences of op within the CSP part of a specification by op^1 and op^2.

In the decomposition of the specification, op^1 and op^2 occur in *both* parts, S_1 and S_2. Obviously, a synchronisation of op must be restricted to originally *corresponding* occurrences of op, that is, $S_1.op^1$ should be synchronised with $S_2.op^1$ and, accordingly, $S_1.op^2$ with $S_2.op^2$.

However, the synchronisation alphabet can no longer *distinguish* between these different occurrences. Therefore, non-corresponding instances of operations can be synchronised as well. This can particularly lead to additional traces for the CSP part of $S_1 \parallel S_2$.

In our example, the event c occurs twice within $S.\texttt{main}$ and thus twice in $S_1.\texttt{main}$ and $S_2.\texttt{main}$. A synchronisation of c within $S_1.\texttt{main} \parallel S_2.\texttt{main}$ can either result in the joint execution of corresponding occurrences, namely $S_1.c^1$ synchronising with $S_2.c^1$ and $S_1.c^2$ synchronising with $S_2.c^2$, as shown on the left hand side of Figure 4.20, or to an invalid synchronisation of $S_1.c^1$ with $S_2.c^2$ and $S_1.c^2$ with $S_2.c^1$, as shown on the right hand side of the same figure. The latter synchronisation triggers the previously identified path $\langle a, c, e \rangle$, which is invalid for $S.\texttt{main}$.

In Section 2.2.1, we introduced simple parameters [Fis00, Fis97] which can be restricted by both, the CSP part and the Object-Z part of a specification. This specific type

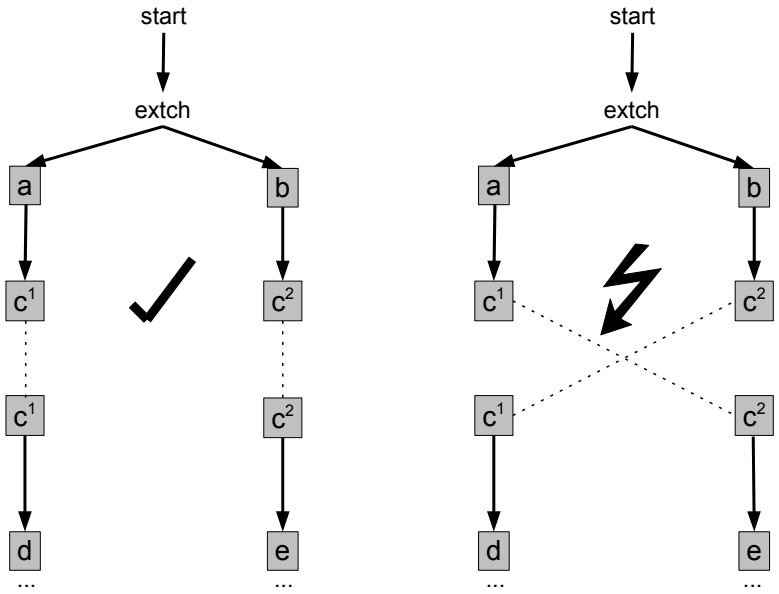


Figure 4.20: Synchronisation of events for external choice

of parameters will be used to define a set of additional *address* parameters to operations with a multiple occurrence in the cut. In our case, they will solely be restricted by the CSP part, and they do not occur in the Object-Z part of a component. We will modify the CSP parts of S_1 and S_2 by *fixing* the values for some of these parameters. As a synchronisation of two instances of an operation is only possible if their extension sets are not disjoint, differently fixed parameters can prevent a false synchronisation.

We illustrate the outcome of this extension on the previous example. For the event c, we will use one address parameter p_1 of type $\{1, 2\}$ and redefine

- $S_1.\texttt{main} := (a \rightarrow c.1 \rightarrow \texttt{Skip}) \,\Box\, (b \rightarrow c.2 \rightarrow \texttt{Skip})$ and
- $S_2.\texttt{main} := (c.1 \rightarrow d \rightarrow \texttt{Skip}) \,\Box\, (c.2 \rightarrow e \rightarrow \texttt{Skip})$.

A synchronisation of c^1 with c^2 over different components is now impossible.

In general, if no parallel composition is involved in a process, one additional address parameter is sufficient to separate two different occurrences of an operation from each other. However, when dealing with parallel composition, synchronising the operation under interest, one parameter is no longer adequate, since it would exclude part of the originally allowed synchronisation.

Recall Example 4.3.11 after replacing the external choice operator with $\|_{\{c\}}$. We get

$$\begin{array}{lll} S_1.\texttt{main} & := & (a \rightarrow c \rightarrow \texttt{Skip}) \\ & \|_{\{c\}} & (b \rightarrow c \rightarrow \texttt{Skip}), \\ S_2.\texttt{main} & := & (c \rightarrow d \rightarrow \texttt{Skip}) \\ & \|_{\{c\}} & (c \rightarrow e \rightarrow \texttt{Skip}). \end{array}$$

In this case, a joint synchronisation of the event c within $S_1.\texttt{main} \parallel_{\{c\}} S_2.\texttt{main}$ is allowed. This requires us to add two *fresh* parameters, not affecting each other, with one of them subsequently restricted for one branch of the parallel composition and the other one restricted for the other branch.

Summarising, we need to preserve and neither extend nor restrict the original synchronisation structure of S within $S_1 \parallel S_2$. The following definitions especially need to ensure that only *corresponding* instances of operations can be synchronised between S_1 and S_2.

In order to find a general solution for this problem by identifying a correct addressing extension for any process, including *nesting* of different types of branching, we recursively traverse its CFG with respect to any operation schema with multiple occurrence in the cut. An algorithm yielding a correct synchronisation is given in Section 5.1. To this end, we outline the general strategy. In addition, we define and show the required conditions on a correct addressing, which are realised by the algorithm. The algorithm proceeds as follows:

Traversing the CFG: Starting with the unique start-node, we recursively traverse the CFG of the process $S.\texttt{main}$. Let $op \in Op$ be the current operation under interest.

Initial parameter: Initially, we use one address parameter p_1 of type $\{1\}$. The type of any parameter can be extended throughout the traversal.

Active Parameters: Any branch of the CFG has one dedicated, *active* address parameter. The underlying idea is that this parameter possibly needs to be assigned with a specific value to prevent a false synchronisation within the associated branching. Initially, p_1 is declared active for the sole initial branch and assigned with the value 1. All assigned values are possibly modified during an execution of the algorithm. Besides, one parameter can be active for more than one branch.

No Branching: In case we proceed over a CFG operator which does not introduce branching, no changes to the parameters are committed.

Branching for $\mathsf{cfop} \in \{\mathsf{extch}, \mathsf{intch}, \mathsf{interleave}, \mathsf{par}_X\}$ **and** $op \notin X$**:** Branching without parallel composition of the operation under interest can lead to two occurrences of op within the cut, which need to be separated. In this case, the currently active parameter is declared active for both, the left and right branch. For the left branch, we keep the originally assigned value whereas for the right branch, we *increase* it by one, and we add the new value to the parameter's type. This ensures that an operation occurring in both branches cannot wrongly be synchronised.

Branching for par_X **and** $op \in X$**:** Branching with a synchronisation of op possibly leads to two occurrences of op, which still need to be able to be synchronised. In this case, the active parameter p_i, which belongs to the branch entering the parallel composition, can no longer be used: it may already be used to prevent a wrong synchronisation within a previous branching. The algorithm introduces two additional, fresh parameters p_{i+1} and p_{i+2}. The first parameter is declared active for the left branch, the second parameter is declared active for the right branch. As we solely

restrict each parameter on one side, a synchronisation of occurrences within the left branch and the right branch is always possible, independent of further restrictions of p_{i+1} and p_{i+2}.

Figure 4.21 illustrates the two different cases for branching.

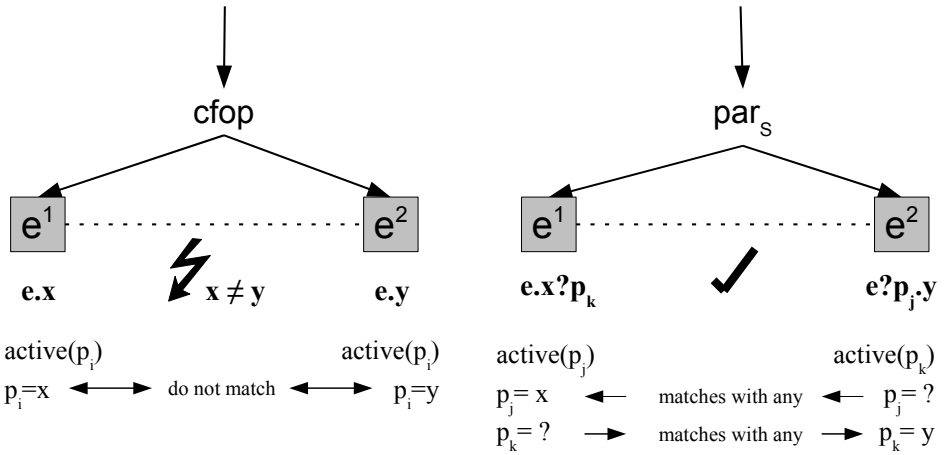


Figure 4.21: Addressing extension for CFG branching

In order to exemplify the necessity for introducing additional parameters in the case of a parallel composition and to clarify the general idea, we give an example. Figure 4.22 shows an extract of a possible control flow graph, for which we consider one operation b, element of a valid cut. The CFG proceeds over an external choice, followed by a parallel composition with b being synchronised and, finally, a two-sided external choice. As b occurs multiple times in the cut, an addressing extension is required. Based on our strategy, we introduce three additional parameters:

- p_1 is responsible for ensuring that no false synchronisation with respect to the outer external choice is possible, that is, b^1 must not be synchronised with any element of $\{b^2, b^3, b^4, b^5\}$. This is achieved by fixing p_1 to the value of 1 for the left branch and to 2 for the right branch.

- p_2 is responsible for excluding a wrong synchronisation within the left inner external choice, that is, between b^2 and b^3.

- p_3 forbids a synchronisation within the right inner external choice, that is, between b^4 and b^5.

- Finally, p_2 and p_3 are indeed necessary to ensure that any two elements of $\{b^2, b^3\}$ and $\{b^4, b^5\}$ can still be synchronised.

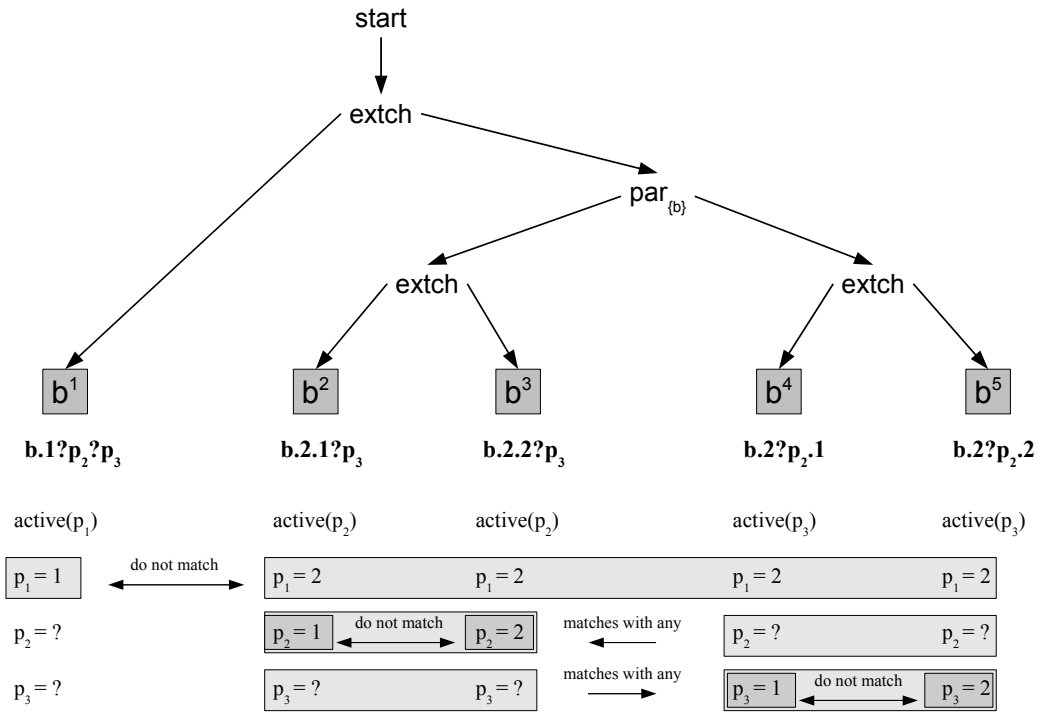


Figure 4.22: Addressing extension for nested branching

Having illustrated and exemplified our general strategy, we now give the details on the addressing extension. Based on the criterion **all-or-none**, all occurrences of an operation have to be assigned to *one* cut set, which we denote by $\mathbf{C}_i$.

We define two conditions on a parameter extension and subsequently show that they are sufficient to preserve the synchronisation structure of S within $S_1 \parallel S_2$. Here, we omit dealing with the original parameters of an operation *op*, since they are irrelevant for the subsequent proof. Both conditions correspond to the previously identified different cases for branching with and without a synchronisation of *op*.

In Definition 2.2.6, we introduced *partial* events. As the CSP part of a specification may restrict the set of simple parameters of an operation, any occurrence of an operation within the CSP part is a partial event. Subsequently, op_p denotes an arbitrary partial event for the channel *op*.

Definition 4.3.12. *(Conditions for correct addressing extension)*
Let $\mathtt{CFG}_S = (N, \longrightarrow)$ *be the control flow graph of a specification S, and let* $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. Furthermore, let* $op \in Op_{\mathbf{C}}$ *such that* op *occurs at least twice in either* $\mathbf{C}_1$ *or* $\mathbf{C}_2$*. Let* op^k *denote an arbitrary occurrence of the operation op within* $\mathtt{CFG}_S$ *and* op_p^k *its corresponding occurrence within* $S.\mathtt{main}$*. The* address requirements *for a correct synchronisation are given by the following two conditions, which need to hold for any* $i \neq j$*:*

Branching without Synchronisation: *If* op^i *and* op^j *are located inside different branches of either an external choice operator, internal choice operator, interleaving operator or a parallel composition operator* par_X *with* $op \notin X$, op_p *needs to comprise one parameter* p_1 *such that its type includes* $x, y \in \mathbb{N}$ *with* $x \neq y$. *This parameter is fixed to x for* op_p^i *and to y for* op_p^j *in both,* S_1.`main` *and* S_2.`main` :

op_p^i *becomes* $op_p^i.x$ *and* op_p^j *becomes* $op_p^j.y$.

This corresponds to the left hand side of Figure 4.21.

Branching with Synchronisation: *If* $\mathsf{op}^i \overset{\mathtt{sd}}{\dashleftarrow\!\dashrightarrow} \mathsf{op}^j$, *the (partial) event* op_p *needs to comprise two parameters* p_1 *and* p_2, *such that the type of* p_1 *includes* $x \in \mathbb{N}$ *and the type of* p_2 *includes* $y \in \mathbb{N}$ *for arbitrary* x, y. *The first parameter is fixed to x for* op_p^i *whereas the second parameter is fixed to y for* op_p^j *in both,* S_1.`main` *and* S_2.`main` :

op_p^i *becomes* $op_p^i.x?p_2$ *and* op_p^j *becomes* $op_p^j?p_1.y$.

This corresponds to the right hand side of Figure 4.21.

We give an intuitive description of these conditions. Example 4.3.11 illustrated that two different occurrences of an operation can spuriously be synchronised over different branches of an external choice operator. This problem can correspondingly occur for any CSP operator, which introduces branching into the CFG. In order to prevent this from happening, the first condition uses a parameter p_1 for the respective operation, which is differently fixed in both branches. Thus, a wrong synchronisation is no longer possible, as the extension sets of the partial events are now disjoint. If any two occurrences of the same operation were not allowed to synchronise beforehand, our addressing extension ensures an empty intersection of their extensions, not allowing for a synchronisation afterwards.

Additionally, we have to ensure that a previous synchronisation is not excluded due to our extension. Here, the second condition requires that for any parallel composition including *op*, two additional parameters are introduced, which can subsequently be restricted for their corresponding branch without influencing a synchronisation over both branches.

In order to restore the original synchronisation structure of a specification by the introduction of additional address parameters, we need to precisely state when two occurrences of the same operation were previously *allowed to synchronise*. Beforehand, we define a condition, stating that a certain synchronisation dependence can be *realised* by means of the underlying CSP process: there indeed exist traces, leading to the joint execution of both events. From now on, we let $foot(tr)$ denote the last element of the CSP trace *tr* according to [Sch99].

Definition 4.3.13. *(Realisation of synchronisation dependence)*
Let $op \in Op$ *such that* op_p^i *and* op_p^j, $i \neq j$, *are two different occurrences of op within the CSP part of S. Let* op^i *and* op^j *denote their corresponding nodes of* $\mathtt{CFG}_S$ *such that* $\mathsf{op}^i \overset{\mathtt{sd}}{\dashleftarrow\!\dashrightarrow} \mathsf{op}^j$.

For the CFG node par_X *being responsible for* $\mathsf{op}^i \overset{\mathtt{sd}}{\dashleftarrow\!\dashrightarrow} \mathsf{op}^j$*, let* P_1 *and* P_2 *denote the CSP processes corresponding to the first branch and the second branch of* par_X *within* $\mathtt{CFG_S}$*, respectively. If*

$$\exists\, tr_1 \in traces(P_1), tr_2 \in traces(P_2) \bullet \\ (tr_1 \restriction X = tr_2 \restriction X) \wedge (foot(tr_1) = op_p^i) \wedge (foot(tr_2) = op_p^j),$$

we say that the synchronisation dependence connecting op_p^i *and* op_p^j can be realised.

For two events to *allow for synchronisation*, their corresponding operation nodes of the CFG must be connected via a synchronisation dependence which can be realised. In addition, the intersection of the extension sets of the partial events corresponding to these nodes is non-empty. All conditions combined ensure that op^i and op^j can indeed synchronously be executed.

Definition 4.3.14. *(Allowed synchronisation)*
Let $op \in Op$ *such that* op_p^i *and* op_p^j*,* $i \neq j$*, are two different occurrences of op within the CSP part of S. Let* op^i *and* op^j *denote their corresponding nodes of the CFG. We say that* op_p^i and op_p^j allow for synchronisation within *S, if, and only if,*

a) $\mathsf{op}^i \overset{\mathtt{sd}}{\dashleftarrow\!\dashrightarrow} \mathsf{op}^j$ *within the CFG of S,*

b) $\mathsf{op}^i \overset{\mathtt{sd}}{\dashleftarrow\!\dashrightarrow} \mathsf{op}^j$ *can be realised,*

c) $\{\!|\, op_p^i \,|\!\} \cap \{\!|\, op_p^j \,|\!\} \neq \varnothing$.

Before proving the correctness of the conditions of the previous definition, we show the following property: if two nodes x, y of are *not* located in different CFG branches attached to the same node, they have to be connected by a CFG path.

Lemma 4.3.15. *(Non-opposite branching requires CFG path)*
Let $\mathtt{CFG_S} = (N, \longrightarrow)$ *be the CFG of a specification S. For any node* $\mathsf{cfop} \in \{\mathsf{extch}, \mathsf{intch}, \mathsf{par}, \mathsf{interleave}\}$*, let* $\mathsf{fst(cfop)}$ *denote one branch and* $\mathsf{snd(cfop)}$ *the other branch of* cfop*, before reaching the join-node* uncfop*. For any* n, n' $\in \mathtt{cf}(N)$*, if*

$$\nexists \mathsf{cfop} \in \{\mathsf{extch}, \mathsf{intch}, \mathsf{par}, \mathsf{interleave}\} \bullet \\ (\mathsf{n} \in \mathsf{fst(cfop)} \wedge \mathsf{n}' \in \mathsf{snd(cfop)}) \vee (\mathsf{n} \in \mathsf{snd(cfop)} \wedge \mathsf{n}' \in \mathsf{fst(cfop)}),$$

either $\mathsf{n} \overset{*}{\longrightarrow} \mathsf{n}'$ *or* $\mathsf{n}' \overset{*}{\longrightarrow} \mathsf{n}$.

Proof. Let cfop^1, cfop^2 denote the innermost operators with n, n′ being located inside one of their respective branches. In case that a node is not located inside of any branching, $\mathsf{cfop}^1 = \mathsf{cfop}^2 = \mathsf{start}$. Thus, we do not separately need to deal with start.

Case 1: $\mathsf{cfop}^1 = \mathsf{cfop}^2$ Based on the assumption, n and n′ have to be located in the *same* branch of the operator. As we chose cfop^i to be the innermost branching, n and n′ are both located on the sole path from cfop^i to n or from cfop^i to n′, dependent on which node is visited first.

Case 2: $\mathsf{cfop}^1 \neq \mathsf{cfop}^2$ Based on the assumption, there is not outer operator cfop^o with n and n' being located in different branches of cfop^o. Therefore, for $i \neq j$, either cfop^i terminates before cfop^j, yielding a path from one node to the other one via uncfop^i. Otherwise, cfop^i is located inside of cfop^j, also yielding a CFG path from one node to the other one. □

The following theorem shows that the previously identified conditions on an addressing extension are sufficient.

Theorem 4.3.16. *(Definition 4.3.12 ensures correct synchronisation of S_1 and S_2)*
Let $\mathtt{CFG}_S = (N, \longrightarrow)$ be the control flow graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. For any $op \in Op_{\mathbf{C}}$ with multiple occurrence within the CFG and CSP part of S, let both conditions of Definition 4.3.12 be satisfied. Then, the original horizontal synchronisation structure of S is preserved within $S_1 \parallel S_2$, whereas no additional synchronisation is introduced. Precisely, for $op_p^i \neq op_p^j$:

(1) Possible synchronisation for duplicated nodes:
$S_1.op_p^i$ and $S_2.op_p^i$ allow for synchronisation in $S_1 \parallel S_2$.

(2) Original synchronisation is preserved within S_1 and S_2:
If $S.op_p^i$ and $S.op_p^j$ allow for synchronisation in S, then $S_1.op_p^i$ [$S_2.op_p^i$] and $S_1.op_p^j$ [$S_2.op_p^j$] allow for synchronisation in S_1 [S_2].

(3) Original synchronisation is preserved within $S_1 \parallel S_2$:
If $S.op_p^i$ and $S.op_p^j$ allow for synchronisation in S, then $S_1.op_p^i$ [$S_1.op_p^j$] and $S_2.op_p^j$ [$S_2.op_p^i$] allow for synchronisation in $S_1 \parallel S_2$.

(4) No additional synchronisation within S_1:
If $S.op_p^i$ and $S.op_p^j$ do not *allow for synchronisation in S, then $S_1.op_p^i$ and $S_1.op_p^j$ do not allow for synchronisation in S_1.*[2]

(5) No additional synchronisation within $S_1 \parallel S_2$:
If $S.op_p^i$ and $S.op_p^j$ do not *allow for synchronisation in S, then $S_1.op_p^i$ [$S_2.op_p^i$] and $S_2.op_p^j$ [$S_1.op_p^j$] do not allow for synchronisation in $S_1 \parallel S_2$.*

Proof. Assume that both conditions of Definition 4.3.12 hold. In case we refer to op_p^i, op_p^j, we implicitly assume $i \neq j$. We show the respective properties by applying the conditions from the definition:

(1) $S_1.op_p^i$ and $S_2.op_p^i$ result from a duplication of $S.op_p^i$. Let us denote the corresponding CFG nodes within the CFG of $S_1 \parallel S_2$ by op_1^i and op_2^i. We have to show all three conditions of Definition 4.3.14.

[2] Note that both events might indeed allow for synchronisation within S_2. This does not pose a problem as in this case, a synchronisation over the cut would have to involve all four events $S_1.op_p^i$, $S_1.op_p^j$, $S_2.op_p^i$ and $S_2.op_p^j$ which is impossible if $S_1.op_p^i$ and $S_1.op_p^j$ cannot be synchronised.

a) Starting from par_{Op_C}, there exist paths $\mathsf{par}_{Op_C} \xrightarrow{\pi} \mathsf{op}_1^i$ and $\mathsf{par}_{Op_C} \xrightarrow{\pi'} \mathsf{op}_2^i$. π and π' do not share any additional nodes, as par_{Op_C} is the outermost operator of the CFG for $S_1 \parallel S_2$. This ensures the conditions on a synchronisation dependence, as given in Definition 2.3.6. ✓

b) This dependence can be realised: for the traces tr and tr' corresponding to the paths π and π', the equation $tr \restriction Op_C = tr' \restriction Op_C$ holds. Traces restricted on the set of cut operations are preserved by the projection of the CSP process $S.\texttt{main}$ on $S_i.\texttt{main}$, as cut events occur in both, S_1 and S_2, and as Definition 4.3.3 does not modify the structure of a process. ✓

c) Additionally, since we refer to two nodes corresponding to the same node of $\texttt{CFG}_S$, and since the addressing extension is identical for both, $S_1.\texttt{main}$ and $S_2.\texttt{main}$, the inequality $\{\mid S_1.op_p^i.add \mid\} \cap \{\mid S_2.op_p^i.add \mid\} \neq \varnothing$ trivially holds for any possible addressing extension add. ✓

(2) Assume that op_p^i and op_p^j allow for synchronisation in S. Again, we show all three conditions for allowed synchronisation of $S_i.op_p^i$ and $S_i.op_p^j$.

a) By assumption, op^i and op^j are connected via a synchronisation dependence in S, and both nodes are elements of $\mathbf{C}_i$. Corresponding to the previous case, they are still connected via a synchronisation dependence in S_1 and in S_2, as Definition 4.3.3 does not modify the branching structure of a process. ✓

b) Let par_X be responsible for the synchronisation dependence between op_p^i and op_p^j, and let tr and tr' be the traces realising the dependence. Then, $tr \restriction X = tr' \restriction X$. As both paths are correspondingly projected within $S_i.\texttt{main}$, we get

$$(tr \restriction (Op_i \cup Op_C)) \restriction X = (tr' \restriction (Op_i \cup Op_C)) \restriction X.$$

Thus, the synchronisation dependence can be realised. ✓

c) Finally, the second condition on correct addressing results in op_p^i being replaced by $op_p^i.x?p_2$ and op_p^j being replaced by $op_p^j?p_1.y$ in both, S_1 and S_2, for some $x, y \in \mathbb{N}$. This implies

$$\{\mid op_p^i.x._- \mid\} \cap \{\mid op_p^j._-.y \mid\} \neq \varnothing$$

based on the assumption

$$\{\mid op_p^i \mid\} \cap \{\mid op_p^j \mid\} \neq \varnothing. ✓$$

(3) Assume that op_p^i and op_p^j allow for synchronisation in S.

a) According to (1), we get two paths π and π' in the CFG of $S_1 \parallel S_2$, which start in par_{Op_C} and reach the respective occurrences of op^i and op^j without additional shared nodes, thus yielding a synchronisation dependence. ✓

b) This dependence can be realised: the projection of $S.\texttt{main}$ on Op_C yields the same traces within $S_1.\texttt{main}$ and $S_2.\texttt{main}$. ✓

c) Finally, in correspondence to the previous case,

$$\{|\, op_p^i.x._- \,|\} \cap \{|\, op_p^j._-.y \,|\} \neq \varnothing. \checkmark$$

(4) Assume that op^i and op^j do not allow for synchronisation in S due to a violation of conditions a), b) or c):

a): In either case, a missing synchronisation dependence cannot be introduced due to the projection. ✓

b): Let par_X be responsible for the synchronisation dependence between op_p^i and op_p^j. Let P_1 and P_2 denote the CSP processes corresponding, to the first and second branch of par_X. Then, there are no traces $tr_1 \in traces(P_1)$ and $tr_2 \in traces(P_2)$, such that $tr_1 \restriction X = tr_2 \restriction X$. As $op \in Op_{\mathbf{C}}$, the projection of the CSP process $S.\texttt{main}$ on $S_1.\texttt{main}$ preserves the original traces with respect to X: no events of Op_2 can be involved, thus ensuring that the synchronisation dependence cannot be realised within $S_1.\texttt{main}$. ✓

c): $\{|\, op_p^i \,|\} \cap \{|\, op_p^j \,|\} = \varnothing$ is preserved by the addressing extension. ✓

(5) Again assume that one of three conditions for allowed synchronisation is violated:

a): We distinguish between two cases:

Case 1: Both nodes are located inside different branches of either an external choice-, internal choice- or interleaving operator or a parallel composition operator par_X with $op \notin X$. The first condition on correct addressing results in op_p^i being replaced by $op_p^i.x$ and op_p^j being replaced by $op_p^i.y$, $x \neq y$, within $S_1.\texttt{main}$ and $S_2.\texttt{main}$. Thus, even though both nodes are possibly connected via a newly added synchronisation dependence within the CFG of $S_1 \| S_2$, the events $S_1.op_p^i.x$ and $S_2.op_p^j.y$ do not allow for synchronisation according to Definition 4.3.14, since $\{|\, op_p^i.x \,|\} \cap \{|\, op_p^j.y \,|\} = \varnothing$ holds. The same holds for S_1 and S_2 switched.

Case 2: The premise of case 1 does not hold. A parallel composition with op being synchronised is impossible, as there is no synchronisation dependence connecting both nodes. Thus, branching is ruled out. Based on Lemma 4.3.15, there exists a CFG path π starting in op^i and reaching op^j (opposite direction accordingly). This path does not include any operation nodes outside of $\mathbf{C}_i$ since otherwise, the cut would be left and re-entered, causing a violation of the correctness criterion **no reaching back**. In particular, for any two paths $\mathsf{par}_{Op_{\mathbf{C}}} \xrightarrow{\pi_1} \mathsf{op}^i$ and $\mathsf{par}_{Op_{\mathbf{C}}} \xrightarrow{\pi_2} \mathsf{op}^j$, the traces tr, tr_1 and tr_2 corresponding to the paths π, π_1 and π_2 yield $tr_1 \restriction Op_{\mathbf{C}} \neq tr_2 \restriction Op_{\mathbf{C}}$, as op_p^j is an element of the latter but not the first trace. This violates that the synchronisation dependence can be realised. The same holds for S_1 and S_2 switched. ✓

b): Again, let par_X be responsible for the synchronisation dependence between op_p^i and op_p^j. A violation of the possible realisation of a synchronisation dependence

yields that there are no $tr_1 \in traces(P_1)$ and $tr_2 \in traces(P_2)$ such that $tr_1 \restriction X = tr_2 \restriction X$. In particular, as $op \in (Op_{\mathbf{C}} \cap X)$, a synchronisation of op within $S_1 \parallel S_2$ would have to involve all four occurrences of op. However, according to (4,b.), the projection of $S.\texttt{main}$ preserves the traces with respect to X up to reaching the cut within $S_1.\texttt{main}$. Thus, op_p^i and op_p^j are not allowed to synchronise within S_1. A diagonal synchronisation between op_p^i and op_p^j is impossible as well, as the synchronisation dependence cannot be realised due to S_1. ✓

c): A violation of Condition c) is trivially preserved within $S_1 \parallel S_2$. ✓ □

Figure 4.23 illustrates the allowed and forbidden synchronisations due to the three conditions of the lemma. A solid line depicts an allowed synchronisation, whereas a dotted lines depicts the opposite.

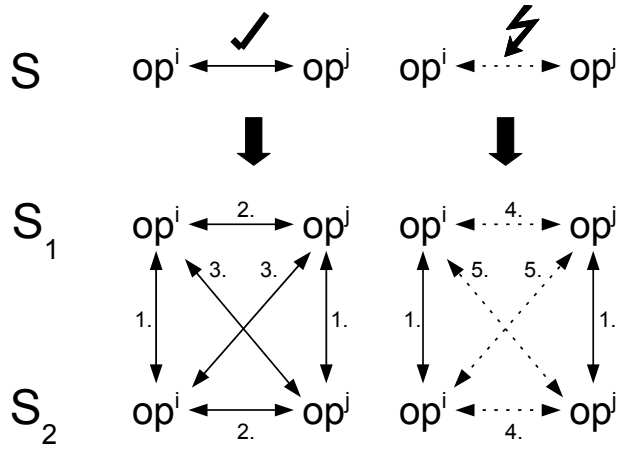


Figure 4.23: Illustration of Theorem 4.3.16

Separating Operations Shared between Op_1 and Op_2

Another aspect which we have to deal with tackles the fact that in general, Op_1 and Op_2 are not disjoint. This can lead to one operation being assigned to both, S_1 and S_2. We need to ensure that the projection of a CSP process correctly eliminates the subset of occurrences of an operation which are no longer part of the respective component. A projection, keeping the set of *all* occurrences, is generally insufficient:

Example 4.3.17. *Let* $S.\texttt{main} := a \rightarrow b \rightarrow a \rightarrow \texttt{Skip}$ *and* $\mathbf{C} = \{b\}$ *be a valid (single) cut. Then, the first occurrence of a should be an element of* $S_1.\texttt{main}$ *whereas the second one should be an element of* $S_2.\texttt{main}$*. A projection of* $S.\texttt{main}$ *on* $\{a,b\}$ *would result in* $S_1.\texttt{main} = S_2.\texttt{main} = S.\texttt{main}$ *and is therefore infeasible.*

As Op_1 and Op_2 are not disjoint, we need to separate the occurrences of an operation $op \in Op_1 \cap Op_2$ within S_1 from the ones within S_2. Corresponding to the previous section, we will use one additional address parameter $p_1 : \{1, 2\}$ for any operation with its occurrences distributed over Op_1 and Op_2. The parameter is fixed to 1 for occurrences within S_1 and accordingly fixed to 2 for occurrences within S_2.

For the event a of Example 4.3.17, we get

- $S_1.\texttt{main} := a.1 \rightarrow b \rightarrow \texttt{Skip}$ and
- $S_2.\texttt{main} := b \rightarrow a.2 \rightarrow \texttt{Skip}$.

Defining the Sets of Events for S_1 and S_2

Based on the additional parameters and their restrictions, the overall system definition needs to be adapted. First, we observe the following:

- For any $op \notin Op_{\mathbf{C}}$, there exist one (if $op \in (Op_1 \cap Op_2)$) or zero additional address parameters. For simplification, we will denote this possible additional parameter by p_1. $[v]$ denotes that the value of the parameter p_1 is set to v, if the parameter exists.
- For $op \in Op_{\mathbf{C}}$, *any* number of parameters is possible. However, for the set of cut operations, *all* possible extensions of operations need to be represented in the synchronisation alphabet. This is due to the correctness criterion **all-or-none** and the fact that the respective addressing is identical for both, S_1 and S_2.

Both observations allow for the following definition:

Definition 4.3.18. *(Event sets of components)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ be the control flow graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut, yielding the four sets $Op_1, Op_2, Op_{\mathbf{C}_1}$ and $Op_{\mathbf{C}_2}$, now possibly comprising additional address parameters. The event sets for the decomposition of S into S_1 and S_2 *are given by*

$$E_1 := \bigcup_{op \in Op_1} \{|\ op\_[.1]\ |\}, \quad E_2 := \bigcup_{op \in Op_2} \{|\ op\_[.2]\ |\},$$

$$E_{\mathbf{C}_1} := \{|\ Op_{\mathbf{C}_1}\ |\}, \qquad E_{\mathbf{C}_2} := \{|\ Op_{\mathbf{C}_2}\ |\}$$

where "_" denotes the original parameters of the channel. Let $E_{\mathbf{C}} := E_{\mathbf{C}_1} \cup E_{\mathbf{C}_2}$, $E_{S_1} := E_1 \cup E_{\mathbf{C}}$ and $E_{S_2} := E_2 \cup E_{\mathbf{C}}$.

The following lemma describes that all events shared between S_1 and S_2 are elements of $E_{\mathbf{C}}$:

Lemma 4.3.19. *(Common events of S_1 and S_2 solely occur in the cut)*
Let E_1, E_2 be defined according to Definition 4.3.18. Then:

$$E_1 \cap E_2 = \varnothing.$$

Proof. Assume that there exists $e \in (E_1 \cap E_2)$. Then, $e \in \{|\ op\ |\}$ holds for some $op \in (Op_1 \cap Op_2)$. Based on the addressing extension for shared operations, op is thus extended by one address parameter of type $\{1, 2\}$. Either this value is set to 1 implying $e \notin E_2$ or to 2 implying $e \notin E_1$, contradiction. □

4.3.4 Renaming for the Decomposition

The previous section introduced additional parameters to operations of S_1 and S_2, required to ensure an equivalent data flow and control flow between S and $S_1 \parallel S_2$. These parameters modify the original *types* of the channels of S. In our correctness proof, which is given in Chapter 5, we thus show that S and $S_1 \parallel S_2$ are equivalent modulo different channel types. As we need to refer to the precise sets of events of a specification, we will from now on write E_S to denote the set of events of a specification S.

For describing the difference between E_S and E_{S_i}, we introduce

- a function f, mapping a channel of S on the corresponding channel within S_i, now comprising additional parameters and
- two event renaming relations $\mathtt{R}_1^{\mathtt{C}} : E_S \rightarrow E_{S_1}$ and $\mathtt{R}_2^{\mathtt{C}} : E_S \rightarrow E_{S_2}$, applied on the process $S.\mathtt{main}$, in order to determine $S_1.\mathtt{main}$ and $S_2.\mathtt{main}$.[3]

We start with the function f mapping channels of S on channels of S_i. It implicitly defines a corresponding extension of the declaration parts of the Object-Z schemas, now additionally containing transmission parameters.[4] According to the notation for transmission parameters, let

$$op.add = add_1 : r_1;\ \ldots;\ add_k : r_k$$

denote the set of address parameters of an operation op, and let

$$op.orig = p_1 d_1 : s_1;\ \ldots;\ p_l d_l : s_l$$

with $d_i \in \{?, !, \epsilon\}$ (where ϵ denotes the empty decoration used for simple parameters) the set of original parameters of op, as defined within the interface of S:

Definition 4.3.20. *(Renaming of channels)*
Let S be a specification, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$. The channel renaming for the decomposition of S into S_1 and S_2 *is given by*

$$f(op : [op.orig]) = \begin{cases} op : [op.orig;\ op.tr\_in^1;\ op.add], & op \in Op_{\mathbf{C}_1}, \\ op : [op.orig;\ op.tr\_in^2;\ op.add], & op \in Op_{\mathbf{C}_2}, \\ op : [op.orig;\ op.add], & otherwise. \end{cases}$$

Note that, in the last case, $op.add$ comprises zero or one address parameter whereas in the other cases, the amount is indefinite. Further note that we never leave out any original parameters, as the types of the shared operations have to coincide.

[3] Note that in CSP-OZ, according to [Fis00], and in contrast to pure Z, renaming of CSP processes is not restricted to functions – relations can be used as well.

[4] As address parameters are not restricted by the Object-Z part, we omit them in the declaration parts of Object-Z schemas.

Next, we introduce two renaming relations, determining two processes, which are subsequently used for the definition of $S_1.\texttt{main}$ and $S_2.\texttt{main}$. For an operation $op \in Op$, we let

$$op.tr\_in = tr_1? : t_1;\ \ldots;\ tr_n? : t_n$$

denote the additional transmission parameters of an *arbitrary* operation. Moreover, let a_i denote the possibly fixed value of the address parameter add_i according to the restriction of address parameters. The following event renaming is *relational*, as it maps an event on a *set* of events. We simply write $op?p$ to denote the set $\{op.x \mid x : t_p\}$. This notation is motivated by the equivalence between $op?p \to P$ and $\Box_{x:t_p}\, op.x \to P$.

Definition 4.3.21. *(Renaming of events)*
Let S be a specification, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$. The (relational) event renaming for the decomposition of S into S_1 and S_2 *is given by*

$$\mathtt{R}_1^{\mathbf{C}} : E_S \to E_{S_1} \text{ and } \mathtt{R}_2^{\mathbf{C}} : E_S \to E_{S_2},$$

defined as

$$\mathtt{R}_1^{\mathbf{C}}(op.x) := \begin{cases} op.x.1, & op \in (Op_1 \cap Op_2) \setminus (Op_{\mathbf{C}_1} \cup Op_{\mathbf{C}_2}), \\ op.x?tr_1 \ldots ?tr_n.a_1 \ldots a_k, & op \in Op_{\mathbf{C}} \wedge |l^{-1}(op)| > 1, \\ op.x?tr_1 \ldots ?tr_n, & op \in Op_{\mathbf{C}} \wedge |l^{-1}(op)| = 1, \\ op.x, & \text{otherwise} \end{cases}$$

and

$$\mathtt{R}_2^{\mathbf{C}}(op.x) := \begin{cases} op.x.2, & op \in (Op_1 \cap Op_2) \setminus (Op_{\mathbf{C}_1} \cup Op_{\mathbf{C}_2}), \\ op.x?tr_1 \ldots ?tr_n.a_1 \ldots a_k, & op \in Op_{\mathbf{C}} \wedge |l^{-1}(op)| > 1, \\ op.x?tr_1 \ldots ?tr_n, & op \in Op_{\mathbf{C}} \wedge |l^{-1}(op)| = 1, \\ op.x, & \text{otherwise}. \end{cases}$$

Graphically explained, the renaming introduces additional transmission and address parameters to the original events, if required. For operations not represented in the cut, no transmission parameters are introduced. Shared operations of Op_1 and Op_2 receive one address parameter fixed to 1 and 2, respectively, whereas local operations to one specification do not receive any additional parameters. For the cut, we introduce a possibly empty set of transmission parameters. For the address parameters, we separate operations with multiple occurrence in the cut from the ones with single occurrence: the first operations receive additional address parameters, whereas the latter ones do not.

The previous definitions allow us to give the *final* definitions for the interfaces and CSP parts of S_1 and S_2. We start with a modification of Definition 4.3.2, which now takes the channel renaming f into account:

Definition 4.3.22. *(Interfaces of components, final definition)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ be the dependence graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. Let f be the channel renaming function according to Definition 4.3.20. The interfaces for the decomposition of S into S_1 and S_2 are defined as

- $S_1.I := f(I)|_{(Op_1 \cup Op_{\mathbf{C}})}$ *and* *(Interface for S_1)*
- $S_2.I := f(I)|_{(Op_2 \cup Op_{\mathbf{C}})}$. *(Interface for S_2)*

In order to modify the CSP parts of the components according to Definition 4.3.4, we apply the event renaming on `main`. Note that the following holds for any renaming relation `R` ([Sch09]):

$$(e \rightarrow P)[\![\mathtt{R}]\!] = e' : \mathtt{R}(e) \rightarrow P[\![\mathtt{R}]\!].$$

Definition 4.3.23. *(CSP parts of components, final definition)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ be the dependence graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. Let $\mathtt{R}_1^{\mathbf{C}}$ and $\mathtt{R}_2^{\mathbf{C}}$ be the event renaming relations according to Definition 4.3.21. The CSP parts for the decomposition of S into S_1 and S_2 are defined as

- $S_1.\mathtt{main} := (S.\mathtt{main}[\![\mathtt{R}_1^{\mathbf{C}}]\!])|_{E_{S_1}}$ *and* *(CSP part for S_1)*
- $S_2.\mathtt{main} := (S.\mathtt{main}[\![\mathtt{R}_2^{\mathbf{C}}]\!])|_{E_{S_2}}$. *(CSP part for S_2)*

In Figure 4.19, we implicitly modified the channel *change_m* of *Increaser* after the introduction of one transmission parameter. As address parameters are not required for the decomposition, the specification's decomposition is final.

Summarising the previous definition, we are now able to give the final definition for the *thorough* decomposition of S into S_1 and S_2.

4.3.5 Definition of the Decomposition

After ensuring a correct data flow within $S_1 \parallel S_2$ based on the introduction of additional transmission parameters and ensuring a correct control flow based on additional address parameters, we finally give the definition of the *thorough* decomposition of S into components S_1 and S_2 by modifying Definition 4.3.8:

Definition 4.3.24. *(Decomposition with respect to a cut, final definition)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ be the dependence graph of a specification S, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut. Let

$$Op_1, Op_2, Op_{\mathbf{C}_1}, Op_{\mathbf{C}_2}, Op_{\mathbf{C}}$$

be defined according to Definition 4.3.1. The decomposition of S with respect to $(\mathbf{C}_1, \mathbf{C}_2)$ into S_1 and S_2 *is defined as*

S_1	
$S_1.I$	*[*according to Definition 4.3.22*]*
$S_1.\mathtt{main}$	*[*according to Definition 4.3.23*]*
$S_1.\mathtt{State}$	*[*according to Definition 4.3.5*]*
$S_1.\mathtt{Init}$	*[*according to Definition 4.3.6*]*
$S_1.op$	*[*according to Definition 4.3.10*]*

S_2	
$S_2.I$	*[*according to Definition 4.3.22*]*
$S_2.\texttt{main}$	*[*according to Definition 4.3.23*]*
$S_2.\texttt{State}$	*[*according to Definition 4.3.5*]*
$S_2.\texttt{Init}$	*[*according to Definition 4.3.6*]*
$S_2.op$	*[*according to Definition 4.3.10*]*

The system, generated from the components, is defined according to Definition 4.3.8 as the parallel composition of both classes, synchronising on the set of cut events:

$$S_1 \parallel_{E_\mathbf{C}} S_2.$$

For the remainder of this thesis, we let $E_{S'} := E_{S_1} \cup E_{S_2}$ and $Op' := Op_1 \cup Op_2 \cup Op_\mathbf{C}$. The following theorem states the main result of this thesis. The correctness proof will be shifted to the next chapter.

Theorem 4.3.25. *(Correctness of the decomposition)*
Let S be a specification, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ *be a cut, yielding a decomposition into* S_1 *and* S_2 *according to Definition 4.3.24. Then, the following holds:*

$$S =_T (S_1 \parallel_{E_\mathbf{C}} S_2)[\![\mathtt{R}']\!], \tag{4.1}$$

where $\mathtt{R}' : E_{S'} \to E_S$ *is defined as*

$$\mathtt{R}'(op.x.t_1 \ldots t_n.a_1 \ldots a_k) := op.x$$

with t_i *denoting the values for the possible* transmission *parameters of op and* a_i *the values for its possible* address *parameters.*

Based on several lemmas and some additional prearrangements, the proof is given in Chapter 5, Section 5.6. The next section illustrates the decomposition on our case study of a candy machine. It is based on the single cut $\mathbf{C} := \{\mathsf{switch}\}$.

4.3.6 Candy Machine Revisited: Decomposition

Recall the main case study of this thesis, the specification of a candy machine, as given in Figure 2.3. We already identified the set $\mathbf{C} := \{\mathsf{switch}\}$ to be a valid single cut in Section 4.2.4.

First, the definition for $\mathbf{Ph}_1, \mathbf{C}_1$ and $\mathbf{Ph}_2$ yields

- $Op_1 = \{pay, payout, abort\}$,
- $Op_\mathbf{C} = \{switch\}$ and
- $Op_2 = \{select, order, term, deliver\}$.

The projections of $S.\texttt{main}$ on the remaining sets of events,

- $S_1.\texttt{main} := S.\texttt{main}|_{\{|pay,payout,abort,switch|\}}$ and
- $S_2.\texttt{main} := S.\texttt{main}|_{\{|switch,select,order,term,deliver|\}}$,

lead to

CandyMachine$_1$

$$
\begin{array}{rcl}
[\ldots] & & \\
\texttt{main} & \stackrel{c}{=} & pay?coin \rightarrow \texttt{main} \,\Box\, Payout \,\Box\, switch \rightarrow \texttt{Skip} \\
Payout & \stackrel{c}{=} & payout?coin \rightarrow Payout \,\Box\, abort \rightarrow \texttt{Skip} \\
[\ldots] & &
\end{array}
$$

CandyMachine$_2$

$$
\begin{array}{rcl}
[\ldots] & & \\
\texttt{main} & \stackrel{c}{=} & \texttt{Skip} \,\Box\, switch \rightarrow Select \\
Select & \stackrel{c}{=} & (select?ca \rightarrow (Select \,\Box\, Order)) \,\Box\, Deliver \\
Order & \stackrel{c}{=} & order \rightarrow Select \\
Deliver & \stackrel{c}{=} & deliver?ca \rightarrow Deliver \,\Box\, term?rest \rightarrow \texttt{Skip} \\
[\ldots] & &
\end{array}
$$

after applying several simplifications. For the sets of state variables of *CandyMachine*$_1$ and *CandyMachine*$_2$, we get

- $S_1.V = \{sum, paid, credits\}$ and
- $S_2.V = \{credits, items, selected\}$.

$S_1.V$ and $S_2.V$ determine the respective state schemas. The initial state schemas are given by

$$
\begin{array}{rcl}
S_1.\texttt{Init} & = & \exists selected : Candies, items : \operatorname{seq} Candies \bullet \\
 & & (sum = 0 \land paid = \langle\,\rangle \land items = \langle\,\rangle) \\
 & \equiv & (sum = 0 \land paid = \langle\,\rangle)
\end{array}
$$

and

$$
\begin{array}{rcl}
S_2.\texttt{Init} & = & \exists sum : \mathbb{N}, paid : \operatorname{seq} Coins, credits : \mathbb{N} \bullet \\
 & & (sum = 0 \land paid = \langle\,\rangle \land items = \langle\,\rangle) \\
 & \equiv & items = \langle\,\rangle.
\end{array}
$$

In order to determine the operation schemas of the components, we first need to compute the set of cut variables with respect to $\mathbf{C}_1 = \{\mathsf{switch}\}$. The operation schema *switch* modifies three different variables, namely *sum*, *credits* and *paid*. However, only one of them is subsequently referenced: *credits*. Based on the three data dependences by reason of *credits*,

- switch $\overset{\texttt{dd}}{\dashrightarrow}_{(credits)}$ select,
- switch $\overset{\texttt{dd}}{\dashrightarrow}_{(credits)}$ order and
- switch $\overset{\texttt{dd}}{\dashrightarrow}_{(credits)}$ term,

we get $CV_1 = \{credits\}$. Therefore, *switch* needs to be extended by one additional transmission parameter $tr_c : \mathbb{N}$. We are now able to define $CandyMachine_1.switch$ and $CandyMachine_2.switch$:

CandyMachine$_1$

$[\ldots]$

`enable_`*switch*

$sum \geq 2$

`effect_`*switch*

$\Delta(sum, credits, paid);\ tr_c! : \mathbb{N}$

$sum' = 0 \land paid' = \langle\,\rangle$
$credits' = sum \land tr_c! = credits'$

CandyMachine$_2$

$[\ldots]$

`enable_`*switch*

`effect_`*switch*

$\Delta(credits);\ tr_c? : \mathbb{N}$

$credits' = tr_c?$

As the sole cut operation *switch* only occurs once in the specification, no address parameters are required. We remain to apply the renaming of the channel *switch* and all of its occurrences within $S.\texttt{main}$, according to the introduction of the sole transmission parameter. The final decomposition is depicted in Figures 4.24 and 4.25.

When dealing with the identification of *reasonable* decompositions, Chapter 6 introduces a bigger case study, consisting of several classes and requiring address parameters as well as transmission parameters.

4.3.7 Improvement of the Decomposition

Up to now, we defined a valid decomposition of a specification, based on a fragmentation of its dependence graph. The given correctness criteria exclude invalid decompositions, thus restricting the set of possible decompositions.

In Section 4.3.1, we defined a restriction of the initial state schema of S on the possible initial valuations of the generated components S_1 and S_2. The implementation of our decomposition approach is based on this specific definition and needs to take any initial

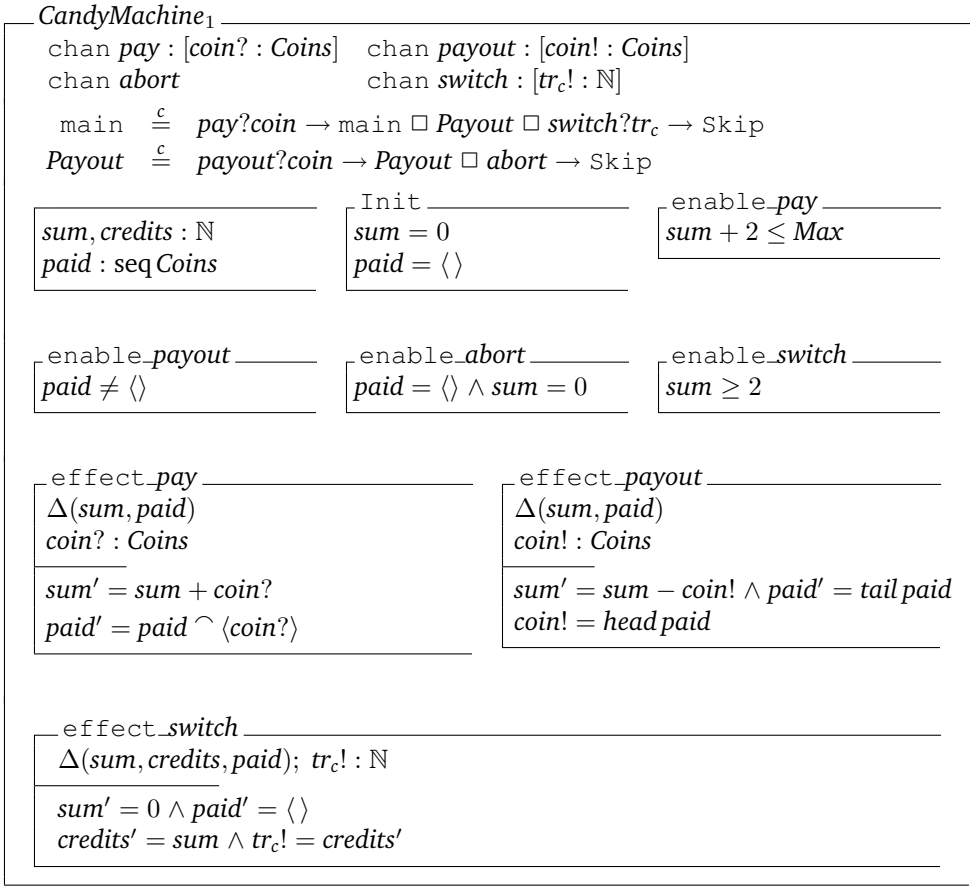


Figure 4.24: Decomposition of the candy machine, first component

data dependence into account. The definition can, however, slightly by altered and improved.

The specification *CandyMachine* comprises an initial state predicate $items = \langle\rangle$, which forms the source of three initial data dependences:

- init $\overset{\texttt{idd}}{\dashrightarrow}_{(items)}$ term, based on $\texttt{enable\_}term = [items = \langle\rangle]$,
- init $\overset{\texttt{idd}}{\dashrightarrow}_{(items)}$ deliver, based on (amongst others) $\texttt{enable\_}term = [items \neq \langle\rangle]$ and
- init $\overset{\texttt{idd}}{\dashrightarrow}_{(items)}$ order, based on $\texttt{effect\_}order$ comprising $item' = (items \frown \langle selected \rangle)$.

We identified {switch} as a valid single cut in Sections 4.2.4 and 4.3.6 due to the fact that all of these three initial data dependences do *not* violate the correctness criterion **no crossing**. The reason is as follows: the variable *items* is never modified or referenced in any operation schema of $Op_1 \cup Op_{\mathbf{C}}$. In particular, $items \notin S_1.V$. Therefore, *items*

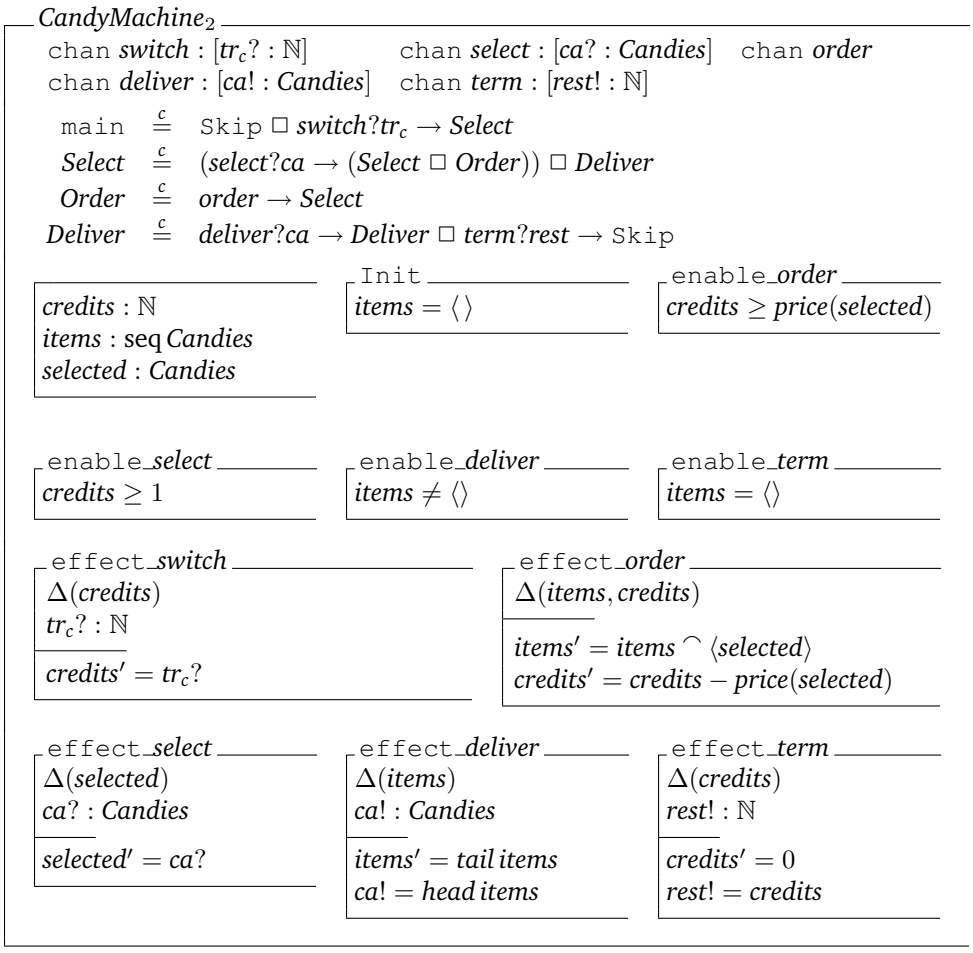


Figure 4.25: Decomposition of the candy machine, second component

does not influence the behaviour of S_1 at all. In this case, the respective initial state predicate can completely be eliminated from $S_1.\texttt{Init}$ and the corresponding initial data dependence can safely be neglected.

Indeed, this elimination of an initial data dependence is only possible for corresponding predicates not being related to the variables of S_1 at all. These observations serve as the basis for the following definitions.

First, when explicitly dealing with predicates within a CSP-OZ specification, we do not refer to the single top-level predicate of an operation but rather to its *atomic sub-predicates*. This is according to [Brü08]. For $op \in Op$, the set $Atoms(op)$ depicts the set of all *atomic predicates* such that the conjunction of all these predicates yields the predicate part of op. We use the same notation for the initial state schema:

$$\bigwedge_{p \in Atoms(op.pred)} p = op.pred \text{ and } \bigwedge_{p \in Atoms(\texttt{Init})} p = \texttt{Init}.$$

Next, we define an equivalence relation on $S.V$ and a *closure set* of a state variable with respect to this relation. Let $vars(p)$ denote the set of state variables, occurring in the predicate p:

Definition 4.3.26. *(Initial closure of state variables)*
Let S be a specification. We define an equivalence relation $\mathcal{R}$ over $(S.V \times S.V)$ by[5]

$$\mathcal{R} := \{(x,y) \mid \exists\, a \in Atoms(\texttt{Init}) \bullet x,y \in vars(a)\} \cup \mathsf{Id}_{S.V}.$$

For any $x \in S.V$, the initial closure *of x is inductively defined as the set $InitClos(x)$, satisfying the following two conditions:*

- $x \in InitClos(x)$,
- $y_1 \in InitClos(x) \wedge (y_1,y_2) \in \mathcal{R} \Rightarrow y_2 \in InitClos(x)$.

$\mathcal{R}$ relates any two state variables such that there exists an atomic predicate within `Init` containing both variables. The initial closure of a state variable x is the set of all state variables, directly or indirectly influencing x within the initial state schema.

Example 4.3.27. *Let S be a specification, $S.V = \{x,y,z\}$ with all elements of type $\mathbb{N}$, and let $S.\texttt{Init} = (x > 2) \wedge (x < y) \wedge (z > 5)$. Then, $\mathcal{R} = \{(x,y),(y,x)\} \cup \mathsf{Id}_{\{x,y,z\}}$. This yields $InitClos(z) = \{z\}$ and $InitClos(x) = InitClos(y) = \{x,y\}$.*
Now let $S.\texttt{Init} = (x = y) \wedge (y = z)$. Then, $\mathcal{R} = \{(x,y),(y,x),(y,z),(z,y)\} \cup \mathsf{Id}_{\{x,y,z\}}$. This yields $InitClos(x) = InitClos(y) = InitClos(z) = \{x,y,z\}$.

These considerations do not influence Definition 4.3.6. The correctness proof in Chapter 5 shows the following: we can safely neglect all initial data dependences originating from an atomic predicate a, such that $InitClos(x) \subseteq (S_2.V \setminus S_1.V)$ for all $x \in vars(a)$.

In our specific case, $InitClos(items) = \{items\}$ and $\{items\} \subseteq (S_2.V \setminus S_1.V)$ holds. Thus, the three previously identified initial data dependences originating from $items = \langle\,\rangle$ can indeed be neglected, justifying the correctness of the cut $\{\mathsf{switch}\}$. In particular, the predicate $items = \langle\,\rangle$ is already removed from $CandyMachine_2.\texttt{Init}$ by applying our definition for a decomposition and further simplifications on $CandyMachine_2.\texttt{Init}$.

We pointed out an optimisation for the decomposition in the following sense: some data dependences do not need to be considered when the correctness criterion **no crossing** is validated. Thus, a larger set of valid decompositions is possible.

4.4 Decomposition for the General Case: Number Swapper

We recall the small case study of a number swapper from Chapter 2 and slightly adapt it as displayed in Figure 4.26: the specification swaps two natural numbers a and b with a initially possessing the value 1 and b continuously receiving a new value as an input. The protocol starts by inputting the new value for b, subsequently swaps both numbers and outputs the new value of b. As $Swapper.\texttt{main}$ restarts, the specification does not allow for the definition of a single cut.

[5] For any set X, we let Id_X denote the identity on X, that is, $\mathsf{Id}_X := \{(x,x) \mid x \in X\}$.

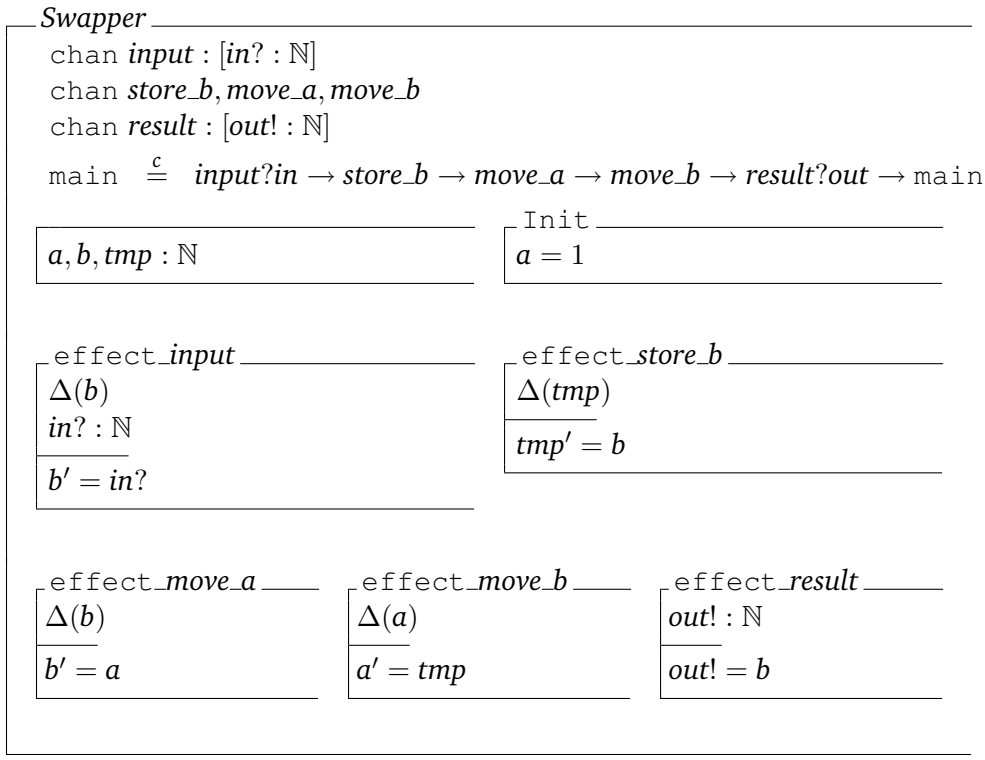


Figure 4.26: CSP-OZ specification for swapping two numbers, extended

A valid (general) cut for this specification is given by $(\mathbf{C}_1, \mathbf{C}_2)$ with $\mathbf{C}_1 = \{\mathsf{store\_b}\}$ and $\mathbf{C}_2 = \{\mathsf{result}\}$. The definition yields

- $Op_1 = \{input\}$,
- $Op_{\mathbf{C}_1} = \{store\_b\}$,
- $Op_2 = \{move\_a, move\_b\}$ and
- $Op_{\mathbf{C}_2} = \{result\}$.

For the sets of state variables, we get

- $S_1.V = \{b, tmp\}$ and
- $S_2.V = \{a, b, tmp\}$.

The initial state schemas are given by

$$\begin{array}{lcll} S_1.\texttt{Init} & = & \exists\, a : \mathbb{N} \bullet (a = 1) & \equiv \text{ true}, \\ S_2.\texttt{Init} & = & \exists\, b : \mathbb{N}, tmp : \mathbb{N} \bullet (a = 1) & \equiv \ a = 1. \end{array}$$

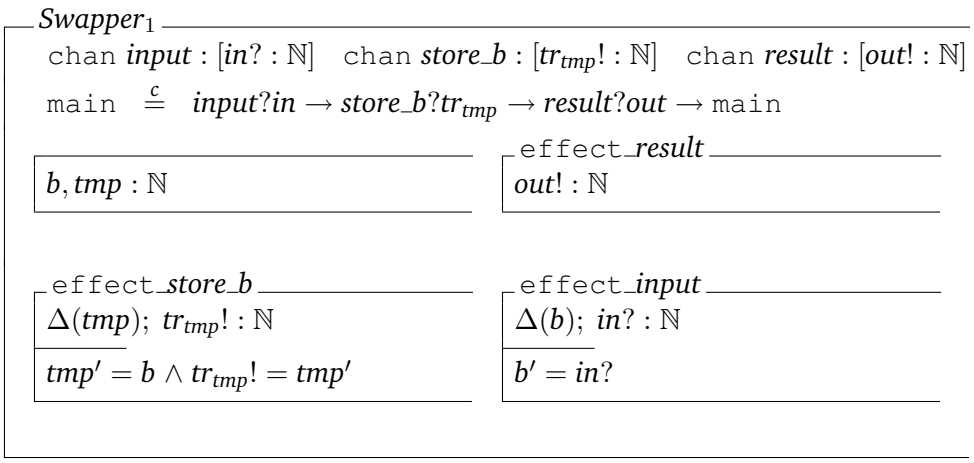


Figure 4.27: Decomposition of the number swapper, first component

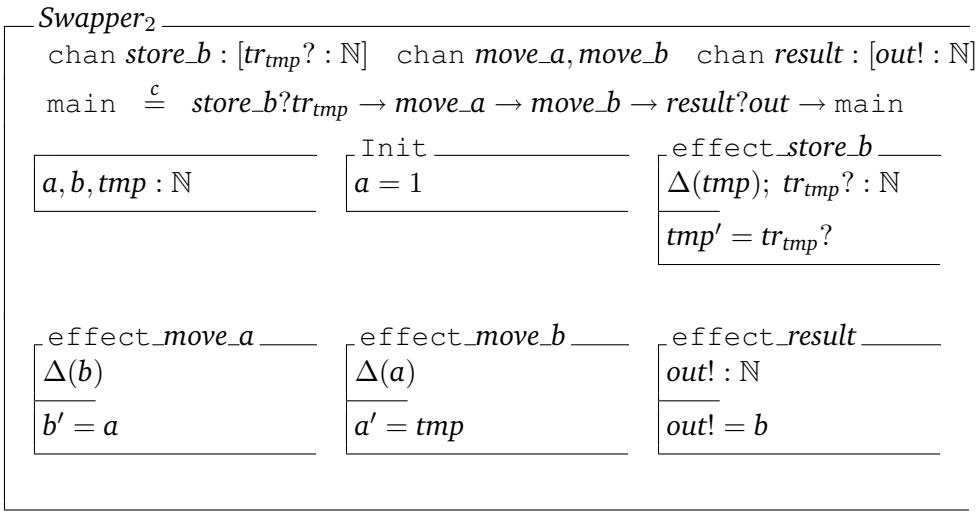


Figure 4.28: Decomposition of the number swapper, second component

As $InitClos(a) = \{a\}$ and $\{a\} \subseteq (S_2.V \setminus S_1.V)$, the initial data dependence init $\overset{\texttt{idd}}{\dashrightarrow}_{(a)}$ move_a is not cut-crossing. Based on the data dependence store_b $\overset{\texttt{dd}}{\dashrightarrow}_{(tmp)}$ move_b, we get $CV_1 = \{tmp\}$, necessitating one transmission parameter tr_{tmp}. As the operation *result* does not modify any state variable, $CV_2 = \varnothing$ holds. No addressing extension is required, thus leading to the final decomposition as given in Figures 4.27 and 4.28.

The correctness property on the specification as described in Figure 4.29 models that the value received by *input* corresponds to the output value of *result* in the next iteration of the protocol. Model checking this property with FDR2 yields its validity for *Swapper* as well as $Swapper_1 \parallel_{\{|store\_b,result|\}} Swapper_2$.

$$\begin{aligned} Prop &= \Box_{j \in \mathbb{N}}(input.j \rightarrow result.1 \rightarrow P(j)) \\ P(j) &= \Box_{k \in \mathbb{N}}(input.k \rightarrow result.j \rightarrow P(k)) \end{aligned}$$

Figure 4.29: Correctness requirement for *Swapper*

4.5 Related Work

The technique proposed in this chapter targets the manual decomposition of a given specification into two components. These subsystems are used in a compositional verification framework which is based on two assume-guarantee proof rules. The approach is closely related to several works, with some of them described next.

The dependence analysis, as given in Section 2.3, is based upon the methodology by Brückner [Brü08] for slicing CSP-OZ specifications. Besides applying a similar analysis of a specification, slicing does not decompose a given specification but rather eliminates irrelevant parts from it. These irrelevant specification elements depend on a certain property under interest, the slicing criterion. A correct decomposition in our context is independent of the verification properties. The decomposition approach is more closely related to program *chopping* [RR95]: chopping is likewise based on the analysis of a (program) dependence graph and tries to identify program points affecting a certain target node based on a specific source node.

Several works in the context of formal specifications present techniques for decomposing a given system into several components. Recently, Butler [But09] sketched a technique for composing Event-B models and decomposing them into sub-models. Here, events can be split, without allowing common variables to different machines. Similar to our approach dealing with transmission parameters, shared parameters are used to pass the influence of one to another machine. The technique is not applied in the context of compositional verification, but rather in the scope of model refinements. In the context of CSP||B and for separate checking of divergence freedom of a model, Evans, Schneider and Treharne [STE05] developed a methodology to decompose CSP||B specifications

into smaller subsystems, called *chunks*. They can consist of a set of CSP processes or contain B machines as well. The decomposition is conducted by examining the existing subsystems and parallel components of a CSP||B specification.

The technique closest to ours is the one by Alur and Nam [NA06, AMN05, Nam07] dealing with assume-guarantee-based reasoning in the context of symbolic model checking. Using symbolic transition systems (STS) as the semantic model, the authors fully automatically decompose and verify a system. A decomposition of a STS yields a set of symbolic *modules*, now comprising additional boolean input- and output variables which are similar to our transmission parameters. The choice of the decomposition is carried out by an automatic partitioning of the set of boolean variables of the STS and it is based on an equal distribution of the set of variables along with a minimisation of required inputs and outputs. The approach is also based on the L^* algorithm, using a generalised version of rule **(B-AGR)** in the validation process. In their semantic domain solely dealing with boolean variables, the authors do not incorporate a dependence analysis based on data flow and control flow, and they do not tackle communication and synchronisation aspects of a specification. The decomposition is based on one particular heuristic which does not take the alphabet size of the assumption into account. As the decomposition is performed automatically, it is impossible to lead the framework to a superior decomposition *by hand* which does not satisfy the constraints for an equal distribution and minimisation.

Another related work discusses the usefulness of assume-guarantee reasoning. The authors investigate the possible decompositions of a program specified as a labelled transition system, based on several case studies and model checkers [CAC06]. The results show that assume-guarantee reasoning outperforms monolithic verification in only a few cases. Two conclusions can be drawn from this work: assume-guarantee reasoning is not in general more effective than direct model checking. Moreover, its effectiveness highly depends on the choice of the decomposition. The authors state that analysts need some guidance to identify those decompositions which are indeed less time- and memory consuming. Chapter 6 will provide some theory on how this can be achieved.

Beforehand, the next chapter will show correctness of our approach by particularly proving Theorem 4.3.25.

5 Correctness of the Decomposition

Contents

5.1 Ensuring Correct Synchronisation 113
5.2 Correctness for the CSP Part 119
5.2.1 Properties of the Decomposition: CSP Part 119
5.2.2 Correctness of the Decomposition: CSP part 132
5.3 Correctness for the Object-Z Part 138
5.3.1 Properties of the Decomposition: Object-Z Part 140
5.3.2 Correctness of the Decomposition: Object-Z part 146
5.4 Correctness of the Renaming for the Decomposition 159
5.5 CSP Laws for Parallel Composition 164
5.6 Proof of the Main Theorem 166

The previous chapter introduced a technique on how to decompose a given CSP-OZ specification into two components, based on an analysis of the specification's dependence graph. Theorem 4.3.25 states the main result of this thesis: in our semantic domain of the CSP traces model, the original specification and its decomposition are *trace-equivalent*. The result is essential and ensures the following: for a property P, specified as a CSP process,

$$(P \sqsubseteq_T S) \Leftrightarrow (P' \sqsubseteq_T (S_1 \parallel_{E_C} S_2)).$$

Here, we need to refer to a process P', resulting from the process P after a renaming with respect to the set of all additional parameters. Recall that S_1 and S_2 already comprise transmission parameters and address parameters according to Section 4.3.4. $P' \sqsubseteq_T (S_1 \parallel S_2)$ can be deduced from $A \sqsubseteq_T S_1$ and $P' \sqsubseteq_T (A \parallel S_2)$ within the compositional learning framework, introduced in Chapter 3. Thus, correctness of the compositional proof rules **(B-AGR)** and **(P-AGR)** along with Theorem 4.3.25 yield the overall correctness of our approach.

Correctness of **(B-AGR)** and **(P-AGR)** were already shown in Chapter 3. The verification of Theorem 4.3.25 will be carried out in the present chapter. The main strategy for the proof uses the compositional semantics of CSP-OZ specifications in terms of CSP_Z according to Figure 2.7: the traces of a CSP-OZ specification S are given by

$$\mathit{traces}(S.\texttt{main} \|_{E_S} S.OZ).$$

Figure 5.1 illustrates the individual proof steps. Precisely, we show:

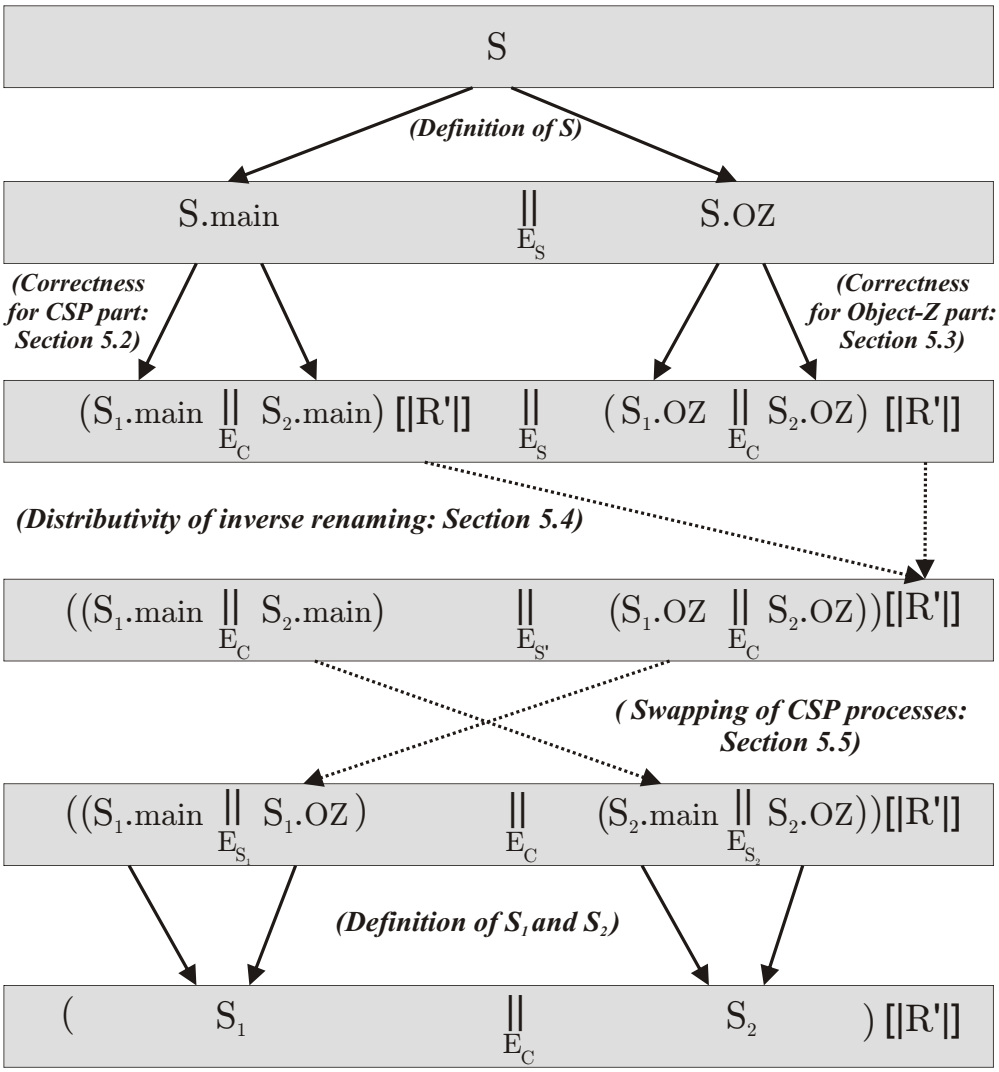


Figure 5.1: Illustration of the steps of the correctness proof

Correctness for the CSP Part, Section 5.2: Based on the compositional semantics of CSP-OZ, $S =_T (S.\texttt{main} \parallel_{E_S} S.OZ)$ holds. In order to refer to the individual parts of the components S_i, we first need to decompose the CSP part and show that the original CSP part is trace equivalent to its decomposition modulo the (inverse) renaming relation, that is,

$$S.\texttt{main} =_T (S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main})[\![R']\!].$$

Correctness for the OZ Part, Section 5.3: Accordingly, we have to show correctness for

the decomposition of the Object-Z part. However,

$$S.OZ =_T (S_1.OZ \parallel_{E_C} S_2.OZ)[\![R']\!]$$

does not hold in general: traces of the Object-Z part alone do not adhere to the CSP part. We need to take the orderings of events with respect to the CSP part into account and show $S.OZ =_T (S_1.OZ \parallel_{E_C} S_2.OZ)[\![R']\!]$ *for the set of traces conforming to the CSP part*.

Distributivity of Inverse Renaming, Section 5.4: After showing the individual correctness of both decompositions, we have to distribute the inverse renaming relation R' over the parallel composition E_S. Thus, we show

$$(S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main})[\![R']\!] \parallel_{E_S} (S_1.OZ \parallel_{E_C} S_2.OZ)[\![R']\!] =_T$$
$$((S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_C} S_2.OZ))[\![R']\!].$$

Redistribution of CSP Processes, Section 5.5: Now being able to refer to $S_i.\texttt{main}$ and $S_i.OZ$ without the need for considering the renaming, we have to *swap* $S_2.\texttt{main}$ and $S_1.OZ$ to step from the parallel composition of the CSP parts and Object-Z parts to the parallel composition of the components S_1 and S_2. We show

$$(S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_C} S_2.OZ) =_T$$
$$(S_1.\texttt{main} \parallel_{E_{S_1}} S_1.OZ) \parallel_{E_C} (S_2.\texttt{main} \parallel_{E_{S_2}} S_2.OZ),$$

which subsequently leads to the overall conclusion, as $S_i =_T (S_i.\texttt{main} \parallel_{E_{S_i}} S_i.OZ)$ holds.

Before getting under way with the individual proof steps, Section 5.1 presents an algorithm, which satisfies the requirements for correct addressing. Afterwards, we show correctness of the decomposition of the CSP part and the Object-Z part in Sections 5.2 and 5.3, respectively. The correctness for the distributivity of the inverse renaming is given in Section 5.4, followed by a lemma, stating the possible redistribution of CSP processes within a context-specific parallel composition in Section 5.5. The chapter concludes with the proof of Theorem 4.3.25, now joining together all the individual proof steps.

5.1 Ensuring Correct Synchronisation

In order to ensure an equivalent control flow of the original specification and its decomposition, Section 4.3.3 introduced the concept of address parameters. In particular, Definition 4.3.12 presented two conditions on these additional parameters and Theorem 4.3.16 showed that they are sufficient to preserve the original control flow.

In this section, we define an algorithm, realising both conditions of Definition 4.3.12. The algorithm was successfully implemented in Java as part of a diploma thesis [Her09] focusing on the integration of the decomposition approach into Syspect [Sys06], a graphical modelling environment for CSP-OZ. In this thesis, the algorithm will be presented in *pseudo code*.

The root procedure inputs the control flow graph of a specification S along with the set of operations $Op' := Op_1 \cup Op_2 \cup Op_{\mathbf{C}}$. It computes the modified interfaces $S_i.I$ and CSP processes $S_i.\texttt{main}$, according to Definition 4.3.22 and Definition 4.3.23, respectively.

For $\texttt{CFG}_S = (N, \longrightarrow)$ being the CFG of the specification under interest, let n denote an arbitrary node of the CFG and, in case that n introduces branching, unn its corresponding join node. For any $op \in Op$, we do not denote the original type, but solely refer to the additional address parameters.

procedure ADDRESSMAIN($\texttt{CFG}_S, Op'$)
for each $(op \in (Op_1 \cap Op_2) \setminus Op_{\mathbf{C}})$
do
- $op \leftarrow op : [p_1 : \{1,2\}]$ **for the definition of** $S_i.I$
- **if** $(\mathsf{op}^i \in (\mathbf{Ph}_1 \cup \mathbf{Ph}_3))$ **do** $op_p^i \leftarrow op_p^i.1$ **for the definition of** $S_1.\texttt{main}$
- **if** $(\mathsf{op}^j \in \mathbf{Ph}_2)$ **do** $op_p^j \leftarrow op_p^j.2$ **for the definition of** $S_2.\texttt{main}$

for each $(op \in Op_{\mathbf{C}_1}$ such that $l^{-1}(op) > 1)$ **do** ADDRESSCUT$(op, \mathbf{C}_1)$
for each $(op \in Op_{\mathbf{C}_2}$ such that $l^{-1}(op) > 1)$ **do** ADDRESSCUT$(op, \mathbf{C}_2)$

Figure 5.2: Algorithm for the address extension: procedure ADDRESSMAIN

The algorithm comprises four different procedures. The root procedure ADDRESSMAIN is given in Figure 5.2. It first processes over all shared operations of S_1 and S_2, which are not located in a cut set. Their corresponding occurrences need to be separated, and they are addressed by one parameter, according to Section 4.3.3.

procedure ADDRESSCUT$(op, \mathbf{C}_i)$
global $Decl(op) \leftarrow \{p_1 : \{1\}\}$
global $Val(op) \leftarrow \varnothing$
comment: ADD modifies $Decl(op)$ and $Val(op)$
ADD$(\mathsf{start}, op, \langle p_1 = 1\rangle, \mathbf{C}_i, \mathsf{false})$
MODIFYCUT$(op, Decl(op), Val(op))$

Figure 5.3: Algorithm for the address extension: procedure ADDRESSCUT

The procedure ADDRESSCUT, depicted in Figure 5.3, is successively called for all operations op with multiple occurrence in either $\mathbf{C}_1$ or $\mathbf{C}_2$. For each operation, ADDRESSCUT holds two global lists:

- $Decl(op)$ comprises the set of additional address parameters of op with their corresponding types. Initially, $Decl(op)$ holds one parameter of type $\{1\}$. The set is used

for the definition of the interfaces $S_i.I$.

- $Val(op)$ contains a set of tuples $(op_p^j, values)$, where *values* is a sequence of valuations $p_i = v_i$ with v_i, denoting the restriction of the address parameter of p_i for the specific occurrence op_p^j of op within $S.\texttt{main}$. Initially, the set is empty. $Val(op)$ is used to define the renaming of $S_i.\texttt{main}$.

ADDRESSCUT calls the core procedure ADD, which recursively traverses the CFG and modifies the global variables $Decl(op)$ and $Val(op)$. This procedure will be explained below.

procedure MODIFYCUT$(op, Decl(op), Val(op))$
let $\{p_1 : t_1, \ldots, p_k : t_k\} = Decl(op)$ **in**
$op \leftarrow op : [p_1 : t_1;\ \ldots;\ p_k : t_k]$ **for the definition of** $S_1.I$ and $S_2.I$
for each $(op_p^j, values) \in Val(op)$ **do**
let (**if** $(\langle p_i = w_i \rangle$ in $values)$ **then** $v_i = w_i$ **else** $v_i = ?p_i)$ **in**
$op_p^j \leftarrow op_p^j.v_1.\ldots.v_k$ **for the definition of** $S_1.\texttt{main}$ and $S_2.\texttt{main}$

Figure 5.4: Algorithm for the address extension: procedure MODIFYCUT

After the procedure ADD terminated, ADDRESSCUT calls a procedure MODIFYCUT (Figure 5.4) which inputs the respective operation and both sets, $Decl(op)$ and $Val(op)$. MODIFYCUT carries out the actual modification of $S.I$ and the renaming of $S.\texttt{main}$, according to the results of ADD. In particular, the interfaces $S_i.I$ are modified based on the parameter declarations within $Decl(op)$. Each occurrence op_p^j of op within $S.\texttt{main}$ is modified with respect to the tuple $(op_p^j, values)$. Here, address parameters p_i are either restricted by $p_i = w_i$ or remain unrestricted, if *values* does not comprise a restriction on p_i.

Finally, the core procedure ADD, as shown in Figure 5.5, proceeds as follows. It traverses the CFG and inputs five parameters.

- The first parameter n denotes the current node visited by the procedure. For the initial call of ADD, this node is the unique start-node of the CFG.
- The second parameter *op* denotes the operation under interest.
- As ADD keeps track of all parameter valuations a subsequent occurrence of *op* needs to adhere to, the third parameter comprises the current restrictions for the address parameters. Initially, according to the singleton of initial parameters, $values = \langle p_1 = 1 \rangle$. Corresponding to the explanations of Section 4.3.3, the last element of *values* denotes the currently *active* parameter and its actual restriction.
- Parameter four identifies the cut set, corresponding to the occurrence of *op*.

procedure ADD(n, op, *values*, **CS**, *cutVisited*)

case $(\mathsf{n} = \mathsf{term}^i.X \wedge \mathsf{succ}(\mathsf{term}^i.X) = \varnothing) \vee \mathsf{n} = \mathsf{stop}^i$ **then exit**

case $\mathsf{n} \in \{\mathsf{skip}^i, \mathsf{seq}^i, \mathsf{call}^i.X, \mathsf{ret}.X\} \vee (\mathsf{n} = \mathsf{term}^i.X \wedge \mathsf{succ}(\mathsf{term}^i.X) \neq \varnothing)$
then ADD(succ(n), op, *values*, **CS**, *cutVisited*)

case $\mathsf{n} = \mathsf{start}.P$ **then**
if (start.P already visited) **then exit**
else ADD(succ(n), op, *values*, **CS**, *cutVisited*)

case $\mathsf{n} \in \mathsf{op}(N)$
then
- **if** $(\mathsf{n} \notin$ **CS and** *cutVisited* $=$ false$)$
  then ADD(succ(n), op, *values*, **CS**, *cutVisited*)
- **if** $(\mathsf{n} \notin$ **CS and** *cutVisited* $=$ true$)$ **then exit**
- **if** $(\mathsf{n} \in$ **CS**$)$
  then
  - **if** $(\nexists i \bullet \mathsf{n} = \mathsf{op}^i)$
    then ADD(succ(n), op, *values*, **CS**, true)
    else
    - $Val(op) \leftarrow Val(op) \cup \{(op_p^i, values)\}$
    - ADD(succ(n), op, *values*, **CS**, true)

case $((\mathsf{n} = \mathsf{par}_X^i \wedge op \notin X) \vee (\mathsf{n} \in \{\mathsf{extch}^i, \mathsf{intch}^i, \mathsf{interleave}^i\}))$
then
- **if** $($**CS** $= \mathbf{C}_c \wedge \mathsf{unn}^i$ is located behind or inside of $\mathbf{C}_c)$
  then let *last values* $= \langle p_k = j \rangle$ **in**
  - $Decl(op) \leftarrow (Decl(op) \setminus \{p_k : \{1, \ldots, l\}\} \cup \{p_k : \{1, \ldots, l+1\}\})$
  - **comment:** Note that $j \leq l$ but not necessarily $j = l$.
  - ADD(succ_one(n), op, *values*, **CS**, *cutVisited*)
  - ADD(succ_two(n), op, $(front\ values) \frown \langle p_k = l+1 \rangle$, **CS**, *cutVisited*)
  else ADD(unn^i, op, *values*, **CS**, *cutVisited*)

case $(\mathsf{n} = \mathsf{par}_X^i \wedge op \in X)$
then
- **if** $($**CS** $= \mathbf{C}_c \wedge \mathsf{unpar}_X^i$ is located behind or inside of $\mathbf{C}_c)$
  then let $\{p_1 : t_1, \ldots, p_k : t_k\} = Decl(op)$ **in**
  - $Decl(op) \leftarrow Decl(op) \cup \{p_{k+1} : \{1\}, p_{k+2} : \{1\}\}$
  - ADD(succ_one(n), op, $values \frown \langle p_{k+1} = 1 \rangle$, **CS**, *cutVisited*)
  - ADD(succ_two(n), op, $values \frown \langle p_{k+2} = 1 \rangle$, **CS**, *cutVisited*)
  else ADD(unpar^i, op, *values*, **CS**, *cutVisited*)

case $(\mathsf{n} = \mathsf{uncfop}^i)$
then
- **if** (uncfop^i already visited) **then exit**
  else ADD(succ(n), op, *values*, **CS**, *cutVisited*)

Figure 5.5: Algorithm for the address extension: procedure ADD

- Finally, a fifth parameter, initially assigned to *false*, specifies if ADD has already reached the respective cut set.

The general idea of ADD is to carry over and realise the requirements on a correct addressing from Definition 4.3.12. The procedure recursively traverses the CFG. As already explained, ADD has side effects on the global variables $Decl(op)$ and $Val(op)$: it continuously adds address parameters (in case of parallel composition with *op* being synchronised) and modifies their values (in case of any other branching).

Precisely, in case of n not having any successor node, the procedure stops. If n is an element of $\{\mathsf{skip}^i, \mathsf{seq}^i, \mathsf{call}^i.X, \mathsf{ret}.X, \mathsf{term}^i.X\}$ with the latter node being followed by a ret-node, the procedure is recursively called for the sole successor node.

Termination of the algorithm is achieved by the fact that $\mathsf{n} = \mathsf{start}.P$ only leads to a recursive call if $\mathsf{start}.P$ was not already visited before. Otherwise, the respective call of ADD terminates. Note that for simplification, our *pseudo code*-algorithm does not explicitly keep track of the already visited nodes. This can obviously be achieved by adding a global variable.

Next, if n is an operation node of the CFG, a case differentiation is required: if n does not correspond to an operation of the cut set, the procedure either continues (if the traversal did not reach the cut set yet) or terminates (in the opposite case, as this signalises that the cut set is left). Accordingly, if n does not represent an occurrence of the operation under interest, the procedure is called for its successor node. In the final case of $\mathsf{n} = \mathsf{op}^i$ for some i, the current tuple $(op_p^i, values)$ is stored in $Val(op)$. This assignment signalises the modification of op_p^i within the procedure MODIFYCUT.

The first core case of the procedure handles the case of branching without synchronisation of *op*. Here, the *type* of the currently active parameter p_k is modified according to the first case of Definition 4.3.12. Precisely, it is extended by one additional value within $Decl(op)$. Additionally, the restriction of p_k within one branch is preserved, whereas it is assigned with the new value within the other branch. An additional **if**-clause ensures that the branching indeed reaches the cut set under interest and does not terminate beforehand. Otherwise, the procedure simply steps over the branching.

In the second core case, the algorithm deals with parallel composition with *op* being synchronised. According to the second case of Definition 4.3.12, we introduce two additional address parameters, with one of them restricted for the first and the other one for the second branch. This is carried out in this specific case of the procedure: $Decl(op)$ is extended by two additional parameters of initial type $\{1\}$, whereas $Val(op)$ is extended by two additional tuples, denoting the initial restrictions for the first and second branch, respectively. The procedure is recursively called for both branches. Again we use an **if**-clause to prevent proceeding of branching, terminating before the cut set.

The final case deals with joining of branching. Here, we again simply proceed with the node's successor. However, as a join node has two incoming edges, we need to ensure that we only proceed once with the node's sole successor.

We will now substantiate the termination and correctness of the algorithm.

Proposition 5.1.1. *(Termination of* ADDRESSMAIN*)*
The algorithm ADDRESSMAIN *terminates for any control flow graph.*

Proof (Sketch). Obviously, we solely need to show termination of ADD. Let π be a path of the CFG of an arbitrary specification. We distinguish two cases:

1. π is a finite path. In accordance to the definition of the CFG, the final node of the path is either $\mathsf{term}^i.X$ or stop^i. However, in both cases, ADD terminates, according to the first case.

2. π is an infinite path. Thus, the path must contain a cycle,, since the CFG's set of nodes is finite. According to the definition of the CFG and our explanations from Section 2.3.2, the sole possibility for cycles are combinations of $\mathsf{call}.P^i$ and $\mathsf{start}.P$ for some process P. However, in case the algorithm visits $\mathsf{start}.P$ for the second time, it again terminates. □

Proposition 5.1.2. *(Correctness of* ADDRESSMAIN*)*
The algorithm ADDRESSMAIN *satisfies both conditions of Definition 4.3.12.*

Proof (Sketch). We recall both conditions from Definition 4.3.12.

Branching without Synchronisation: *If* op^i *and* op^j *are located inside different branches of either an external choice operator, internal choice operator, interleaving operator or a parallel composition operator* par_X *with* $op \notin X$*,* op_p *needs to comprise one parameter* p_1*, such that its type includes* $x, y \in \mathbb{N}$ *with* $x \neq y$*. This parameter is fixed to* x *for* op_p^i *and to* y *for* op_p^j *in both,* $S_1.\texttt{main}$ *and* $S_2.\texttt{main}$:

op_p^i *becomes* $op_p^i.x$ *and* op_p^j *becomes* $op_p^j.y$.

Let op^i and op^j be two according nodes. Consider the first of the two core cases of ADD: it applies for op^i and op^j and thus, op_p^i and op_p^j will be addressed according to this case. The addressing sets the value of the parameter p_k to the value l in one and to the value $l+1$ in the other branch. Any further branching preserves the inequality of both values. Thus, p_k satisfies the first condition of the definition.

Branching with Synchronisation: *If* $\mathsf{op}^i \overset{\mathtt{sd}}{\dashleftarrow\!\dashrightarrow} \mathsf{op}^j$*, the (partial) event* op_p *needs to comprise two parameters* p_1 *and* p_2*, such that the type of* p_1 *includes* $x \in \mathbb{N}$ *and the type of* p_2 *includes* $y \in \mathbb{N}$ *for arbitrary* x, y*. The first parameter is fixed to* x *for* op_p^i*, whereas the second parameter is fixed to* y *for* op_p^j *in both,* $S_1.\texttt{main}$ *and* $S_2.\texttt{main}$:

op_p^i *becomes* $op_p^i.x?p_2$ *and* op_p^j *becomes* $op_p^j?p_1.y$.

Let op^i and op^j be two according nodes. Consider the second of the two core cases of ADD: op_p^i and op_p^j will be addressed according to this case as both nodes are located in different branches of a parallel composition par_X^i with $op \in X$. Two additional parameters p_{k+1}, p_{k+2} are introduced, with one of them fixed for op_p^i and the other one fixed for op_p^j (initially by 1 and possibly modified later on). Therefore, the second condition of the definition is satisfied as well. □

This completes the definition of the algorithm and the motivation for its termination and correctness. The following sections carry out the individual steps of the proof of Theorem 4.3.25.

5.2 Correctness for the CSP Part

The operational semantics of CSP-OZ allows for a compositional proof of Theorem 4.3.25 in the following sense: we show that the individual decompositions of the CSP part $S.\texttt{main}$ and the Object-Z part $S.OZ$ are semantics-preserving in the domain of the CSP traces model. Subsequently, and by using some additional properties, we combine both results to deduce the overall correctness of the decomposition.

For both, the CSP part and the Object-Z part, we will show semantic equivalence *modulo renaming*. This means, that we relate the original events from E_S to events from $E_{S'}$, now possibly comprising transmission parameters and address parameters.

In this section, the correctness proof of the CSP part is conducted. We have to show

$$S.\texttt{main} =_T (S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main})[\![\mathrm{R}']\!],$$

that is, the proof will show the equivalence of both CSP processes, factoring out the different parameter extensions.[1] At first sight, this particular proof step seems to be rather easy, as the CSP process $S.\texttt{main}$ is one-to-one reflected in the CFG of a specification.

However, as a first obstacle, the set of traces of $S.\texttt{main}$ does not correspond to the set of paths of the CFG: in general, the first set is strictly larger due to possible interleaving. This complicates the proof, as reasoning with respect to the specification's CFG becomes impractical.

Another difficulty arises from the *projection* of CSP processes and traces according to Definitions 4.3.3 and 2.2.8, respectively. Unfortunately, their definitions do *not* satisfy the law

$$\mathit{traces}(P|_X) = \mathit{traces}(P) \upharpoonright X$$

when we are dealing with parallel composition of processes. As we need to bridge the gap between both definitions, this particularly complicates dealing with this individual operator.

Before carrying out the actual correctness proof of the decomposition of the CSP part, we start with some related properties.

5.2.1 Properties of the Decomposition: CSP Part

Showing correctness of the decomposition of the CSP part requires several properties, which the actual correctness proof uses. They are given next.

Disallowed Distribution of Initial Branching Events

A first property describes that in case of branching within the CFG, the *initial events* (see Definition 2.2.9) of both branches are never distributed over E_1 and E_2, that is, over the sets of local events for the components:

[1] We explicitly deal with the renaming relation in Section 5.4.

Lemma 5.2.1. *(No distribution of initial events)*
For $\circ \in \{\Box, \sqcap, \|_S, \ |||\ \}$*, let* $P = Q_1 \circ Q_2$ *be a process, occurring within* $S.\texttt{main}$*. Then, for any valid decomposition of S according to Definition 4.3.24:*

$$(\mathit{initials}(Q_1) \cap E_i) \neq \varnothing \Rightarrow (\mathit{initials}(Q_2) \cap E_j) = \varnothing,$$

for $i \neq j$*, and vice versa.*

Proof. Without loss of generality, assume that $e_1 \in (\mathit{initials}(Q_1) \cap E_1)$ and that there exists some $e_2 \in (\mathit{initials}(Q_2) \cap E_2)$, yielding $\mathsf{e}_2 \in \mathbf{Ph}_2$ for the corresponding DG node. Based on the definition for $\mathbf{Ph}_1$ and $e_1 \in E_1$, there exists a path π, such that

$$\mathsf{start} \xrightarrow{\pi} \mathsf{e}_1$$

and $\pi \cap \mathbf{C}_1 = \varnothing$. Obviously, any prefix of this path does not intersect with $\mathbf{C}_1$ as well. Let π' denote the prefix, leading from start to the binary operator $\circ$. As $e_2 \in \mathit{initials}(Q_2)$, the path π' can be extended to a path π_2 not comprising any additional nodes from $\mathtt{op}(N)$:

$$\mathsf{start} \xrightarrow{\pi_2} \mathsf{e}_2 \text{ corresponding to } \mathsf{start} \xrightarrow{\pi'} \mathsf{cfop} \xrightarrow{\bullet} \mathsf{e}_2,$$

and $\pi_2 \cap \mathbf{C}_1 = \varnothing$. We deduce that $\mathsf{e}_2 \in (\mathbf{Ph}_1 \cap \mathbf{Ph}_2)$, contradicting the correctness criterion **disjointness** for a valid cut. □

The lemma basically states that a branching introduced within a certain phase yields that *all* initial events of both branches are represented in this specific phase or the subsequent cut set. Figure 5.6 shows one instance of a *disallowed* distribution of initial events. Here, $\circ = \mathsf{extch}$, and the violation occurs with respect to the first cut set.

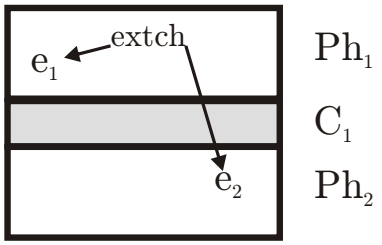


Figure 5.6: Illustration of a violation of Lemma 5.2.1

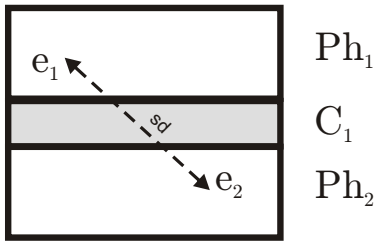


Figure 5.7: Illustration of a violation of Lemma 5.2.2

Disallowed Split of Synchronisation

A rather obvious property is the following: in case that two operation nodes are connected by a synchronisation dependence, they must not be distributed over different elements of $\{(\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \mathbf{C}_1, \mathbf{Ph}_2, \mathbf{C}_2\}$:

Lemma 5.2.2. *(No split of synchronisation)*
Let $\texttt{CFG}_S = (N, \longrightarrow)$ *be the control flow graph of a specification S, and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. Let* $\mathcal{D} = \{(\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \mathbf{C}_1, \mathbf{Ph}_2, \mathbf{C}_2\}$*. Then:*

$$\mathsf{n} \overset{\texttt{sd}}{\longleftrightarrow} \mathsf{n'} \Rightarrow (\forall M \in \mathcal{D} \bullet \mathsf{n} \in M \Leftrightarrow \mathsf{n'} \in M)$$

Proof. As n and n' are connected by a synchronisation dependence, both nodes have the same operation name: $l^{-1}(\mathsf{n}) = l^{-1}(\mathsf{n'})$ holds. The correctness criterion **all-or-none** rules out that both nodes are either distributed over both cut sets or over one cut set and one phase. We remain to show that both nodes must not be distributed over $(\mathbf{Ph}_1 \cup \mathbf{Ph}_3)$ and $\mathbf{Ph}_2$. However, if this was the case, the connecting synchronisation dependence would violate **no crossing**. □

A possible violation of the lemma, with two synchronised nodes distributed over $\mathbf{Ph}_1$ and $\mathbf{Ph}_2$, is illustrated in Figure 5.7.

Redistribution of Processes for Binary Operators

Next, we show a property which we will use throughout the actual proof. The property states that we can *redistribute* the component processes with respect to the parallel composition $\|_{E_C}$ and all *binary* operators, that is, $\circ \in \{\Box, \sqcap, \overset{\circ}{,}, \|_S, \,|||\,\}$. Precisely, for $P = (T \circ U)$, $T_i = T|_{E_{S_i}}$ and $U_i = U|_{E_{S_i}}$, Figure 5.8 illustrates our proof strategy for $S.\texttt{main}$ being composed of two processes. The top-down-equivalence will be shown in the following lemma.

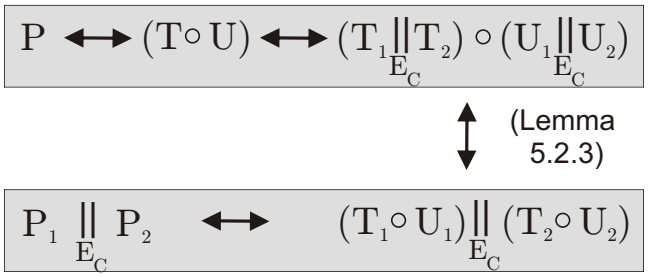


Figure 5.8: Illustration of the CSP correctness proof of binary operators

The subsequent lemma uses the LTS semantics of CSP according to Definition 2.2.11. Here, we will refer to the firing rules or CSP [Ros98], which determine the set of possible transitions and thus the labelled transition system of a process. In addition to the set of possible events a process may perform, we need to deal with τ-transitions, symbolising *invisible* events.

In our semantic domain of the CSP traces model, we apply several simplifications: first of all and according to [Brü08], we do not distinguish between the processes `Stop` and `Div`. This is justified by the fact that both processes are incapable of performing any event, and they are thus trace equivalent. Furthermore, we do not consider the special

event ✓, signalising termination of a process. As a consequence, we do not separate the processes `Skip` and `Stop`. The LTS semantics makes use of an additional symbol Ω, denoting the *end state* of the transition system. Here, we do not separately deal with this symbol and rather refer to the corresponding process `Stop`. These considerations allow us to simplify and restrict some of the CSP firing rules.

Lemma 5.2.3. *(Redistribution of CSP processes within the decomposition)*
Let $P = (T \circ U)$ for $\circ \in \{\Box, \sqcap, \overset{\circ}{,}, \;|||\;, \|_S\}$ be a reachable state of the LTS of S.`main`. Then,

$$(T_1 \;\|_{E_C}\; T_2) \circ (U_1 \;\|_{E_C}\; U_2) =_T (T_1 \circ U_1) \;\|_{E_C}\; (T_2 \circ U_2)$$

for $\circ \neq \|_S$ and

$$(T_1 \;\|_{E_C}\; T_2) \;\|_S\; (U_1 \;\|_{E_C}\; U_2) =_T (T_1 \;\|_{S \cap E_{S_1}}\; U_1) \;\|_{E_C}\; (T_2 \;\|_{S \cap E_{S_2}}\; U_2),$$

where $T_i = T|_{E_{S_i}}$ and $U_i = U|_{E_{S_i}}$, $i \in \{1, 2\}$.

Proof. As we are interested in trace equivalence, external choice and internal choice can equally be treated. Moreover, being a special case of parallel composition with an empty synchronisation alphabet, we do not explicitly need to deal with interleaving.

The method of proof, which we choose here, is (weak) *bisimilarity* [Mil89]: if we can show that the labelled transition systems of the left hand side and the right hand side of the equation are bisimilar, we can deduce their trace equivalence [Pnu85]. In the following, let *op.x.t.a* indicate an event of $E_{S'}$ with the valuations for

- the original parameters according to x,
- the transmission parameters according to t and
- the address parameters according to a.

Subject to the individual operator we refer to, we define a weak bisimulation

$$\mathcal{R} = \{(A, B) \mid A = (C_1 \;\|_{E_C}\; C_2) \circ (D_1 \;\|_{E_C}\; D_2), B = (C_1 \circ D_1) \;\|_{E_C}\; (C_2 \circ D_2)\} \cup \mathcal{R}',$$

and we show that $(T_1 \;\|_{E_C}\; T_2) \circ (U_1 \;\|_{E_C}\; U_2)$ and $(T_1 \circ U_1) \;\|_{E_C}\; (T_2 \circ U_2)$ are the initial states of $\mathcal{R}$. Here, $C_i \in L^{CSP}$ ($D_i \in L^{CSP}$) denotes any reachable state within the labelled transition system of T_i (U_i), and $\mathcal{R}'$ denotes a case-specific extension of $\mathcal{R}$.

Based on the definition of bisimulation, we have to show two directions:

(1) If $(A, B) \in \mathcal{R}$ and $A \xrightarrow{e} A'$ for $e \in E_{S'} \cup \{\tau\}$, then there exists some B' such that $B \xrightarrow{\widehat{e}} B'$ and $(A', B') \in \mathcal{R}$.

(2) If $(A, B) \in \mathcal{R}$ and $B \xrightarrow{e} B'$ for $e \in E_{S'} \cup \{\tau\}$, then there exists some A' such that $A \xrightarrow{\widehat{e}} A'$ and $(A', B') \in \mathcal{R}$.

Here, we let $P \xrightarrow{\widehat{e}} P'$ stand for

$$P \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_k \xrightarrow{e} P_{k+1} \xrightarrow{\tau} \ldots \xrightarrow{\tau} P',$$

that is, P transits into P' by e, possibly surrounded by additional τ-transitions. We show the property for $\circ \in \{\Box, \fatsemi, \|_S\}$ and we construct the bisimulation relations $\mathcal{R}$ with respect to the individual binary operator, instantiating $\circ$. In any case, we separate between both required conditions (1) and (2). Within the individual proofs, we additionally distinguish between $(A,B) \in \mathcal{R}'$ and $(A,B) \notin \mathcal{R}'$, and we need to consider τ-transitions.

External Choice: For the case of external choice, we extend the relation $\mathcal{R}$ by defining

$$\begin{aligned}\mathcal{R}' := \mathsf{Id}_{L^{CSP}} \cup \{(A,B) \mid A = (C_1 \parallel_{E_C} C_2), B \in X_1\} \cup \\ \{(A,B) \mid A = (D_1 \parallel_{E_C} D_2), B \in X_2\},\end{aligned}$$

for

$$\begin{aligned}X_1 := \{(C_1 \parallel_{E_C} (C_2 \Box D_2)) \mid \mathit{initials}(D_2) \subseteq E_C\} \cup \\ \{((C_1 \Box D_1) \parallel_{E_C} C_2) \mid \mathit{initials}(D_1) \subseteq E_C\},\end{aligned}$$

and

$$\begin{aligned}X_2 := \{(D_1 \parallel_{E_C} (C_2 \Box D_2)) \mid \mathit{initials}(C_2) \subseteq E_C\} \cup \\ \{((C_1 \Box D_1) \parallel_{E_C} D_2) \mid \mathit{initials}(C_1) \subseteq E_C\},\end{aligned}$$

where again, $C_i \in L^{CSP}$ ($D_i \in L^{CSP}$) denotes any reachable state within the labelled transition system of T_i (U_i). Here, $\mathsf{Id}_{L^{CSP}} := \{(P,P) \mid P \in L^{CSP}\}$, depicting the identity on L^{CSP}. We do not explicitly deal with $(A,B) \in \mathsf{Id}_{L^{CSP}}$, as in this case, the bisimulation diagram can trivially be completed.

(1) $(A,B) \in \mathcal{R}'$ and $A \xrightarrow{e} A'$.

τ-case: Let $A \xrightarrow{\tau} A'$. We start with the case of $(A,B) \in \mathcal{R}'$. If

$$(C_1 \parallel_{E_C} C_2) \xrightarrow{\tau} (C_1' \parallel_{E_C} C_2'),$$

according to the firing rules for parallel composition, the transition is either performed by C_1 or C_2. We consider the first case, the other case is analogous. Then, $C_2' = C_2$. From $C_1 \xrightarrow{\tau} C_1'$, we get

$$C_1 \parallel_{E_C} (C_2 \Box D_2) \xrightarrow{\tau} C_1' \parallel_{E_C} (C_2 \Box D_2)$$

as well as

$$(C_1 \Box D_1) \parallel_{E_C} C_2 \xrightarrow{\tau} (C_1' \Box D_1) \parallel_{E_C} C_2$$

The following *bisimulation diagram* illustrates this case. ✓

$$\begin{array}{lccl} A = & (C_1 \parallel_{E_c} C_2) & \xrightarrow{\tau} & (C_1' \parallel_{E_c} C_2') = A' \\ & \mid & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \mid & & \vdots \\ B = & (C_1 \parallel_{E_c} (C_2 \,\Box\, D_2)) & \xrightarrow{\tau} & (C_1' \parallel_{E_c} (C_2' \,\Box\, D_2)) = B' \end{array}$$

Next, we consider $A \xrightarrow{\tau} A'$ and $(A,B) \notin \mathcal{R}'$. τ-transitions do not resolve an external choice. For the case of C_1 performing τ, based on the firing rules for external choice and parallel composition, the bisimulation diagram can be completed as follows. The other cases are similar. ✓

$$\begin{array}{lccl} A = & ((C_1 \parallel_{E_c} C_2) \,\Box\, (D_1 \parallel_{E_c} D_2)) & \xrightarrow{\tau} & ((C_1' \parallel_{E_c} C_2) \,\Box\, (D_1 \parallel_{E_c} D_2)) = A' \\ & \mid & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \mid & & \vdots \\ B = & ((C_1 \,\Box\, D_1) \parallel_{E_c} (C_2 \,\Box\, D_2)) & \xrightarrow{\tau} & ((C_1' \,\Box\, D_1) \parallel_{E_c} (C_2 \,\Box\, D_2)) = B' \end{array}$$

***op*-case:** Next, let $A \xrightarrow{op.x.t.a} A'$. For the case of $(A,B) \in \mathcal{R}'$, let $A = C_1 \parallel_{E_c} C_2$ and $A \xrightarrow{op.x.t.a} A'$. Obviously, any of the processes from X_1 can simulate $op.x.t.a$, resulting in two $\mathcal{R}'$-related processes, since the comprised external choice solely extends the set of possible steps, independent of any restriction on $initials(D_i)$. The case $A = D_1 \parallel_{E_c} D_2$ and X_2 is analogous. ✓

$$\begin{array}{lccl} A = & (C_1 \parallel_{E_c} C_2) & \xrightarrow{op.x.t.a} & (C_1' \parallel_{E_c} C_2') = A' \\ & \mid & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \mid & & \vdots \\ B = & (C_1 \parallel_{E_c} (C_2 \,\Box\, D_2)) & \xrightarrow{op.x.t.a} & (C_1' \parallel_{E_c} (C_2' \,\Box\, D_2)) = B' \end{array}$$

Now consider the case $(A,B) \notin \mathcal{R}'$, that is,

$$A = (C_1 \parallel_{E_c} C_2) \,\Box\, (D_1 \parallel_{E_c} D_2).$$

Then, either $C_1 \parallel_{E_c} C_2 \xrightarrow{op.x.t.a} A'$ or $D_1 \parallel_{E_c} D_2 \xrightarrow{op.x.t.a} A'$. Without loss of generality, we assume the first. Two separate cases have to be considered:

$op.x.t.a \in E_c$**:** Then, C_1 and C_2 have to synchronise on the execution of $op.x.t.a$. Thus,

$$C_1 \xrightarrow{op.x.t.a} A_1' \text{ and } C_2 \xrightarrow{op.x.t.a} A_2'$$

for some $A' = A_1' \parallel_{E_c} A_2'$, again based on the firing rules of the operational semantics of CSP. This yields that

$$(C_1 \,\Box\, D_1) \xrightarrow{op.x.t.a} A_1' \text{ and } (C_2 \,\Box\, D_2) \xrightarrow{op.x.t.a} A_2'$$

and therefore,

$$((C_1 \,\square\, D_1) \parallel_{E_C} (C_2 \,\square\, D_2)) \overset{op.x.t.a}{\rightarrow} (A_1' \parallel_{E_C} A_2').$$

As both successor states are identical, they are related by $\mathcal{R}'$. The bisimulation diagram for this case is given next. ✓

$$\begin{array}{lccl} A = & ((C_1 \parallel_{E_C} C_2) \,\square\, (D_1 \parallel_{E_C} D_2)) & \overset{op.x.t.a}{\longrightarrow} & (A_1' \parallel_{E_C} A_2') = A' \\ & \Big| & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \Big| & & \vdots \\ B = & ((C_1 \,\square\, D_1) \parallel_{E_C} (C_2 \,\square\, D_2)) & \overset{op.x.t.a}{\longrightarrow} & (A_1' \parallel_{E_C} A_2') = B' \end{array}$$

$op.x.t.a \notin E_C$**:** Then, either, but exactly one of the four components can execute $op.x.t.a$. Without loss of generality, let this component be C_1.

$$((C_1 \parallel_{E_C} C_2) \,\square\, (D_1 \parallel_{E_C} D_2)) \overset{op.x.t.a}{\longrightarrow} A'$$

yields

$$(C_1 \parallel_{E_C} C_2) \overset{op.x.t.a}{\longrightarrow} A'.$$

We get $C_1 \overset{op.x.t.a}{\longrightarrow} C_1'$ for some process C_1', such that $A' = (C_1' \parallel_{E_C} C_2)$. Furthermore,

$$(C_1 \,\square\, D_1) \overset{op.x.t.a}{\longrightarrow} C_1'$$

and thus,

$$((C_1 \,\square\, D_1) \parallel_{E_C} (C_2 \,\square\, D_2)) \overset{op.x.t.a}{\longrightarrow} (C_1' \parallel_{E_C} (C_2 \,\square\, D_2)).$$

Finally, $initials(D_2) \subseteq E_C$ holds: as $op.x.t.a \in initials(C_1 \parallel_{E_C} C_2)$, the set of initial events of $D_1 \parallel_{E_C} D_2$ and therefore the one of D_2 needs to be a subset of E_{S_1} due to Lemma 5.2.1. D_2 being a reachable state of $U|_{E_{S_2}}$ yields $initials(D_2) \subseteq E_C$, as no events from E_1 are possible. Thus,

$$((C_1' \parallel_{E_C} C_2), (C_1' \parallel_{E_C} (C_2 \,\square\, D_2))) \in \mathcal{R}',$$

which concludes this case. ✓

$$\begin{array}{lccl} A = & ((C_1 \parallel_{E_C} C_2) \,\square\, (D_1 \parallel_{E_C} D_2)) & \overset{op.x.t.a}{\longrightarrow} & (C_1' \parallel_{E_C} C_2) = A' \\ & \Big| & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \Big| & & \vdots \\ B = & ((C_1 \,\square\, D_1) \parallel_{E_C} (C_2 \,\square\, D_2)) & \overset{op.x.t.a}{\longrightarrow} & (C_1' \parallel_{E_C} (C_2 \,\square\, D_2))) = B' \end{array}$$

(2) For the reverse direction, let $(A,B) \in \mathcal{R}$ and $B \xrightarrow{e} B'$.

τ**-case:** Again, we start with $B \xrightarrow{\tau} B'$. For the case of $\mathcal{R}'$, consider

$$(C_1 \parallel_{E_C} (C_2 \,\square\, D_2)) \xrightarrow{\tau} (C_1 \parallel_{E_C} (C_2 \,\square\, D_2')),$$

that is, $D_2 \xrightarrow{\tau} D_2'$. First, as $\mathit{initials}(D_2) \subseteq E_C$, the process D_2 can solely perform synchronised events with C_1. However, a synchronisation between C_1 and D_2 is impossible due to Theorem 4.3.16, (5). Therefore, D_2 is incapable of performing any event within $C_1 \parallel_{E_C} (C_2 \,\square\, D_2)$. This allows us to simulate the τ-transition by $(C_1 \parallel_{E_C} C_2) \xrightarrow{\tau} (C_1 \parallel_{E_C} C_2)$. In any other case, including $(A,B) \notin \mathcal{R}'$, we apply the exact same rules and ideas of the forward direction. ✓.

op-**case:** In the case of $(A,B) \in \mathcal{R}'$, we solely consider $B = (C_1 \parallel_{E_C} (C_2 \,\square\, D_2))$ - the other three cases are accordingly shown. Let $B \xrightarrow{op.x.t.a} B'$. Again, it is impossible that D_2 performs any event within $C_1 \parallel_{E_C} (C_2 \,\square\, D_2)$. Furthermore, any (local or synchronised) step of C_i can be simulated by $C_1 \parallel_{E_C} C_2$. ✓

$$\begin{array}{lccc}
B = & (C_1 \parallel_{E_C} (C_2 \,\square\, D_2)) & \xrightarrow{op.x.t.a} & (C_1' \parallel_{E_C} (C_2' \,\square\, D_2)) \quad = B' \\
 & \Big| & & \vdots \\
 & \mathcal{R} & & \mathcal{R} \\
 & \Big| & & \vdots \\
A = & (C_1 \parallel_{E_C} C_2) & \xrightarrow{op.x.t.a} & (C_1' \parallel_{E_C} C_2') \quad = A'
\end{array}$$

Now let $B = ((C_1 \,\square\, D_1) \parallel_{E_C} (C_2 \,\square\, D_2))$. Here, both sides need to synchronise on $op.x.t.a$. Again, two cases need to be considered:

$op.x.t.a \in E_C$**:** Then, there exist B_1', B_2' such that

$$(C_1 \,\square\, D_1) \xrightarrow{op.x.t.a} B_1' \text{ and } (C_2 \,\square\, D_2) \xrightarrow{op.x.t.a} B_2'$$

for some B_1', B_2' and $B' = (B_1' \parallel_{E_C} B_2')$. Based on Theorem 4.3.16, (5), a synchronisation between C_1 and D_2 or C_2 and D_1 is impossible, as C and D were unable to synchronise before the decomposition. Therefore, without loss of generality, we deduce $C_1 \xrightarrow{op.x.t.a} B_1'$ and $C_2 \xrightarrow{op.x.t.a} B_2'$. Following up,

$$(C_1 \parallel_{E_C} C_2) \xrightarrow{op.x.t.a} (B_1' \parallel_{E_C} B_2')$$

and thus,

$$((C_1 \parallel_{E_C} C_2) \,\square\, (D_1 \parallel_{E_C} D_2)) \xrightarrow{op.x.t.a} (B_1' \parallel_{E_C} B_2').$$

As both successor states are identical, they are $\mathcal{R}$-related. ✓

$$\begin{array}{lcll} B = & ((C_1 \,\Box\, D_1) \parallel_{E_C} (C_2 \,\Box\, D_2)) & \overset{op.x.t.a}{\longrightarrow} & (B_1' \parallel_{E_C} B_2') = B' \\ & \mid & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \mid & & \vdots \\ A = & ((C_1 \parallel_{E_C} C_2) \,\Box\, (D_1 \parallel_{E_C} D_2)) & \overset{op.x.t.a}{\longrightarrow} & (B_1' \parallel_{E_C} B_2') = A' \end{array}$$

$op.x.t.a \notin E_C$**:** Again, exactly one of the four components executes $op.x.t.a$, which we assume to be C_1. From

$$((C_1 \,\Box\, D_1) \parallel_{E_C} (C_2 \,\Box\, D_2)) \overset{op.x.t.a}{\longrightarrow} B'$$

we get

$$(C_1 \,\Box\, D_1) \overset{op.x.t.a}{\longrightarrow} C_1'$$

for some process C_1' such that $B' = (C_1' \parallel_{E_C} (C_2 \,\Box\, D_2))$. From this, we get $C_1 \overset{op.x.t.a}{\longrightarrow} C_1'$ and thus,

$$(C_1 \parallel_{E_C} C_2) \overset{op.x.t.a}{\longrightarrow} (C_1' \parallel_{E_C} C_2).$$

This yields

$$((C_1 \parallel_{E_C} C_2) \,\Box\, (D_1 \parallel_{E_C} D_2)) \overset{op.x.t.a}{\longrightarrow} (C_1' \parallel_{E_C} C_2).$$

Finally, again based on Lemma 5.2.1, we get $initials(D_2) \subseteq E_C$ and

$$((C_1' \parallel_{E_C} C_2), (C_1' \parallel_{E_C} (C_2 \,\Box\, D_2))) \in \mathcal{R}'.\checkmark$$

$$\begin{array}{lcll} B = & ((C_1 \,\Box\, D_1) \parallel_{E_C} (C_2 \,\Box\, D_2)) & \overset{op.x.t.a}{\longrightarrow} & (C_1' \parallel_{E_C} (C_2 \,\Box\, D_2)) = B' \\ & \mid & & \vdots \\ & \mathcal{R} & & \mathcal{R} \\ & \mid & & \vdots \\ A = & ((C_1 \parallel_{E_C} C_2) \,\Box\, (D_1 \parallel_{E_C} D_2)) & \overset{op.x.t.a}{\longrightarrow} & (C_1' \parallel_{E_C} C_2) = A' \end{array}$$

Sequential Composition: Here, $\mathcal{R}' = \mathsf{Id}_{L^{CSP}}$.

(1) Let $(A, B) \in \mathcal{R}$ and $A \xrightarrow{e} A'$ for

$$A = ((C_1 \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)).$$

τ**-case:** $A \xrightarrow{\tau} A'$ yields that $C_1 = \texttt{Skip}$ and $C_2 = \texttt{Skip}$, based on the firing rule for sequential composition and thus, $A' = (D_1 \parallel_{E_C} D_2)$. Therefore,

$$C_1 \fatsemi D_1 \xrightarrow{\tau} D_1 \text{ and } C_2 \fatsemi D_2 \xrightarrow{\tau} D_2,$$

yielding $((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) \xrightarrow{\tau^2} (D_1 \parallel_{E_C} D_2)$. Both successor states are related by $\mathsf{Id}_{L^{CSP}}$, that is, $\mathcal{R}'$. ✓

$$\begin{array}{ccc} ((C_1 \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) & \xrightarrow{\tau} & (D_1 \parallel_{E_C} D_2) \\ \vert & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ \vert & & \vdots \\ ((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) & \xrightarrow{\widehat{\tau}} & (D_1 \parallel_{E_C} D_2) \end{array}$$

***op*-case:** Let $A \overset{op.x.t.a}{\rightarrow} A'$. Two cases need to be considered:

***op.x.t.a* $\in E_C$:** If $(C_1 \parallel_{E_C} C_2) \overset{op.x.t.a}{\rightarrow} A'$, we have

$$A' = ((C_1' \parallel_{E_C} C_2') \fatsemi (D_1 \parallel_{E_C} D_2))$$

for some $C_i' \in L^{CSP}$ and thus,

$$C_1 \overset{op.x.t.a}{\rightarrow} C_1' \text{ and } C_2 \overset{op.x.t.a}{\rightarrow} C_2',$$

from which can stepwise deduce

$$((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) \overset{op.x.t.a}{\rightarrow} ((C_1' \fatsemi D_1) \parallel_{E_C} (C_2' \fatsemi D_2)).$$

$$\begin{array}{ccc} ((C_1 \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) & \xrightarrow{op.x.t.a} & ((C_1' \parallel_{E_C} C_2') \fatsemi (D_1 \parallel_{E_C} D_2)) \\ \vert & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ \vert & & \vdots \\ ((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) & \xrightarrow{op.x.t.a} & ((C_1' \fatsemi D_1) \parallel_{E_C} (C_2' \fatsemi D_2)) \end{array}$$

If $(D_1 \parallel_{E_C} D_2) \overset{op.x.t.a}{\rightarrow} A'$, both, C_1 and C_2, need to have terminated, thus requiring $C_1 = \texttt{Skip}$, $C_2 = \texttt{Skip}$, and we proceed analogously. ✓

$$\begin{array}{ccc} ((\texttt{Skip} \parallel_{E_C} \texttt{Skip}) \fatsemi (D_1 \parallel_{E_C} D_2)) & \xrightarrow{op.x.t.a} & ((\texttt{Skip} \parallel_{E_C} \texttt{Skip}) \fatsemi (D_1' \parallel_{E_C} D_2')) \\ \vert & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ \vert & & \vdots \\ ((\texttt{Skip} \fatsemi D_1) \parallel_{E_C} (\texttt{Skip} \fatsemi D_2)) & \xrightarrow{op.x.t.a} & ((\texttt{Skip} \fatsemi D_1') \parallel_{E_C} (\texttt{Skip} \fatsemi D_2')) \end{array}$$

***op.x.t.a* $\notin E_C$:** In this case, again, either $C_1 \parallel_{E_C} C_2$ or $D_1 \parallel_{E_C} D_2$ perform *op.x.t.a*, where the latter requires $(C_1 \parallel_{E_C} C_2) = \texttt{Skip}$. The proof is straightforward and according to the previous proof steps. The bisimulation diagram for the first case, where we assume that C_1 performs *op.x.t.a*, is given next. ✓

$$\begin{array}{ccc} ((C_1 \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) & \overset{op.x.t.a}{\longrightarrow} & ((C_1' \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) \\ | & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ | & & \vdots \\ ((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) & \overset{op.x.t.a}{\longrightarrow} & ((C_1' \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) \end{array}$$

(2) Let $(A, B) \in \mathcal{R}$ and $B \xrightarrow{e} B'$ for $B = ((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2))$.

τ-case: Let $B \xrightarrow{\tau} B'$. Then, without loss of generality, $(C_1 \fatsemi D_1) \xrightarrow{\tau} D_1$, based on the firing rule for sequential composition. We deduce that $(C_1 \parallel_{E_C} C_2) \xrightarrow{\tau} (\texttt{Skip} \parallel_{E_C} C_2)$ holds. The bisimulation diagram is given next. ✓

$$\begin{array}{ccc} ((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) & \xrightarrow{\tau} & ((\texttt{Skip} \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) \\ | & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ | & & \vdots \\ ((C_1 \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) & \xrightarrow{\tau} & ((\texttt{Skip} \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) \end{array}$$

***op*-case:** Let $B \xrightarrow{op.x.t.a} B'$. Again, there are two separate cases:

***op.x.t.a* $\in E_C$:** Theorem 4.3.16, (5), ensures that a synchronisation within $(C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)$ can only occur between C_1 and C_2 or between D_1 and D_2. In case that D_1 and D_2 synchronise on *op.x.t.a*, C_1 and C_2 are equivalent to `Skip`. The remainder of this particular proof step is straightforward. We give the bisimulation diagram for the *C*-case next. ✓

$$\begin{array}{ccc} ((C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)) & \overset{op.x.t.a}{\longrightarrow} & ((C_1' \fatsemi D_1) \parallel_{E_C} (C_2' \fatsemi D_2)) \\ | & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ | & & \vdots \\ ((C_1 \parallel_{E_C} C_2) \fatsemi (D_1 \parallel_{E_C} D_2)) & \overset{op.x.t.a}{\longrightarrow} & ((C_1' \parallel_{E_C} C_2') \fatsemi (D_1 \parallel_{E_C} D_2)) \end{array}$$

***op.x.t.a* $\notin E_C$:** From the structure of the process $(C_1 \fatsemi D_1) \parallel_{E_C} (C_2 \fatsemi D_2)$, we know that C_i terminates before D_i. In addition, we have to show that C_2 terminates before D_1, which implies that C_1 terminates before D_2. Based on that, in order to complete this case, we may safely use $C_1 = \texttt{Skip}$ and $C_2 = \texttt{Skip}$, in case that D_i performs *op.x.t.a* $\notin E_C$. Assume that D_1 performs *op.x.t.a* $\notin E_C$. Then, by definition, either *op.t.x.a* $\in l[\mathbf{Ph}_1]$ or *op.t.x.a* $\in l[\mathbf{Ph}_3]$. In the first case, as *C* terminates before *D* and based on the definition of $\mathbf{Ph}_1$, the process *C* is completely assigned to $\mathbf{Ph}_1$. Thus, due to $\alpha C_2 \subseteq E_{S_2}$, C_2 can only perform synchronised events with C_1, which need to happen prior to

op.x.t.a. In the latter case, the set of events for C_2 are executed before any event of $l[\mathbf{Ph}_3]$, again yielding the termination of C_2 prior to D_1. The bisimulation diagram for a sole step of D_1 is given next. ✓

$$\begin{array}{ccc} ((\texttt{Skip} \fatsemi D_1) \parallel_{E_\mathbf{C}} (\texttt{Skip} \fatsemi D_2)) & \xrightarrow{op.x.t.a} & ((\texttt{Skip} \fatsemi D_1') \parallel_{E_\mathbf{C}} (\texttt{Skip} \fatsemi D_2)) \\ \vert & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ \vert & & \vdots \\ ((\texttt{Skip} \parallel_{E_\mathbf{C}} \texttt{Skip}) \fatsemi (D_1 \parallel_{E_\mathbf{C}} D_2)) & \xrightarrow{op.x.t.a} & ((\texttt{Skip} \parallel_{E_\mathbf{C}} \texttt{Skip}) \fatsemi (D_1' \parallel_{E_\mathbf{C}} D_2)) \end{array}$$

Parallel Composition: Again, $\mathcal{R}' = \varnothing$. For the case of parallel composition, weak bisimilarity of the processes

$$A = (C_1 \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2) \text{ and}$$

$$B = (C_1 \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)$$

has to be shown. First, we consider the case of a τ-transition for both directions.

(1) Let $(A, B) \in \mathcal{R}$ and $A \xrightarrow{\tau} A'$. This case is immediate based on the rules for promoting τ-transitions within a parallel composition. ✓

(2) For $B \xrightarrow{\tau} B'$, we proceed analogously.

For either $A \xrightarrow{op.x.t.a} A'$ or $B \xrightarrow{op.x.t.a} B'$, several cases need to be separated, making a case differentiation over all cases rather tedious. As most of these cases refer to the transition laws for CSP, corresponding to the applications for the external choice and sequential composition, we precisely deal with the decisive cases and only sketch the straightforward cases.

Figure 5.9 shows the different cases, which need to be considered for A or B performing *op.x.t.a*. These are:

(a) $op.x.t.a \in (S \cap E_\mathbf{C})$,

(b) $op.x.t.a \in (S \cap E_1)$,

(c) $op.x.t.a \in (S \cap E_2)$,

(d) $op.x.t.a \in (E_\mathbf{C} \setminus S)$,

(e) $op.x.t.a \in (E_1 \setminus S)$ and

(f) $op.x.t.a \in (E_2 \setminus S)$.

For the bisimulation proof, there is one decisive case for both directions. We give the intuitive ideas first:

- $A = (C_1 \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)$ performing an event from $S \setminus E_\mathbf{C}$ might cause a wrong synchronisation between C_2 and D_1 or C_1 and D_2. However, as the event is either an element of E_1 or E_2 but never an element of both sets, this is impossible.

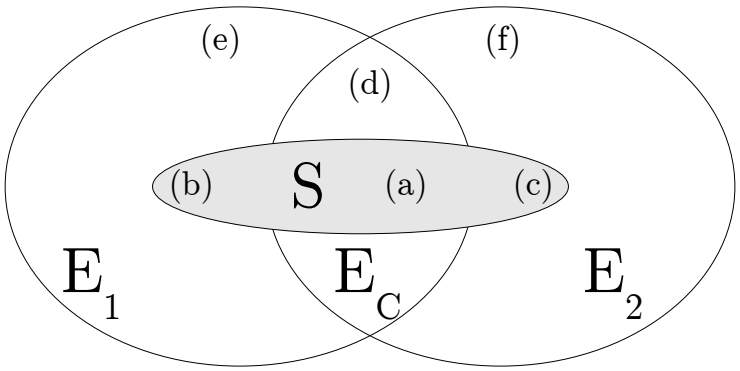


Figure 5.9: Case differentiation for Lemma 5.2.3, parallel composition

- $B = (C_1 \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)$ performing an event from $E_\mathbf{C} \setminus S$ might cause a wrong synchronisation between C_2 and D_1 or C_1 and D_2 as well. Here, Theorem 4.3.16 shows that the addressing extension prevents this from happening.

Next, we show the bisimilarity conditions for all six cases:

$op.x.t.a \in (S \cap E_\mathbf{C})$**:** Independent of Condition (1) or (2), all four processes C_1, C_2, D_1 and D_2 have to synchronise on $op.x.t.a$. Showing both conditions is immediate, based on applying the firing rules for CSP. ✓

$op.x.t.a \in (S \cap E_1)$**:** For implication (1), assume that

$$((C_1 \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)) \xrightarrow{op.x.t.a} ((C_1' \parallel_{E_\mathbf{C}} C_2') \parallel_S (D_1' \parallel_{E_\mathbf{C}} D_2')).$$

Based on $op.x.t.a \in E_1$, the synchronisation must be performed by C_1 and D_1. Thus, $C_2' = C_2$, $D_2' = D_2$ and $(C_1 \parallel_{S \cap E_{S_1}} D_1) \xrightarrow{op.x.t.a} (C_1' \parallel_{S \cap E_{S_1}} D_1')$. This yields

$$((C_1 \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)) \xrightarrow{op.x.t.a} ((C_1' \parallel_{S \cap E_{S_1}} D_1') \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)),$$

as $op.x.t.a \notin E_\mathbf{C}$.

$$\begin{array}{ccc} ((C_1 \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)) & \xrightarrow{op.x.t.a} & ((C_1' \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1' \parallel_{E_\mathbf{C}} D_2)) \\ | & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ | & & \vdots \\ ((C_1 \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)) & \xrightarrow{op.x.t.a} & ((C_1' \parallel_{S \cap E_{S_1}} D_1') \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)) \end{array}$$

Implication (2) is straightforward. ✓

$op.x.t.a \in (S \cap E_2)$**:** Analogous to the previous case. ✓

$op.x.t.a \in (E_\mathbf{C} \setminus S)$**:** Here, implication (1) is straightforward. For implication (2), let

$$((C_1 \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)) \xrightarrow{op.x.t.a} ((C_1' \parallel_{S \cap E_{S_1}} D_1') \parallel_{E_\mathbf{C}} (C_2' \parallel_{S \cap E_{S_2}} D_2')).$$

Theorem 4.3.16 yields that only C_1 and C_2 or D_1 and D_2 are able to synchronise. We assume the first, thus yielding $D_1' = D_1$ and $D_2' = D_2$. We get $(C_1 \parallel_{E_\mathbf{C}} C_2) \xrightarrow{op.x.t.a} (C_1' \parallel_{E_\mathbf{C}} C_2')$ and finally

$$((C_1 \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)) \xrightarrow{op.x.t.a} ((C_1' \parallel_{E_\mathbf{C}} C_2') \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)). \checkmark$$

$$\begin{array}{ccc} ((C_1 \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2 \parallel_{S \cap E_{S_2}} D_2)) & \xrightarrow{op.x.t.a} & ((C_1' \parallel_{S \cap E_{S_1}} D_1) \parallel_{E_\mathbf{C}} (C_2' \parallel_{S \cap E_{S_2}} D_2)) \\ \vert & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ \vert & & \vdots \\ ((C_1 \parallel_{E_\mathbf{C}} C_2) \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)) & \xrightarrow{op.x.t.a} & ((C_1' \parallel_{E_\mathbf{C}} C_2') \parallel_S (D_1 \parallel_{E_\mathbf{C}} D_2)) \end{array}$$

$op.x.t.a \in (E_1 \setminus S)$**:** Independent of Condition (1) or (2), exactly one of the four processes needs to perform $op.x.t.a$, which is straightforward. ✓

$op.x.t.a \in (E_2 \setminus S)$**:** Analogous to the previous case. ✓

□

5.2.2 Correctness of the Decomposition: CSP part

Finally, we show correctness for the decomposition of $S.\texttt{main}$ by using the results from the previous sections. Again, we use weak bisimulation as the method of proof: we construct a weak bisimulation relation, comprising tuples

$$(P, P[\![\mathtt{R}_1^\mathtt{C}]\!]|_{E_{S_1}} \parallel_{E_\mathbf{C}} P[\![\mathtt{R}_2^\mathtt{C}]\!]|_{E_{S_2}}),$$

where P denotes any reachable state of the LTS of $S.\texttt{main}$. As we show trace equivalence modulo the renaming by transmission parameters and address parameters, we accordingly show bisimilarity not explicitly denoting the renaming.[2] For simplification, we let $P_\mathtt{R}$ denote $P[\![\mathtt{R}^\mathtt{C}]\!]$.

In the theorem, we will use a lemma from [Brü08], namely Lemma 6.1.2. It relates the possible transitions of a CSP process to transitions of a projection of this process. Next, we state the main theorem of this section:

Theorem 5.2.4. *(Correctness of the decomposition: CSP part)*
Let S be a specification, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut, yielding a decomposition into S_1 and S_2, according to Definition 4.3.24. Then, the following holds:

$$S.\texttt{main} =_T (S_1.\texttt{main} \parallel_{E_\mathbf{C}} S_2.\texttt{main})[\![\mathtt{R}']\!].$$

[2] As a matter of course, the renaming is solely *syntactically* neglected. We still have to use the *properties* of the additional parameters, as they ultimately ensure the correctness of the decomposition.

Proof: We show that $S.\texttt{main}$ and $S_1.\texttt{main} \parallel_{E_\mathtt{C}} S_2.\texttt{main}$ are the initial states of a weak bisimulation

$$\mathcal{R} := \{(P, P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \mid P \in L^{CSP} \text{ reachable state of LTS of } S.\texttt{main},\ P_\mathtt{R} = P[\![\mathtt{R}^\mathtt{C}]\!]\}.$$

Again, we need to show two implications:

(1) If $(P, P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \in \mathcal{R}$ and $P \xrightarrow{op.x} P'$ for $op.x \in E_S$ $[P \xrightarrow{\tau} P']$, then there exists some Q', such that

$$(P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \xrightarrow{\widehat{op.x.t.a}} Q' \ [(P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \xrightarrow{\widehat{\tau}} Q']$$

for some $op.x.t.a \in E_{S'}$ and $(P', Q') \in \mathcal{R}$.

(2) If $(P, P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \in \mathcal{R}$ and

$$(P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \xrightarrow{op.x.t.a} Q' \ [P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \xrightarrow{\tau} Q'],$$

then there exists some P', such that $P \xrightarrow{\widehat{op.x}} P'$ for $op.x \in E_S$ $[P \xrightarrow{\widehat{\tau}} P']$ and $(P', Q') \in \mathcal{R}$.

For the first implication, it is sufficient to show that $op.x.t.a \in E_{S'}$ *exists*.

(**1**)**:** First, let $P \xrightarrow{\tau} P'$. Based on the firing laws for CSP, a τ-transition is preserved by a renaming as well as hiding. Thus, we immediately get

$$(P_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P_\mathtt{R}|_{E_{S_2}}) \xrightarrow{\tau} (P'_\mathtt{R}|_{E_{S_1}} \parallel_{E_\mathtt{C}} P'_\mathtt{R}|_{E_{S_2}}).$$

Now let $P \xrightarrow{op.x} P'$ for $op.x \in E_S$. First, assume that P is composed of two parallel processes, with op being synchronised. Based on Theorem 4.3.16, (2) and (3), if P performs a synchronised step, the process $P_\mathtt{R}$ can accordingly perform this step, as the renaming preserves the synchronisation structure. Thus, we do not separately need to deal with this particular structure of P. Next, we have to distinguish between three cases for $op.x$:

$op.x \in E_\mathtt{C}$**:** In this case, we apply the first property of Lemma 6.1.2, [Brü08]. For $e \in E$, it states that performing e for P and $P|_E$ leads to corresponding successor states Q and $Q|_E$:

$$(P \xrightarrow{e} Q \wedge e \in E) \Rightarrow P|_E \xrightarrow{e} Q|_E.$$

In our context and based upon the previous observation, the property yields

$$P_\mathtt{R}|_{E_{S_1}} \xrightarrow{op.x.t_1.a_1} P'_\mathtt{R}|_{E_{S_1}} \text{ and } P_\mathtt{R}|_{E_{S_2}} \xrightarrow{op.x.t_2.a_2} P'_\mathtt{R}|_{E_{S_2}}$$

for some $op.x.t_i.a_i \in E_{S'}$. In order to deduce that $P_\mathtt{R}|_{E_{S_1}}$ and $P_\mathtt{R}|_{E_{S_2}}$ can do a *synchronous* step, there needs to exist some $op.t.x.a \in E_{S'}$, which both processes can perform. This is the case: the transmission parameters are not

restricted by the CSP part at all. Thus, any values are possible for t_1 and t_2. For the address parameters, Theorem 4.3.16, (1), showed that their values are identical for both, $S_1.\texttt{main}$ and $S_2.\texttt{main}$. We deduce that there exists $op.t.x.a \in E_{S'}$, such that

$$(P_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P_{\mathrm{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} (P'_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P'_{\mathrm{R}}|_{E_{S_2}})$$

and $(P', P'_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P'_{\mathrm{R}}|_{E_{S_2}}) \in \mathcal{R}$. ✓

$op.x \in E_1$**:** Again, we deduce

$$P_{\mathrm{R}}|_{E_{S_1}} \xrightarrow{op.x.t_1.a_1} P'_{\mathrm{R}}|_{E_{S_1}}.$$

Based on $op.x \notin E_{\mathrm{C}}$ and the operational semantics of CSP, we get

$$(P_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P_{\mathrm{R}}|_{E_{S_2}}) \xrightarrow{op.x.t_1.a_1} (P'_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P_{\mathrm{R}}|_{E_{S_2}}).$$

By using the firing rule for CSP hiding and $op.x \notin E_2$, we deduce that

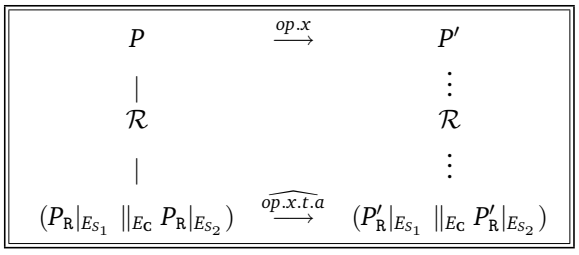
$$P_{\mathrm{R}}|_{E_{S_2}} \xrightarrow{\tau} P'_{\mathrm{R}}|_{E_{S_2}}$$

and thus,

$$(P_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P_{\mathrm{R}}|_{E_{S_2}}) \xrightarrow{\widehat{op.x.t_1.a_1}} (P'_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P'_{\mathrm{R}}|_{E_{S_2}}).$$

Again, $(P', P'_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P'_{\mathrm{R}}|_{E_{S_2}}) \in \mathcal{R}$. ✓

$op.x \in E_2$**:** Analogous to the second case. ✓



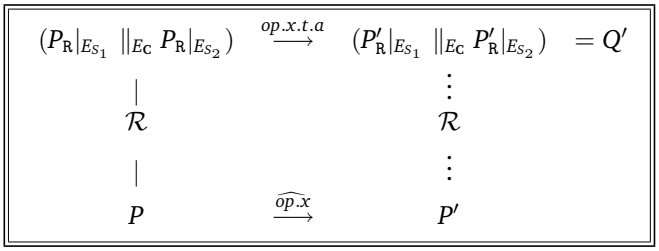
(**2**)**:** Let

$$(P_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P_{\mathrm{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} Q'$$

for some $op.x.t.a \in E_{S'}$. We show that there exists $P' \in L^{CSP}$, such that $P \xrightarrow{op.x} P'$ and

$$Q' = (P'_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} P'_{\mathrm{R}}|_{E_{S_2}}),$$

by induction on the structure of P.



Induction Basis: Let $P = e \to Q$ for some $e \in E_S$ and $Q \in L^{CSP}$. Based on Definition 4.3.3, we have

$$(e \to Q)|_E := \begin{cases} Q|_E, & e \notin E, \\ e \to Q|_E, & \textit{otherwise}. \end{cases}$$

We have to distinguish between three cases:[3]

$\mathtt{R}^{\mathtt{C}}(e) \subseteq E_{\mathtt{C}}$**:** Based on $(e \to P)[\![\mathtt{R}]\!] = e' : \mathtt{R}(e) \to P[\![\mathtt{R}]\!]$ ([Sch09]),

$$(P_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} P_{\mathtt{R}}|_{E_{S_2}}) = ((e' : \mathtt{R}_1^{\mathtt{C}}(e) \to Q_{\mathtt{R}}|_{E_{S_1}}) \parallel_{E_{\mathtt{C}}} (e'' : \mathtt{R}_2^{\mathtt{C}}(e) \to Q_{\mathtt{R}}|_{E_{S_2}})).$$

As $(P_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} P_{\mathtt{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} Q'$, we get $op.x.t.a \in (\mathtt{R}_1^{\mathtt{C}}(e) \cap \mathtt{R}_2^{\mathtt{C}}(e))$. Thus, $e = op.x$. Moreover,

$$(P_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} P_{\mathtt{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} (Q_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} Q_{\mathtt{R}}|_{E_{S_2}}),$$

as both processes have to synchronise on $op.x.t.a \in E_{\mathtt{C}}$. Obviously, $P \xrightarrow{e} Q$ holds, and finally, $(Q, Q_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} Q_{\mathtt{R}}|_{E_{S_2}}) \in \mathcal{R}$. ✓

$\mathtt{R}^{\mathtt{C}}(e) \in E_1$**:** Let $e' = \mathtt{R}^{\mathtt{C}}(e)$. In this case,

$$(P_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} P_{\mathtt{R}}|_{E_{S_2}}) = ((e' \to Q_{\mathtt{R}}|_{E_{S_1}}) \parallel_{E_{\mathtt{C}}} Q_{\mathtt{R}}|_{E_{S_2}}) =: X,$$

as the projection eliminates e' for the right hand side of the parallel composition. In case that $op.t.x.a = e'$ holds, the process X switches to $Q_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} Q_{\mathtt{R}}|_{E_{S_2}}$, and we reside in the first case. However, we need to show that X must not be able to perform any other event than e', that is, $Q_{\mathtt{R}}|_{E_{S_2}}$ is incapable of performing a non-synchronised step. But this is the case: from Lemma 5.2.1 and based on $e' \in E_1$, we know that the set of initial events of $Q_{\mathtt{R}}|_{E_{S_2}}$ is a subset of E_{S_1} and thus $E_{\mathtt{C}}$. Therefore, $Q_{\mathtt{R}}|_{E_{S_2}}$ can initially only do a synchronous step which is impossible as the sole initial event for the parallel composition is an event from E_1. ✓

$\mathtt{R}^{\mathtt{C}}(e) \in E_2$**:** According to the previous case. ✓

Induction Hypothesis: Assume that the property is shown for T and U.

[3] Here, we need to refer to the renaming $\mathtt{R}^{\mathtt{C}}$ as we have to distinguish between the different sets of events which e is assigned to in the decomposition.

Induction Step: The induction step needs to distinguish between $P = T \circ U$ with $\circ \in \{\Box, \sqcap, \fatsemi, \|_S, \,|||\,\}$. Both choice operators have the same interpretation in the CSP traces model. Moreover, interleaving is a special case of parallel composition with an empty synchronisation alphabet. That leaves a case differentiation for $\circ \in \{\Box, \fatsemi, \|_S\}$. In any of the three cases, we apply Lemma 5.2.3. As the lemma already dealt with τ-transitions, we do not need to consider them again.

$P = T \Box U$**:** Let

$$((T \Box U)_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; (T \Box U)_{\mathtt{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} Q'.$$

Definition 4.3.3 yields

$$((T_{\mathtt{R}}|_{E_{S_1}} \Box U_{\mathtt{R}}|_{E_{S_1}}) \;\|_{E_{\mathtt{C}}}\; (T_{\mathtt{R}}|_{E_{S_2}} \Box U_{\mathtt{R}}|_{E_{S_2}})) \xrightarrow{op.x.t.a} Q'.$$

Next, we apply Lemma 5.2.3 for the case of external choice and deduce

$$((T_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; T_{\mathtt{R}}|_{E_{S_2}}) \Box (U_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; U_{\mathtt{R}}|_{E_{S_2}})) \xrightarrow{\widehat{op.x.t.a}} Q'.$$

Without loss of generality, the left hand side performs a (synchronous or asynchronous) step. From the induction hypothesis, we deduce the existence of T', such that $T \xrightarrow{\widehat{op.x}} T'$ and $Q' = (T'_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; T'_{\mathtt{R}}|_{E_{S_2}})$. The operational semantics of CSP yields $(T \Box U) \xrightarrow{\widehat{op.x}} T'$. ✓

$$\begin{array}{ccccc}
(T \Box U)_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; (T \Box U)_{\mathtt{R}}|_{E_{S_2}} & \xrightarrow{op.x.t.a} & T'_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; T'_{\mathtt{R}}|_{E_{S_2}} & = Q' \\
\Big| & & \vdots & \\
\mathcal{R} & & \mathcal{R} & \\
\Big| & & \vdots & \\
T \Box U & \xrightarrow{\widehat{op.x}} & T' & = P'
\end{array}$$

$P = T \fatsemi U$**:** Let

$$((T \fatsemi U)_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; (T \fatsemi U)_{\mathtt{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} Q'.$$

Definition 4.3.3 yields

$$((T_{\mathtt{R}}|_{E_{S_1}} \fatsemi U_{\mathtt{R}}|_{E_{S_1}}) \;\|_{E_{\mathtt{C}}}\; (T_{\mathtt{R}}|_{E_{S_2}} \fatsemi U_{\mathtt{R}}|_{E_{S_2}})) \xrightarrow{op.x.t.a} Q'$$

and the application of Lemma 5.2.3 for the case of sequential composition

$$((T_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; T_{\mathtt{R}}|_{E_{S_2}}) \fatsemi (U_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; U_{\mathtt{R}}|_{E_{S_2}})) \xrightarrow{\widehat{op.x.t.a}} Q'.$$

First, let

$$((T_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; T_{\mathtt{R}}|_{E_{S_2}}) \fatsemi (U_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; U_{\mathtt{R}}|_{E_{S_2}})) \xrightarrow{\widehat{op.x.t.a}} (Q'_1 \fatsemi (U_{\mathtt{R}}|_{E_{S_1}} \;\|_{E_{\mathtt{C}}}\; U_{\mathtt{R}}|_{E_{S_2}})).$$

From the induction hypothesis, we deduce the existence of T', such that $T \xrightarrow{\widehat{op.x}} T'$ and

$$Q_1' = (T'_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} T'_{\mathtt{R}}|_{E_{S_2}}).$$

The operational semantics of CSP yields $(T \fatsemi U) \xrightarrow{\widehat{op.x}} (T' \fatsemi U)$. Finally,

$$Q' = (T'_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} T'_{\mathtt{R}}|_{E_{S_2}}) \fatsemi (U_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} U_{\mathtt{R}}|_{E_{S_2}}),$$

which is equivalent to

$$(T'_{\mathtt{R}}|_{E_{S_1}} \fatsemi U_{\mathtt{R}}|_{E_{S_1}}) \parallel_{E_{\mathtt{C}}} (T'_{\mathtt{R}}|_{E_{S_2}} \fatsemi U_{\mathtt{R}}|_{E_{S_2}}),$$

according to Lemma 5.2.3. The latter process is equal to

$$(T' \fatsemi U)_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} (T' \fatsemi U)_{\mathtt{R}}|_{E_{S_2}},$$

based on Definition 4.3.3. We conclude that the successor states are $\mathcal{R}$-related. A step for the process $U_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} U_{\mathtt{R}}|_{E_{S_2}}$ is analogous, based on the fact that in this case, $T_{\mathtt{R}}|_{E_{S_1}} = \mathtt{Skip}$ and $T_{\mathtt{R}}|_{E_{S_2}} = \mathtt{Skip}$ needs to hold. ✓

$$\begin{array}{ccc}
((T \fatsemi U)_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} (T \fatsemi U)_{\mathtt{R}}|_{E_{S_2}}) & \xrightarrow{op.x.t.a} & ((T' \fatsemi U)_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} (T' \fatsemi U)_{\mathtt{R}}|_{E_{S_2}}) \\
| & & \vdots \\
\mathcal{R} & & \mathcal{R} \\
| & & \vdots \\
(T \fatsemi U) & \xrightarrow{\widehat{op.x}} & (T' \fatsemi U)
\end{array}$$

$P = T \parallel_S U$**:** We assume

$$((T \parallel_S U)_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} (T \parallel_S U)_{\mathtt{R}}|_{E_{S_2}}) \xrightarrow{op.x.t.a} Q'.$$

Applying 4.3.3 and subsequently Lemma 5.2.3 for the case of parallel composition yields

$$((T_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} T_{\mathtt{R}}|_{E_{S_2}}) \parallel_S (U_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} U_{\mathtt{R}}|_{E_{S_2}})) \xrightarrow{\widehat{op.x.t.a}} (Q_1' \parallel_{E_{\mathtt{C}}} Q_2').$$

Two cases for $op.x.t.a$ have to be considered:

$op.x.t.a \in S$**:** The induction hypothesis yields the existence of two processes T' and U', such that $T \xrightarrow{\widehat{op.x}} T'$ and $U \xrightarrow{\widehat{op.x}} U'$ holds for $Q_1' = (T'_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} T'_{\mathtt{R}}|_{E_{S_2}})$ and $Q_2' = (U'_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} U'_{\mathtt{R}}|_{E_{S_2}})$. The operational semantics of CSP yields $(T \parallel_S U) \xrightarrow{\widehat{op.x}} (T' \parallel_S U')$. Finally,

$$\begin{aligned} Q' &= Q_1' \parallel_{E_{\mathtt{C}}} Q_2' \\ &= (T'_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} T'_{\mathtt{R}}|_{E_{S_2}}) \parallel_S (U'_{\mathtt{R}}|_{E_{S_1}} \parallel_{E_{\mathtt{C}}} U'_{\mathtt{R}}|_{E_{S_2}}), \end{aligned}$$

with the latter process being equivalent to

$$(T'_{\mathrm{R}}|_{E_{S_1}} \parallel_{S \cap E_{S_1}} U'_{\mathrm{R}}|_{E_{S_1}}) \parallel_{E_{\mathrm{C}}} (T'_{\mathrm{R}}|_{E_{S_2}} \parallel_{S \cap E_{S_2}} U'_{\mathrm{R}}|_{E_{S_2}}),$$

according to Lemma 5.2.3 and, consecutively, equivalent to

$$(T' \parallel_S U')_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} (T' \parallel_S U')_{\mathrm{R}}|_{E_{S_2}}$$

by Definition 4.3.3. We conclude that the successor states are $\mathcal{R}$-related. ✓

$$\begin{array}{ccc} ((T \parallel_S U)_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} (T \parallel_S U)_{\mathrm{R}}|_{E_{S_2}}) & \xrightarrow{op.x.t.a} & ((T' \parallel_S U')_{\mathrm{R}}|_{E_{S_1}} \parallel_{E_{\mathrm{C}}} (T' \parallel_S U')_{\mathrm{R}}|_{E_{S_2}}) \\ | & & \vdots \\ \mathcal{R} & & \mathcal{R} \\ | & & \vdots \\ (T \parallel_S U) & \xrightarrow{\widehat{op.x}} & (T' \parallel_S U') \end{array}$$

op.x.t.a $\notin S$: Analogous to the previous case, except for applying the induction hypothesis only once for either T or U. ✓ □

This completes the proof of the correct decomposition of the CSP part. Next, we accordingly show correctness for the decomposition of the Object-Z part of a specification.

5.3 Correctness for the Object-Z Part

In the introduction of this chapter, we pointed out the general strategy for the correctness proof. In particular, for showing correctness of the decomposition of a specification's Object-Z part, we need to take the traces of its CSP part into account:

$$S.OZ =_T (S_1.OZ \parallel_{E_{\mathrm{C}}} S_2.OZ)[\![\mathrm{R}']\!]$$

is only satisfied, if the ordering of events for the Object-Z part adheres to the sequences of the CSP part. We illustrate this with a small example:

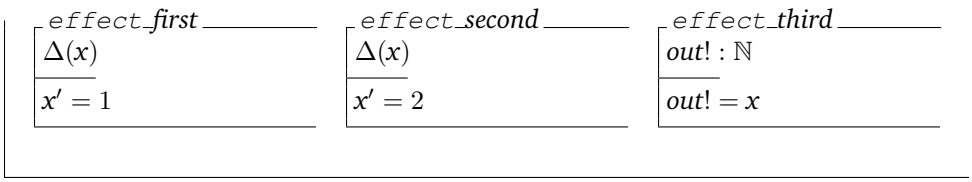
Example 5.3.1. *Consider the following specification Simple. Its CSP part subsequently performs three operations. The operation first assigns the value* 1 *to the sole state variable x. Next, second assigns* 2 *to x in case that the precondition* $x = 1$ *is satisfied. Finally, third outputs the value of x:*

Simple

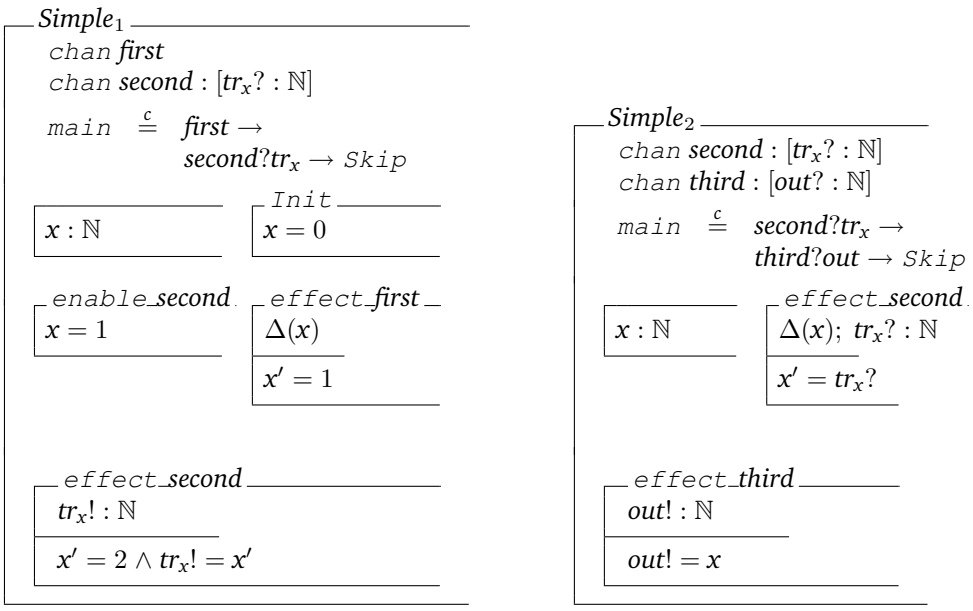
`chan` *first*, *second* `chan` *third* : [*out*? : $\mathbb{N}$]

`main` $\stackrel{c}{=}$ *first* $\rightarrow$ *second* $\rightarrow$ *third*?*out* $\rightarrow$ `Skip`

	`Init`	`enable_second`
$x : \mathbb{N}$	$x = 0$	$x = 1$



The possible (single) cut $E_{\mathbf{C}} = \{\mid second \mid\}$ leads to the following decomposition, requiring an additional transmission parameter:



Based on the CSP part of Simple, there is only one possible sequence of operations, namely $\langle first, second, third\rangle$. However, in sole regard to the specification's Object-Z part, the ordering $\langle first, third, second\rangle$ is possible. As the introduction of transmission parameters refers to the CFG and thus the CSP part of a specification, correct values for the state variables cannot be ensured. In particular, the event trace $\langle first, third.1, second.2\rangle$ is an element of traces$(Simple_1.OZ \parallel_{E_{\mathbf{C}}} Simple_2.OZ)$, whereas $\langle first, third.1, second.2\rangle$ is not *an element of traces$(Simple.OZ)$: in the decomposition, the value of x needs to be transmitted before third takes place, which is not the case, if the execution of the final two events is switched.*

In the following correctness proof, we refer to the LTS semantics of Object-Z, as introduced in Definition 2.2.3. The proof itself requires us to reason about the intermediate states of *S.OZ*, that is, its state valuations, as we are now explicitly dealing with data dependences. However, the semantic equivalence we aim at, is trace equivalence within the CSP traces model, *disregarding* states of the Object-Z part. This allows that the valuations of the state variables within $S.OZ$ and $S_1.OZ \parallel_{E_{\mathbf{C}}} S_2.OZ$ are possibly *inconsistent* if their values do not influence the observable behaviour of the class, that is, the traces of the CSP part.

In Section 5.3.2, we clarify what *inconsistency* between state valuations means. Moreover, we define the assumption that traces of the Object-Z part need to adhere to the CSP part.

Beforehand, we start by showing some properties related to the decomposition of the Object-Z part, which the actual correctness proof uses.

5.3.1 Properties of the Decomposition: Object-Z Part

Corresponding to the previous section for the CSP part, we introduce and prove some properties of the decomposition of the Object-Z part: the first section will summarise some characteristics of the DG, which the definition for a valid cut necessitates. Afterwards, we show correctness of the restriction of the initial state schema and its optimisation.

No Dependences between Different Segments

A valid cut rules out several dependence edges between different segments of the DG. In particular, edges must not reach back to a cut set or circumvent the cut.

As we continuously need to refer to the correctness criteria **no reaching back** and **no crossing** with respect to the phases and cut sets of the fragmented DG, we will now give a lemma, summarising possible violations of these conditions. Here, we consider edges with the target node being at an *earlier stage* than the source node:

Definition 5.3.2. *(Earlier stage)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the DG of a specification S and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. We say that* $\mathsf{n} \in \mathtt{op}(N)$ *is at an* earlier stage *than* $\mathsf{n'} \in \mathtt{op}(N)$*, if and only if one of the four conditions*

1) $\mathsf{n} \in \mathbf{Ph}_1$ *and* $\mathsf{n'} \in \mathbf{Ph}_2$*,*

2) $\mathsf{n} \in \mathbf{C}_1$ *and* $\mathsf{n'} \in \mathbf{Ph}_2$*,*

3) $\mathsf{n} \in \mathbf{Ph}_2$ *and* $\mathsf{n'} \in \mathbf{Ph}_3$*,*

4) $\mathsf{n} \in \mathbf{C}_2$ *and* $\mathsf{n'} \in \mathbf{Ph}_3$

holds.

The definition is motivated by a property, which we subsequently show: if a DG node is at an earlier stage with respect to another one, a control flow edge or data dependence from the latter to the first node causes a violation of one of the correctness criteria **disjointness**, **no reaching back** or **no crossing**:

Lemma 5.3.3. *(No data dependences to an earlier stage)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the DG of a specification S and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. If* $\mathsf{n} \in \mathtt{op}(N)$ *is at an earlier stage than* $\mathsf{n'} \in \mathtt{op}(N)$*, there must not be a data dependence from* n' *to* n*:* $\mathsf{n'} \dashrightarrow \mathsf{n}$ *is impossible.*

Proof. For the first and the third case of Definition 5.3.2, a data dependence n'⇢ n violates the correctness criterion **no crossing**. For the other cases, **no reaching back** is violated. □

A corresponding lemma considering control flow edges is given next.

Lemma 5.3.4. *(No control flow edges to an earlier stage)*
Let $\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$ *be the DG of a specification S and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. If* $\mathsf{n} \in \mathtt{op}(N)$ *is at an earlier stage than* $\mathsf{n'} \in \mathtt{op}(N)$*, there must not be a control flow edge from* n' *to* n*:* $\mathsf{n'} \longrightarrow \mathsf{n}$ *is impossible.*

Proof. Consider the first case of Definition 5.3.2: $\mathsf{n} \in \mathbf{Ph}_1$ and $\mathsf{n'} \in \mathbf{Ph}_2$. Assume that $\mathsf{n'} \longrightarrow \mathsf{n}$. But then, $\mathsf{n} \in (\mathbf{Ph}_1 \cap \mathbf{Ph}_2)$, contradicting **disjointness**. The third case is analogous. For the other cases, **no reaching back** is violated, according to Lemma 5.3.3. □

Figure 5.10 illustrates the definition and both lemmas, where edges denote disallowed control flow edges and data dependences.

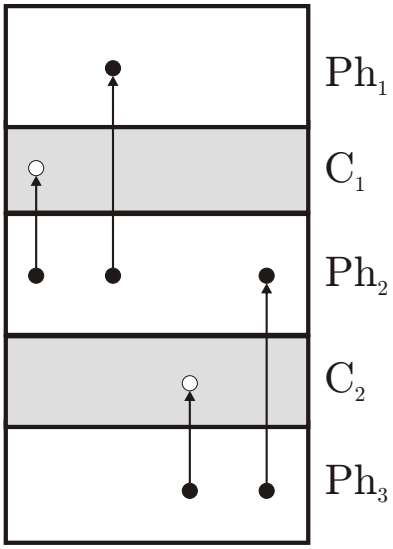


Figure 5.10: Illustration of Definition 5.3.2 and Lemmas 5.3.3, 5.3.4

We aim at lifting Lemma 5.3.4 to *paths* of the CFG, that is, we want to state that the CFG must not comprise paths connecting a node with another one from an earlier stage. However, this is not always the case, as recursive calls from the third phase back to the first one are possible. More generally, our correctness proof will need to distinguish between paths returning to an earlier stage with recursion (which is possible) and without recursion (which is impossible). This motivates the following definition:

Definition 5.3.5. *(Recursion-free CFG path)*
Let $\mathtt{CFG}_S = (N, \longrightarrow)$ *be the CFG of a specification S, and let* $\pi \in \mathtt{path}_{\mathtt{CFG}}$*. We say that*

π is (outer-) recursion-free*, if, and only if, π does not comprise two subsequent nodes* $\mathsf{call}.X \in \mathbf{Ph}_3$ *and* $\mathsf{start}.X \in \mathbf{Ph}_1$ *for any* $X \in L^{CSP}$.

The following lemma bridges the gap between paths of the CFG of S and traces of the CSP part by characterising traces of $S.\texttt{main}$ without corresponding CFG paths:

Lemma 5.3.6. *(CSP trace to an earlier stage requires interleaving or recursion)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ *be the DG of a specification S, and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut. Let* $\mathsf{n} \in \texttt{op}(N)$ *be at an earlier stage than* $\mathsf{n'} \in \texttt{op}(N)$*, and let n and n′ denote their corresponding occurrences within* $S.\texttt{main}$*. If there exists* $tr \in traces(S.\text{main})$ *and indices* $i < j$*, such that* $tr.i = n'$ *and* $tr.j = n$*, then one of the following two cases applies:*

1) there exists a CFG path from n' *to* n*, which is not recursion-free, or*

2) n *and* n' *are located in different branches of the CFG, attached to the same interleaving node or parallel composition node.*

Proof. We show the following: if the opposite of 1) holds, that is, if no recursion-free CFG path from n' to n exists, both nodes must not be connected by a CFG path *at all*, based on Lemma 5.3.4. From this, we deduce that the second case needs to apply.

Assume that there exists a CFG path π from n' to n, which is recursion-free. Let $\mathsf{n'} \in \mathbf{Ph}_2$ and $\mathsf{n} \in \mathbf{Ph}_1$. According to Lemma 5.3.4, it is impossible that π proceeds from $\mathbf{Ph}_2$ over $\mathbf{C}_1$ to $\mathbf{Ph}_1$ or directly from $\mathbf{Ph}_2$ to $\mathbf{Ph}_1$. Therefore, π has to proceed over $\mathbf{C}_2$ and $\mathbf{Ph}_3$ back to $\mathbf{Ph}_1$, which requires a recursion within π, contradiction. The other cases are similar.

Therefore, a CFG path from n' to n does not exist. We conclude the proof by applying Lemma 6.1.4 from [Brü08]: two events with a subsequent execution within $S.\texttt{main}$ require a CFG path from the first to the latter node or, if such a path does not exist, they have to be located in different branches of the CFG, attached to the same interleaving node or parallel composition node. □

Correctness of `Init`-restriction

Chapter 4, Definition 4.3.6, introduced the restriction of $S.\texttt{Init}$ to determine the initial state schemas of S_1 and S_2. In addition, Section 4.3.7 introduced an optimisation for the decomposition of a specification, allowing us to *neglect* certain initial data dependences when checking the correctness criterion **no crossing**.

In this section, we show that both, the definition and the optimisation, are correct in the following sense, where we let $V := S.V$, $V_1 := S_1.V$, $V_2 := S_2.V$ and $i \in \{1, 2\}$:

1.) for any state $s \in S.\texttt{State}$ such that $S.\texttt{Init}(s)$ holds, the restriction $s \restriction V_i \in S_i.\texttt{State}$ satisfies $S_i.\texttt{Init}$ and

2.) two states $s^i \in S_i.\texttt{State}$, for which $S_i.\texttt{Init}(s^i)$ holds, can appropriately be combined to a state $s \in S.\texttt{State}$ such that $S.\texttt{Init}(s)$ holds.

Before proving these particular properties, we introduce some notations. First, recall that the set $Atoms(S.\texttt{Init})$ denotes the set of all *atomic predicates* for the initial state schema:

$$\bigwedge_{a \in Atoms(S.\texttt{Init})} a = S.\texttt{Init}.$$

For any such predicate a, let $a[x/v]$ depict the predicate, resulting from replacing any free occurrence of the variable x in a with the value v of type t_x. Henceforth, if a is defined over a set of state variables $\{x_1,\ldots,x_n\}$ and $v_i : t_{x_i}$, we write

$$(v_1,\ldots,v_n) \models a \Leftrightarrow a[x_i/v_i] = \text{true}\,.$$

For instance, $(7,3) \models x > y$. Here, we assume an *ordering* on the state variables, such that a unique mapping $x_i \rightarrow v_i$ is indeed possible. Moreover, we write $s \models S.\texttt{Init}$ instead of $S.\texttt{Init}(s)$. Finally, let $\texttt{Init} \restriction V$ denote the $\texttt{Init}$-schema of a specification, restricting only variables from V.

The next lemma states the first of the two properties specified above:

Lemma 5.3.7. *(Correctness of* `Init`*-restriction, first part)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ *be the DG of a specification S, and let* $(\mathbf{C}_1, \mathbf{C}_2)$ *be a cut, separating the set V into* V_1 *and* V_2*, according to Definition 4.3.5. Then, for all states* $s \in S.\texttt{State}$*:*

$$s \models S.\texttt{Init} \Rightarrow (s \restriction V_1 \models S_1.\texttt{Init} \wedge s \restriction V_2 \models S_2.\texttt{Init}).$$

Proof. Assume $s \models S.\texttt{Init}$ for some $s \in S.\texttt{State}$. Let $(V \setminus V_1) = \{v_1,\ldots,v_n\}$, and let $V_1 = \{w_1,\ldots,w_m\}$. We have to show

$$s \restriction V_1 \models \exists v_1,\ldots,v_n \bullet S.\texttt{Init}$$

and

$$s \restriction V_2 \models \exists w_1,\ldots,w_m \bullet S.\texttt{Init}.$$

Recall that, in general, $V_1 \cap V_2 \neq \varnothing$, and thus, $(V \setminus V_1) \neq V_2$. Since

$$S.\texttt{Init} = \bigwedge_{a \in \mathit{Atoms}(S.\texttt{Init})} a,$$

$s \models S.\texttt{Init}$ is equivalent to $\forall\, a \in \mathit{Atoms}(S.\texttt{Init}) \bullet s \models a$. Let $\mathit{Free}(a)$ be the set of free state variables within a. Without loss of generality, let

$$\mathit{Free}(a) = \{x_1,\ldots,x_n,y_1,\ldots,y_m\}$$

for $x_i \in V_1$ and $y_j \in (V \setminus V_1)$. Then,

$$S_1.\texttt{Init}.a = \exists y_1,\ldots,y_m \bullet a(x_1,\ldots,x_n,y_1,\ldots,y_m)$$

and

$$S_2.\texttt{Init}.a = \exists x_1,\ldots,x_n \bullet a(x_1,\ldots,x_n,y_1,\ldots,y_m).$$

We deduce

$$\begin{aligned}
s \models a \;&\Leftrightarrow\; a[x_i/s.x_i][y_j/s.y_j] = \text{true}\\
&\Leftrightarrow\; s \restriction V_1 \models a[y_j/s.y_j] \text{ and } s \restriction V_2 \models a[x_i/s.x_i]\\
&\Rightarrow\; s \restriction V_1 \models \exists y_1,\ldots,y_m \bullet a \text{ and } s \restriction V_2 \models \exists x_1,\ldots,x_n \bullet a\\
&\Leftrightarrow\; s \restriction V_1 \models S_1.\texttt{Init}.a \text{ and } s \restriction V_2 \models S_2.\texttt{Init}.a. \qquad \square
\end{aligned}$$

For the second property, we have to consider the optimisation from Section 4.3.7: as already mentioned, we aim at neglecting several initial data dependences. These are the ones originating from an atomic predicate a solely referring to variables x, such that $\mathit{InitClos}(x) \subseteq (V_2 \setminus V_1)$ holds. It is reasonable to neglect those dependences, because all of these predicates remain unchanged within $S_2.\texttt{Init}$, which we show next. As additionally, state variables from $V_2 \setminus V_1$ are not modified within S_1 at all, the initial data dependences within $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$, originating from these predicates, no longer cause a violation of **no crossing**. Note that all remaining initial data dependences must *not* be neglected.

Lemma 5.3.8. *(Correctness of optimisation)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ be the DG of a specification S, and let $(\mathbf{C}_1, \mathbf{C}_2)$ be a cut, separating the set V into V_1 and V_2, according to Definition 4.3.5. In addition, let

$$\{y_1, \ldots, y_m\} = \{x \mid \mathit{InitClos}(x) \subseteq (V_2 \setminus V_1)\}.$$

Then, the initial state predicate of S restricted to $\{y_1, \ldots, y_m\}$ is equal to the initial state predicate of S_2 restricted to the same set, that is

$$S.\texttt{Init} \restriction \{y_1, \ldots, y_m\} = S_2.\texttt{Init} \restriction \{y_1, \ldots, y_m\}.$$

Proof. Let $\mathit{InitClos}(x) \subseteq (V_2 \setminus V_1)$. Then, for any atomic predicate $a \in \mathit{Atoms}(S.\texttt{Init})$, we get $\mathit{vars}(a) \subseteq (V_2 \setminus V_1)$. Based on

$$S_2.\texttt{Init} = \exists w_1, \ldots, w_m \bullet S.\texttt{Init},$$

for $V_1 = \{w_1, \ldots, w_m\}$, any of these atoms is preserved within $S_2.\texttt{Init}$. □

Next, we complement Lemma 5.3.7 by proving the second of the two properties, specified above: from two states $s^i \in S_i.\texttt{State}$ satisfying $S_i.\texttt{Init}$, we can construct $s \in \texttt{State}$ satisfying $S.\texttt{Init}$. For finally showing correctness of the optimisation, this construction necessarily needs to take the previously described subset of $(V_2 \setminus V_1)$ into account.

Lemma 5.3.9. *(Correctness of `Init`-restriction, second part)*
Let $\texttt{DG}_S = (N, \longrightarrow_{\texttt{DG}})$ be the DG of a specification S, and let $(\mathbf{C}_1, \mathbf{C}_2)$ be a cut, separating the set V into V_1 and V_2, according to Definition 4.3.5. Let $V_1 = \{x_1, \ldots, x_n\}$,

$$\{y_1, \ldots, y_m\} = \{x \mid \mathit{InitClos}(x) \subseteq (V_2 \setminus V_1)\} \text{ and}$$
$$\{z_1, \ldots, z_l\} = (V_2 \setminus V_1) \setminus \{y_1, \ldots, y_m\}.$$

For all states of S, we use the variable ordering $(x_1, \ldots, x_n, y_1, \ldots, y_m, z_1, \ldots, z_l)$. Let $s^i \in S_i.\texttt{State}$, such that $s^i \models S_i.\texttt{Init}$. Then, there exist $c_i : t_{z_i}$, $i \in \{1, \ldots, l\}$, such that for

$$s := (s^1.x_1, \ldots, s^1.x_n, s^2.y_1, \ldots, s^2.y_m, c_1, \ldots, c_l),$$

$s \models S.\texttt{Init}$.

Proof. Again, let $(V \setminus V_1) = \{v_1, \ldots, v_n\}$, and let $V_1 = \{w_1, \ldots, w_m\}$. Based on the definition of $S_i.\texttt{Init}$, we have

1) $(s^1.x_1, \ldots, s^1.x_n) \models \exists v_1, \ldots, v_n \bullet S.\texttt{Init}$ and

2) $(s^2.y_1, \ldots, s^2.y_m, s^2.z_1, \ldots, s^2.z_l) \models \exists w_1, \ldots, w_m \bullet \texttt{Init}$.

We need to show that there indeed exist c_i of type t_{z_i}, such that

$$S.\texttt{Init}[x_1/s^1.x_1] \ldots [x_n/s^1.x_n][y_1/s^2.y_1] \ldots [y_m/s^2.y_m][z_1/c_1] \ldots [z_l/c_l]$$

evaluates to true. First, for all $a \in \mathit{Atoms}(S.\texttt{Init})$:

$$\mathit{vars}(a) \cap \{x_1, \ldots, x_n, z_1, \ldots, z_l\} \cap \{y_1, \ldots, y_m\} = \varnothing.$$

This is based on Definition 4.3.26: assume the opposite, then there exists an atomic predicate from $S.\texttt{Init}$, containing a variable $y \in (V_2 \setminus V_1)$, such that $\mathit{InitClos}(y) \subseteq (V_2 \setminus V_1)$ holds. In addition, the predicate either refers to a variable $x \in V_1$ or to a variable $z \in (V_2 \setminus V_1)$, for which $\mathit{InitClos}(z) \nsubseteq (V_2 \setminus V_1)$. In both cases, we get a contradiction, as either x itself or some $z' \in \mathit{InitClos}(z)$ is an element of $(V_1 \cap \mathit{InitClos}(y))$. Therefore, any atomic predicate within $S.\texttt{Init}$ is either defined over a subset of $\{x_1, \ldots, x_n, z_1, \ldots, z_l\}$ or a subset of $\{y_1, \ldots, y_m\}$. s indeed satisfies $S.\texttt{Init}$: let $a \in \mathit{Atoms}(S.\texttt{Init})$ be defined over $\{x_1, \ldots, x_n, z_1, \ldots, z_l\}$. As

$$(s^1.x_1, \ldots, s^1.x_n) \models \exists v_1, \ldots, v_n \bullet S.\texttt{Init},$$

in particular,

$$(s^1.x_1, \ldots, s^1.x_n) \models \exists c_1, \ldots, c_n \bullet a$$

holds. Now assume that $a \in \mathit{Atoms}(S.\texttt{Init})$ is defined over $\{y_1, \ldots, y_m\}$.

$$(s^2.y_1, \ldots, s^2.y_m, s^2.z_1, \ldots, s^2.z_l) \models \exists w_1, \ldots, w_m \bullet \texttt{Init}$$

particularly implies that $(s^2.y_1, \ldots, s^2.y_m) \models a$. As

$$\mathit{vars}(a) \cap \{x_1, \ldots, x_n, z_1, \ldots, z_l\} \cap \{y_1, \ldots, y_m\} = \varnothing,$$

we conclude that $s \models S.\texttt{Init}$. □

Example 5.3.10. *Recall the initial state schema of the candy machine from Figure 2.3:*

$$\mathit{CandyMachine}.\texttt{Init} = (\mathit{sum} = 0) \land (\mathit{paid} = \langle\,\rangle) \land (\mathit{items} = \langle\,\rangle).$$

For the valid single cut $\mathbf{C} = \{\mathsf{switch}\}$*, we get sum, paid* $\in V_1$ *and items* $\in (V_2 \setminus V_1)$*. Obviously,* $\mathit{InitClos}(\mathit{items}) \subseteq (V_2 \setminus V_1)$*. Thus,*

$$S.\texttt{Init} \restriction \{\mathit{items}\} = S_2.\texttt{Init} \restriction \{\mathit{items}\} = (\mathit{items} = \langle\,\rangle).$$

In addition, any state $s = (s^1.\mathit{sum}, s^1.\mathit{paid}, s^2.\mathit{items})$*. such that* $s^i \models S_i.\texttt{Init}$*, yields* $s \models S.\texttt{Init}$.

Note, that the previous lemmas showed the correctness of the optimisation from Section 4.3.7 but not ultimately of the `Init`-restriction. It remains to be shown that the existential quantification does not violate the initial *correlation* between several state variables. For instance, one could assume that a split-up of a predicate $x \geq y$ into two predicates $\exists x \bullet x \geq y$ and $\exists y \bullet x \geq y$ may cause an observable difference between S and $S_1 \parallel_{E_C} S_2$. The correctness proof of the decomposition of the Object-Z part in Section 5.3.2 will show that this is *not* the case, mainly by using the correctness criterion **no crossing** with respect to initial data dependences.

5.3.2 Correctness of the Decomposition: Object-Z part

Next, we show correctness for the decomposition of $S.OZ$ by using the previous results of this section. We start by clarifying the restriction that traces of the Object-Z part need to adhere to the ones of the CSP part. Instead of showing that

$$S.OZ =_T (S_1.OZ \parallel_{E_C} S_2.OZ)[\![\mathtt{R}']\!]$$

holds, we show the weaker property

$$\begin{aligned} &\forall\, tr \text{ such that } (tr \rhd Op) \in traces(S.\mathtt{main}) \rhd Op \bullet \\ &tr \in traces(S.OZ) \Leftrightarrow tr \in traces((S_1.OZ \parallel_{E_C} S_2.OZ)[\![\mathtt{R}']\!]), \end{aligned} \tag{5.1}$$

It describes that any trace, for which we assume the ordering of operations to be determined by the CSP part, is an element of $traces(S.OZ)$, if, and only if, it is contained in $traces((S_1.OZ \parallel_{E_C} S_2.OZ)[\![\mathtt{R}']\!])$.

The crucial point considering the renaming is as follows: addressing parameters are not restricted by the Object-Z part, and we can entirely omit dealing with them. However, we have to consider the set of *transmission* parameters, as they are necessary to restore the original data flow within the decomposition. In correspondence to the correctness proof of the CSP part and for simplification, we omit denoting the additional parameters, which are introduced by the renaming relation.

We sketch the main strategy for showing Equation 5.1: for any trace of $S.OZ$, we define a trace of $S_1.OZ \parallel_{E_C} S_2.OZ$ and vice versa, such that both traces are equivalent. Recall Section 2.2.2 and the definition of $Traces(S.OZ)$: according to the LTS semantics from Definition 2.2.3, an Object-Z trace consists of an alternating sequence of states and events. As we ultimately aim at showing trace equivalence with respect to the CSP traces model, we solely have to require trace equivalence over the set $traces(OZ)$, the set of Object-Z traces projected on *events*. In particular, we will observe that the trace equivalence with respect to $Traces(S.OZ)$ cannot be shown: some state variables do not need to have corresponding values within $\pi \in Traces(S.OZ)$ and its analogon π_i within $Traces(S_1.OZ \parallel_{\{E_C\}} S_2.OZ)$.

In general, the decomposition of a specification eliminates a subset of the original set of operations. Therefore, a trace of $\pi_i \in S_i.OZ$ is a *projection* of a trace of $\pi \in S.OZ$. In order to simplify reasoning about their correspondence and the usage of indices, we assume an event `noev`, depicting *stuttering* [CGP99] in π_i: it substitutes for any event e of π, not

occurring in π_i. Thus, both traces have a corresponding length. Furthermore, for any `noev`-step in a trace, the succeeding state is identical to the one before `noev`, that is

$$s \xrightarrow{\texttt{noev}} s' \Rightarrow s = s'.$$

When dealing with traces $\pi \in \mathit{Traces}(S.OZ)$, we use $s_i := \pi[i]$ to denote the ith *state* of the trace π and $e_i := \pi.i$ to denote its ith *event*. Furthermore, we let $e_i = op_i.in_i.sim_i.out_i$. We proceed accordingly for $\pi_i \in \mathit{Traces}(S_i.OZ)$, where we use an additional top index. Based on the usage of `noev`, we can always refer to corresponding positions within π and π_i. Summarising, traces are referred to as

$$\begin{aligned} \pi &= \langle s_0, e_0, s_1, e_1, \dots \rangle, \\ \pi_1 &= \langle s_0^1, e_0^1, s_1^1, e_1^1, \dots \rangle \text{ and} \\ \pi_2 &= \langle s_0^2, e_0^2, s_1^2, e_1^2, \dots \rangle. \end{aligned}$$

Before carrying out the actual proof, we illustrate our general strategy and the possible inconsistencies in the state space valuations by an example.

Example 5.3.11. *Consider the extended number swapper from Figure 4.26 and its decomposition from Figures 4.27 and 4.28. The following table compares three valid traces $\pi \in \mathit{Traces}(\mathit{Swapper.OZ})$, $\pi_1 \in \mathit{Traces}(\mathit{Swapper}_1.OZ)$ and $\pi_2 \in \mathit{Traces}(\mathit{Swapper}_2.OZ)$. Here, `"_"` denotes an arbitrary value. We choose the value 2 for the input parameter of the operation input.*

π		π_1		π_2	
	$\langle$		$\langle$		$\langle$
s_0 :	$(a = 1, b = \_, \mathit{tmp} = \_)$,	s_0^1 :	$(b = \_, \mathit{tmp} = \_)$,	s_0^2 :	$(a = 1, b = \_, \mathit{tmp} = \_)$,
e_0 :	$\mathit{input}.2$,	e_0^1 :	$\mathit{input}.2$,	e_0^2 :	`noev`,
s_1 :	$(a = 1, b = 2, \mathit{tmp} = \_)$,	s_1^1 :	$(b = 2, \mathit{tmp} = \_)$,	s_1^2 :	$(a = 1, \mathbf{b = \_}, \mathit{tmp} = \_)$,
e_1 :	$\mathit{store\_b}$,	e_1^1 :	$\mathit{store\_b}.2$,	e_1^2 :	$\mathit{store\_b}.2$,
s_2 :	$(a = 1, b = 2, \mathit{tmp} = 2)$,	s_2^1 :	$(b = 2, \mathit{tmp} = 2)$,	s_2^2 :	$(a = 1, \mathbf{b = \_}, \mathit{tmp} = 2)$,
e_2 :	$\mathit{move\_a}$,	e_2^1 :	`noev`,	e_2^2 :	$\mathit{move\_a}$,
s_3 :	$(a = 1, b = 1, \mathit{tmp} = 2)$,	s_3^1 :	$(\mathbf{b = 2}, \mathit{tmp} = 2)$,	s_3^2 :	$(a = 1, b = 1, \mathit{tmp} = 2)$
e_3 :	$\mathit{move\_b}$,	e_3^1 :	`noev`,	e_3^2 :	$\mathit{move\_b}$,
s_4 :	$(a = 2, b = 1, \mathit{tmp} = 2)$,	s_4^1 :	$(\mathbf{b = 2}, \mathit{tmp} = 2)$,	s_4^2 :	$(a = 2, b = 1, \mathit{tmp} = 2)$
e_4 :	$\mathit{result}.1$,	e_4^1 :	$\mathit{result}.\_$,	e_4^2 :	$\mathit{result}.1$,
s_5 :	$(a = 2, b = 1, \mathit{tmp} = 2)$	s_5^1 :	$(\mathbf{b = 2}, \mathit{tmp} = 2)$	s_5^2 :	$(a = 2, b = 1, \mathit{tmp} = 2)$
	$\rangle$		$\rangle$		$\rangle$

As move_a and move_b do not occur in S_1 and as input is not represented in S_2, these events are replaced by `noev` *within π_1 and π_2, respectively.*

Within the decomposition, the parameter value for result is solely determined by S_2.result. Therefore, the event has an arbitrary parameter value within π_1, the synchronisation ensures that exclusively the value from π_2 is possible.

Five cells are highlighted in gray: for these states, the value for b is inconsistent *between either π and π_1 or between π and π_2. However, these inconsistent values do not influence the equivalence between $\pi \restriction E_S$ and the joint execution of $\pi_1 \restriction E_{S'}$ and $\pi_2 \restriction E_{S'}$: in both cases, we get*

$$\langle input.2, store\_b[.2], move\_a, move\_b, result.1 \rangle,$$

where store_b receives an additional transmission parameter within the parallel composition of $\pi_1 \restriction E_{S'}$ and $\pi_2 \restriction E_{S'}$.

As we will show in this section, inconsistent values for some state variables never affect the trace equivalence in the CSP traces model.

The decomposition of S leads to a partitioning of V into V_1 and V_2. In the remainder of this proof, we need to distinguish between four different sets of state variables:

$V_1 \setminus V_2$**:** the set of state variables solely represented in S_1,

$V_2 \setminus V_1$**:** the set of state variables solely represented in S_2,

$CV := CV_1 \cup CV_2$**:** the set of *cut variables*, according to Definition 4.3.9,

$\overline{CV} := (V_1 \cap V_2) \setminus CV$**:** the set of remaining shared state variables.

The latter set can be characterised as the set of state variables occurring in both, S_1 and S_2, which are either not modified within $\mathbf{C}_1$ ($\mathbf{C}_2$) or which do not influence $\mathbf{Ph}_2$ ($\mathbf{Ph}_3 \cup \mathbf{Ph}_1$). As we will see later, this is the sole set of state variables, for which values in π and π_i are possibly inconsistent.

We start the proof with the forward direction of Equation 5.1.

Left-to-Right Implication

Let $tr \in traces(S.OZ)$, such that $(tr \rhd Op) \in traces(S.\mathtt{main}) \rhd Op$. We have to show

$$tr \in traces((S_1.OZ \parallel_{E_C} S_2.OZ)[\![\mathtt{R}']\!]).$$

Let $\pi = \langle s_0, e_0, s_1, e_1, \dots \rangle \in Traces(S.OZ)$, such that $\pi \restriction E_S = tr$ holds. Recall that $e_i = op_i.in_i.sim_i.out_i$. We proceed in three steps: based upon π, we define two traces

$$\pi_i = \langle s_0^i, e_0^i, s_1^i, e_1^i, \dots \rangle \textbf{ (Step 1)},$$

with possibly $e_j^i = \mathtt{noev}$ for some events. Next, we inductively show

$$\pi_i \in Traces(S_i.OZ) \textbf{ (Step 2)}.$$

Finally, we deduce

$$tr = \pi \restriction E_S \in (\pi_1 \restriction E_{S'} \parallel_{E_C} \pi_2 \restriction E_{S'})[\![\mathtt{R}']\!] \textbf{ (Step 3)}.$$

Here, we refer to the definition of [Ros98] for the parallel composition of two *traces*.

Step 1: For the definition of $\pi_i \in \mathit{Traces}(S_i.OZ)$, we need to consider events and states. Obviously, as we aim at showing equivalence within the CSP traces model, the definition for π_i on events needs to match with the trace *tr*, except for events replaced by `noev`. This gives rise to the following definition:[4]

$$e_j^1 := \begin{cases} e_j.t, & \mathsf{op}_j \notin \mathbf{Ph}_2, \\ \texttt{noev}, & \mathit{otherwise}, \end{cases} \qquad (5.2)$$

$$e_j^2 := \begin{cases} e_j.t, & \mathsf{op}_j \notin (\mathbf{Ph}_1 \cup \mathbf{Ph}_3), \\ \texttt{noev}, & \mathit{otherwise}, \end{cases}$$

where *t* denotes the values for the additional transmission parameters. Note that these are *uniquely* defined, based on Definition 4.3.10.

The definition of the states is more complicated. The initial states of π_1 and π_2 are simply defined as the restrictions of s_0 on V_1 and V_2, respectively:

$$s_0^1 := s_0 \restriction V_1 \text{ and } s_0^2 := s_0 \restriction V_2. \qquad (5.3)$$

Next, we define s_k^i for $k \geq 1$. The states for π_i mostly correspond to the states of π, restricted to the remaining set of state variables. Therefore, in most cases, we simply set $s_k^i := s_k \restriction V_i$. However, in some cases, the state valuations do not match. This is the case, if some modification of a state variable within π is not represented in π_i, thus causing an inconsistency between the values of the respective state variable, which is possibly preserved afterwards.

Precisely, there are three different cases, for which the value for a state variable *x* within π_i must not be modified, that is $s_k^i := s_{k-1}^i$, instead of $s_k^i := s_k$.

(1) In Example 5.3.11, consider the transition $s_2 \xrightarrow{move\_a} s_3$ within π. The event is replaced by `noev` within π_1. As the modification for *b* to the value 1 gets lost within the trace for $S_1.OZ$, setting $s_3^1.b$ to $s_3.b$ would contradict $\pi_1 \in \mathit{Traces}(S_1.OZ)$ and is therefore unreasonable. Instead, we have to define $s_3^1.b := s_2^1.b$ – the original value is preserved in case that an event is replaced by `noev` within π_i.

(2) Now consider the transition $s_4 \xrightarrow{result.1} s_5$. As the value for *b* is not modified within *result*, the equation $s_5.b = s_4.b = 1$ holds. The corresponding call of *result* within π_1 does not change *b* as well. Therefore, we again need to *preserve* the (inconsistent) value $s_4^1.b = 2$.

(3) For the final case, assume that in the example, the predicate part of the operation *store_b* additionally comprises a modification of *b*. As $\mathit{store\_b} \in \mathit{Op}_{\mathbf{C}_1}$ and as *b* is not a cut variable, this modification is solely conducted in S_1. The old value for *b* needs to be preserved within π_2.

[4] Here, we refer to the specific occurrence of the DG node op_j corresponding to the uniquely related occurrence op_j of the operation *op*.

The previous considerations motivate the following definition. Let $i, j \in \{1, 2\}$. For any $k \geq 1$ and $x \in V_i$, we define the value of $s_k^i.x$ within π_i as:

$$s_k^i.x := \begin{cases} s_{k-1}^i.x, & e_{k-1}^i = \texttt{noev} \text{ or} \\ & (x \in (V_1 \cap V_2) \text{ and } x \notin \mathit{ref}(op_{k-1}) \text{ and } x \notin \mathit{mod}(op_{k-1})) \text{ or} \\ & (x \notin \mathit{CV}_j \text{ and } x \in \mathit{mod}(op_{k-1}) \text{ and } e_{k-1}^i \in E_{\mathbf{C}_j}, \text{ for } j \neq i), \\ s_k.x, & \text{otherwise.} \end{cases} \tag{5.4}$$

The three cases for $s_{k-1}^i.x$ correspond to the previous explanations. The definition can be summarised as follows: any modification of x within π_i results in corresponding values $s_k.x$ and $s_k^i.x$. In any other case, the pre-state value for the variables is kept.

Step 2: In a next step, we have to show $\pi_i \in \mathit{Traces}(S_i.OZ)$. Based on the previous definition, the values for the state variables within π and π_i are possibly diverse. However, the following, crucial lemma shows that if some state variable x is *referenced* by the operation op_k^i, the pre-state value of x is identical in π and π_i, that is, $s_k.x = s_k^i.x$ holds:

Lemma 5.3.12. *(Equality of values for referenced state variables, left-to-right)*
Let $\pi \in \mathit{Traces}(S.OZ)$, and let π_i be defined according to Equations 5.2, 5.3 and 5.4. Then:

$$\forall\, n \geq 0, \forall\, x \in V_i, \forall\, op^i \in S_i.Op \bullet x \in \mathit{ref}(op_n^i) \Rightarrow s_n.x = s_n^i.x.$$

Proof. The property obviously holds for $n = 0$, as $s_0^i = s_0 \restriction V_i$. Let $n > 0$, $x \in \mathit{ref}(op_n^1)$ and thus, $x \in \mathit{ref}(op_n)$, for some $x \in S.\texttt{State}$. Assume that $s_n.x \neq s_n^1.x$. The case $i = 2$ is analogous. Initially, $s_0.x = s_0^1.x$ holds. Therefore, there exist some $0 \leq k < n$ and $op_k.par_k$, such that $x \in \mathit{mod}(op_k)$, but $x \notin \mathit{mod}(op_k^1)$. This is due to Equation 5.4: *if* some state variable is modified within π_1, the modification is *identical* for π and π_1. Assume that k is the *latest* such position, that is, there is no further modification of x between s_{k+1} and s_n. For the transition sequence $\langle e_k, \ldots, e_n \rangle$, we apply Lemma 6.1.4 from [Brü08]: either, there exists a CFG path, connecting both corresponding DG-nodes e_k and e_n or they are located in different CFG branches, attached to the same interleaving node or parallel composition node. As x is not modified in between, we deduce that there either exists a direct data dependence (in the first case) or an interference data dependence (in the latter case) from e_k to e_n.

This particular dependence will now be used to deduce a contradiction. Here, several different cases have to be considered. Figure 5.11 illustrates the current situation, where the DG nodes corresponding to e_k and e_n are connected by a data dependence.

As $x \in \mathit{mod}(op_k)$ and $s_{k+1}.x \neq s_{k+1}^1.x$, either $e_k^1 = \texttt{noev}$ or $e_k^1 \in E_{\mathbf{C}_2}$ according to Equation 5.4.

$e_k^1 = \texttt{noev}$**:** In this case, the corresponding DG node e_k is an element of $\mathbf{Ph}_2$, as only operations corresponding to nodes from $\mathbf{Ph}_2$ are eliminated from S_1. We need to consider four cases for e_n:

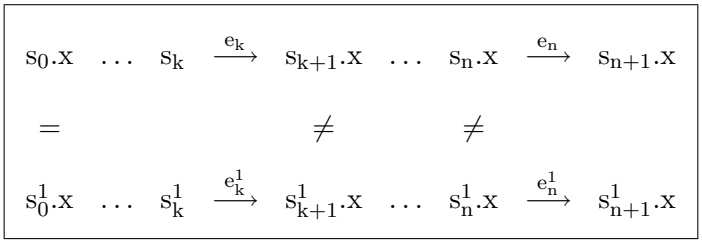


Figure 5.11: Illustration of Lemma 5.3.12

$\mathsf{e}_n \in (\mathbf{Ph}_1 \cup \mathbf{Ph}_3)$**:** The data dependence from from e_k to e_n violates condition **no crossing**. ✓

$\mathsf{e}_n \in \mathbf{C}_1$**:** The data dependence from e_k to e_n violates Lemma 5.3.3 (and particularly condition **no reaching back**). ✓

$\mathsf{e}_n \in \mathbf{C}_2$**:** In this case, the original predicate part of op_n is eliminated for the definition of $S_1.op_n$ and solely replaced by the transmission parameter predicates. Thus, $x \in ref(op_n^1)$ is impossible. ✓

$\mathsf{e}_n \in \mathbf{Ph}_2$**:** op_n is eliminated from S_1 in its entirety and again, $x \in ref(op_n^1)$ is impossible. ✓

$e_k^1 \in E_{\mathbf{C}_2}$**:** In particular, $x \notin CV_2$ needs to hold, according to the third case of Equation 5.4.

$\mathsf{e}_n \in (\mathbf{Ph}_1 \cup \mathbf{Ph}_3)$**:** The data dependence e_k to e_n causes x to be contained in the set of cut variables of $E_{\mathbf{C}_2}$, contradicting $x \notin CV_2$. ✓

$\mathsf{e}_n \in \mathbf{C}_1$**:** According to the previous case, as the referencing of $x \in \mathbf{C}_1$ still causes x to be a cut variable. ✓

$\mathsf{e}_n \in \mathbf{C}_2$**:** Analogous to the corresponding case for $e_k^1 = \texttt{noev}$. ✓

$\mathsf{e}_n \in \mathbf{Ph}_2$**:** Analogous to the corresponding case for $e_k^1 = \texttt{noev}$. ✓ □

Next, we show a corresponding lemma for the values of *local* state variables:

Lemma 5.3.13. *(Equality of values for local state variables, left-to-right)*
Let $\pi \in Traces(S.OZ)$, and let π_i be defined according to Equations 5.2, 5.3 and 5.4. Let $j \neq i$. Then:

$$\forall n \geq 0, \forall x \in V_i, \forall op^i \in S_i.Op \bullet x \in (V_i \setminus V_j) \Rightarrow s_n.x = s_n^i.x.$$

Proof. Again, the property holds for $n = 0$. As $x \in (V_i \setminus V_j)$, none of the first three cases from Equation 5.4 ever applies: an event replaced by `noev` within π_i is solely represented in S_j and never refers to local variables from S_i. The remaining two cases only apply for $x \in (V_1 \cap V_2)$. Therefore, $s_n.x = s_n^i.x$ always holds. □

The previous lemmas will be used throughout the following theorem, which shows that π_i is indeed an element of $Traces(S_i.OZ)$:

Theorem 5.3.14. *(Correctness of the decomposition: Object-Z part, first part)*
Let π_i be defined according to Equations 5.2, 5.3 and 5.4. Then, $\pi_i \in Traces(S_i.OZ)$.

Proof. The proof is conducted by induction on the length of π_i. In the induction base, we show $s_0^i \models S_i.\texttt{Init}$. In the induction step, based on the assumption that $\langle s_0^i, e_0^i, \ldots, s_k^i \rangle$ is an element of $Traces(S_i.OZ)$, we show

$$s_k^i \xrightarrow{e_k^i} s_{k+1}^i,$$

where s_{k+1}^i complies to the conditions of Equation 5.4.

Induction Base:

$s_0^i \models S_i.\texttt{Init}$ directly follows from Equation 5.3 and Lemma 5.3.7.

Induction Step:

We start by considering the *guard* of op_k^i, which needs to be satisfied. Furthermore, the operation must be executable with parameter values corresponding to e_k. Finally, performing op_k^i needs to allow for the successor state s_{k+1}^i to comply to the conditions of Equation 5.4.

In the following proof, we use the predicate interpretation of an `enable-` and `effect-`schema in terms of Z:

$$s \xrightarrow{op.in.sim.out} s' \Leftrightarrow (\texttt{enable\_}op(s, in, sim) \wedge \texttt{effect\_}op(s, in, sim, out, s'))$$

Furthermore, we write $s \oplus t$ to denote that the *state t overrides the state s*. Precisely, for s defined over V and t defined over a subset V', let $s \oplus t$ be defined over V as follows:

$$(s \oplus t).x := \begin{cases} t.x, & x \in V', \\ s.x, & \text{otherwise.} \end{cases}$$

Let $e_k = op_k.in_k.sim_k.out_k$. By assumption,

$$\texttt{enable\_}op_k(s_k, in_k, sim_k) \wedge \texttt{effect\_}op_k(s_k, in_k, sim_k, out_k, s_{k+1})$$

holds. Besides, by using Lemma 5.3.12, we know that $s_k.x = s_k^i.x$ for all *referenced* variables within op_k^i.

Guard is Satisfied: We need to show $\texttt{enable\_}op_k^i(s_k^i, in_k, sim_k)$. For $op_k^i = \texttt{noev}$, there is obviously nothing to show. Moreover, if $op_k^i \in Op_{\mathbf{C}_j}$ for $j \neq i$, then $\texttt{enable\_}op_k^i =$ true. In any other case, $\texttt{enable\_}op_k^i = \texttt{enable\_}op_k$ holds, according to Definition 4.3.10. We deduce:

$$
\begin{aligned}
& \texttt{enable\_}op_k(s_k, in_k, sim_k) \\
\overset{(1)}{\Rightarrow}\ & \texttt{enable\_}op_k^i(s_k \restriction V_i, in_k, sim_k) \\
\overset{(2)}{\Rightarrow}\ & \texttt{enable\_}op_k^i((s_k \restriction V_i) \oplus (s_k^i \restriction ref(op_k)), in_k, sim_k) \\
\overset{(3)}{\Rightarrow}\ & \texttt{enable\_}op_k^i(s_k^i, in_k, sim_k).
\end{aligned}
$$

Implication (1) is due to $\texttt{enable\_}op_k^i = \texttt{enable\_}op_k$ and the fact that any operation from S_i solely refers to variables from V_i. Implication (2) is due to $s_k = s_k^i$ on $ref(op_k)$ (induction hypothesis and Lemma 5.3.12). The last implication, (3), follows by the fact that non-referenced variables within $\texttt{enable\_}op_k$ do not affect the truth-value of the associated predicate.

Operation is Executable with Compatible Successor State: We will now show

$$\texttt{effect\_}op_k^i(s_k^i, in_k, sim_k, out_k, s_{k+1}^i)$$

by distinguishing the three cases

1. $op_k^i = \texttt{noev}$,
2. $op_k^i \in Op_{\mathbf{C}_j}$ for $i \neq j$ and
3. all remaining possibilities.

$op_k^i = \texttt{noev}$**:** Again, noev does not pose a problem, as in this case, $s_{k+1}^i = s_{k+1}$, corresponding to Equation 5.4.

$op_k^i \in Op_{\mathbf{C}_j}$ **for** $i \neq j$**:** In this case, the predicate part of op_k is replaced by $\bigwedge_{w \in CV_j} w' = tr_w?$ within op_k^i. We deduce:

$$
\begin{aligned}
& \texttt{effect\_}op_k(s_k, in_k, sim_k, out_k, s_{k+1}) \\
\overset{(1)}{\Rightarrow}\ & \texttt{effect\_}op_k^i(s_k \restriction V_i, in_k, sim_k, out_k, (s_k \restriction V_i) \oplus (s_{k+1} \restriction CV_j)) \\
\overset{(2)}{\Rightarrow}\ & \texttt{effect\_}op_k^i((s_k \restriction V_i) \oplus s', in_k, sim_k, out_k, ((s_k \restriction V_i) \oplus s') \oplus (s_{k+1} \restriction CV_j)) \\
\overset{(3)}{\Rightarrow}\ & \texttt{effect\_}op_k^i(s_k^i, in_k, sim_k, out_k, s_k^i \oplus (s_{k+1} \restriction CV_j)),
\end{aligned}
$$

for

$$s' := (s_k^i \restriction (ref(op_k) \cup V_i \setminus V_j)).$$

The first implication is based on the fact that solely cut variables are modified by op_k^i, that is, the pre-state value needs to be kept for all remaining variables. Moreover, as the value for the output parameters are not restricted within op_k^i, the output value out_k can indeed be used. Implication (2) follows by $s_k.x = s_k^i.x$ on $ref(op_k)$ and all local variables (induction hypothesis, Lemma 5.3.12 and Lemma 5.3.13). As all other variables do not affect the execution of the operation, implication (3) is immediate.

The state $s_{k+1}^i = s_k^i \oplus (s_{k+1} \restriction CV_j)$ satisfies the conditions of Equation 5.4 for any $x \in V$:

$x \notin (V_1 \cap V_2)$**:** Impossible, as for $op_k^i \in Op_{\mathbf{C}_j}$, all variables occurring within op_k^i are shared variables. ✓

$x \in CV_j$**:** Here, $s_{k+1}^i = s_{k+1}$, according to Equation 5.4. ✓

$x \in (V_1 \cap V_2) \setminus CV_j$**:** For $x \in mod(op_k)$, we get $s_{k+1}^i = s_k^i$, according to Equation 5.4, third case. Otherwise, the variable is not modified within op_k. If it is referenced, $s_k = s_k^i$, which is preserved by the operation and thus, $s_{k+1}^i = s_{k+1}$, again matching with Equation 5.4. Otherwise, $x \notin (mod(op_k) \cup ref(op_k))$, and $s_{k+1}^i = s_k^i$ matches with Equation 5.4, second case. ✓

All remaining cases: Now, $\texttt{effect\_}op_k^i = \texttt{effect\_}op_k$ holds according to Definition 4.3.10, and we get:

$$\begin{aligned}
& \texttt{effect_}op_k(s_k, in_k, sim_k, out_k, s_{k+1}) \\
\overset{(1)}{\Rightarrow}\ & \texttt{effect_}op_k^i(s_k \restriction V_i, in_k, sim_k, out_k, s_{k+1} \restriction V_i) \\
\overset{(2)}{\Rightarrow}\ & \texttt{effect_}op_k^i((s_k \restriction V_i) \oplus s', in_k, sim_k, out_k, s_{k+1} \restriction V_i) \\
\overset{(3)}{\Rightarrow}\ & \texttt{effect_}op_k^i(s_k^i, in_k, sim_k, out_k, (s_{k+1} \restriction V_i) \oplus (s_k^i \restriction X)),
\end{aligned}$$

for $i \neq j$,

$$s' := (s_k^i \restriction (ref(op_k) \cup V_i \setminus V_j)),$$

and

$$X := (V_1 \cap V_2) \setminus (mod(op_k) \cup ref(op_k)).$$

For implication (1), we use $\texttt{effect\_}op_k^i = \texttt{effect\_}op_k$. Implication (2) follows by $s_k.x = s_k^i.x$ on $ref(op_k)$ and all local variables (induction hypothesis, Lemma 5.3.12 and Lemma 5.3.13).

For the final implication, (3), we use $\texttt{effect\_}op_k^i = \texttt{effect\_}op_k$ and $s_k.x = s_k^i.x$, yielding that any *modification* of a variable within op_k is correspondingly possible within op_k^i. We are left to deal with non-modified variables: for these, as all local state variables and referenced state variables have consistent values in the pre-state, they have consistent values in the post state as well. The *remaining* state variables are exactly those described by the set X: variables neither modified nor referenced within op_k, which are not local to S_i. For these, the pre-state value must be preserved within the post state.

Finally, the state $(s_{k+1} \restriction V_i) \oplus (s_k^i \restriction X)$ indeed satisfies the conditions of Equation 5.4 (where only the second and fourth case can apply), as for any state variable, the pre-state value is kept for $(V_1 \cap V_2) \setminus (ref(op_k) \cup mod(op_k))$ and otherwise, the value from s_{k+1} is used. □

Step 3: So far, we constructed two traces π_i out of $\pi \in Traces(S.OZ)$ for which we showed $\pi_i \in Traces(S_i.OZ)$. It remains to be shown that $tr = \pi \restriction E_S \in (\pi_1 \restriction E_{S'} \parallel_{E_\mathbf{C}} \pi_2 \restriction E_{S'})[\![\mathrm{R}']\!]$

holds. This is an immediate deduction due to $\pi_i \restriction E_{S_i} = tr \restriction E_{S_i}$ modulo renaming and the definition for the parallel composition of two traces ([Ros98]). This completes the proof of the left-to-right implication.

Right-to-Left Implication

Let $tr \in traces((S_1.OZ \parallel_{E_C} S_2.OZ)[\![R']\!])$, such that $(tr \rhd Op) \in traces(S.\texttt{main}) \rhd Op$.[5] We have to show $tr \in traces(S.OZ)$. Based on $tr \in traces((S_1.OZ \parallel_{E_C} S_2.OZ)[\![R']\!])$, there exist $\pi_i \in Traces(S_i.OZ)$, such that $tr \in (\pi_1 \restriction E_{S'} \parallel_{E_C} \pi_2 \restriction E_{S'})[\![R']\!]$.

Again, we proceed in three steps: we define a trace

$$\pi = \langle s_0, e_0, s_1, e_1, \dots \rangle$$

out of $\pi_i = \langle s_0^i, e_0^i, s_1^i, e_1^i, \dots \rangle$ (**Step 1**). In **(Step 2)**, we inductively show $\pi \in Traces(S.OZ)$. Finally, we deduce $tr = \pi \restriction E_S \in traces(S.OZ)$ (**Step 3**).

Step 1: For the definition of π on events, we obviously choose the events from tr:

$$e_j := tr.j. \tag{5.5}$$

For the definition of the states of tr, we start by defining s_0. Here, we use the result from Lemma 5.3.9: in the following, let $V_1 = \{x_1, \dots, x_n\}$,

$$V_Y := \{y_1, \dots, y_m\} = \{x \mid InitClos(x) \subseteq (V_2 \setminus V_1)\}$$

and

$$V_Z := \{z_1, \dots, z_l\} = (V_2 \setminus V_1) \setminus \{y_1, \dots, y_m\}.$$

Furthermore, let $c_1, \dots, c_l$, with $c_i : t_{z_i}$, such that

$$S.\texttt{Init}[x_1/s_0^1.x_1] \dots [x_n/s_0^1.x_n][y_1/s_0^2.y_1] \dots [y_m/s_0^2.y_m][z_1/c_1] \dots [z_l/c_l]$$

holds. Then:

$$s_0 := (s_0^1.x_1, \dots, s_0^1.x_n, s_0^2.y_1, \dots, s_0^2.y_m, c_1, \dots, c_l). \tag{5.6}$$

Note that we can freely choose *any* values c_i for z_i, as long as they extend $s_0^1.x_j$ and $s_0^2.y_k$ to a valid initial valuation. Lemma 5.3.9 showed that such values for z_i indeed exist. Intuitively, the freedom of choice is substantiated by the fact that for the set V_Z, the initial values within $S_2.\texttt{Init}$ are *irrelevant* for the specification S_2: in case that any such variable is referenced, it must have been modified before, as otherwise, an initial data dependence would violate the condition **no crossing**. Thus, these values never become relevant within S_2, and we can safely refrain from using them within s_0.

[5] Based on the correctness for the CSP part from Section 5.2, tr can equally refer to both, $traces(S.\texttt{main})$ or $traces((S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main})[\![R']\!])$.

The definition for s_k, $k \geq 1$, is given next:

$$s_k.x \quad := \quad \begin{cases} s_k^1.x, & x \in (V_1 \setminus V_2), \\ s_k^2.x, & x \in (V_2 \setminus V_1), \\ s_k^1.x, & x \in (V_1 \cap V_2),\ x \in (mod(op_{k-1}^1) \cap mod(op_{k-1}^2)), \\ s_k^1.x, & x \in (V_1 \cap V_2),\ x \in (mod(op_{k-1}^1) \setminus mod(op_{k-1}^2)), \\ s_k^2.x, & x \in (V_1 \cap V_2),\ x \in (mod(op_{k-1}^2) \setminus mod(op_{k-1}^1)), \\ s_{k-1}.x, & x \in (V_1 \cap V_2),\ x \notin (mod(op_{k-1}^1) \cup mod(op_{k-1}^2)). \end{cases} \tag{5.7}$$

Summarising, for state variables local to S_i, we choose the value of s_k^i. For shared variables, we adopt modifications from the respective traces and keep the pre-state value, if no modification is conducted. If a variable is modified in *both* traces, the modification must be corresponding. This is based on the usage of transmission parameters, ensuring that shared state variables must not distinctly be modified by the same operation. Thus, for the third case, we could equally define $s_k.x := s_k^2.x$.

Step 2: In accordance with the left-to-right implication, we show a property describing that state variables *referenced* by an operation op_k^i always have identical values within π and π_i:

Lemma 5.3.15. *(Equality of values for referenced state variables, right-to-left)*
Let $\pi_i \in Traces(S_i.OZ)$, and let π be defined according to Equations 5.5, 5.6 and 5.7. Then:

$$\forall\, n \geq 0, \forall\, x \in V_i, \forall\, op^i \in S_i.Op \bullet x \in ref(op_n^i) \Rightarrow s_n.x = s_n^i.x.$$

Proof. We first show that the property holds for $n = 0$. For the sets V_1 and V_Y, the states s_0 and s_0^i are identically defined. This is not the case for the set V_Z. However, $z \in ref(op_0^i)$ would yield that $e_0^i \in E_2$, as the set V_Z solely comprises variables local to S_2. This is impossible: any event, which can initially be executed within a trace of $S.\texttt{main}$, is an element of E_{S_1}. Otherwise, the corresponding DG-node e_0^i would violate the correctness criterion **disjointness**, based on $\mathsf{e}_0^i \in \mathbf{Ph}_2 \cap (\mathbf{Ph}_1 \cup \mathbf{C}_1)$.

Let $n > 0$, $x \in ref(op_n^i)$ and thus, $x \in ref(op_n)$ for some $x \in S.\texttt{State}$. Based on Equation 5.7, any modification conducted within π_i is identical within π. This allows us to apply the same ideas from Lemma 5.3.12 for op_n^1. In particular, if $x \in (mod(op_k) \setminus mod(op_k^1))$ for some op_k, either $op_k^1 = \texttt{noev}$ or it is an element of $E_{\mathbf{C}_2}$, resulting in the exact same case differentiation as in Lemma 5.3.12.

For op_n^2, we have to consider one additional case: for $x \in V_Z$, we cannot assume $s_0.x = s_0^2.x$. If x is modified somewhere in π_2, the modification is identical to the one in π, and we reside in the previous case. Now assume that $x \in ref(op_n)$, and x is never modified in π_2. In this case, there exists an initial data dependence from $S.\texttt{Init}$ to the corresponding DG node e_n. Since op_n^2 references $x \in V_Z$, op_n is an element of Op_2 and thus, e_n is an element of $\mathbf{Ph}_2$. This yields a contradiction, as the connecting initial data dependence violates **no crossing**.

Figure 5.12 illustrates the proof idea of the lemma. Here, $(*)$ denotes that $\mathrm{s_0.x} = \mathrm{s_0^2.x}$ only holds for $V \setminus V_Z$.

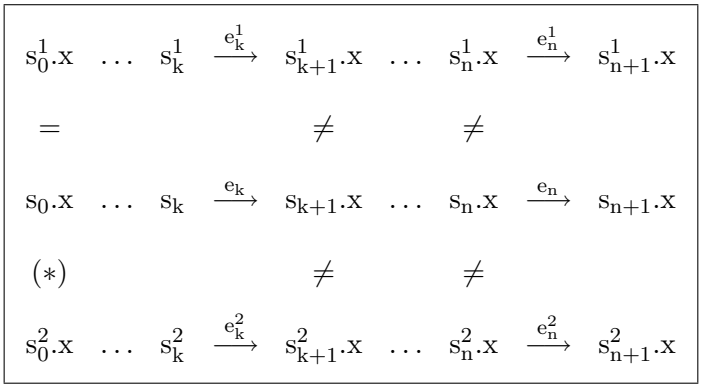


Figure 5.12: Illustration of Lemma 5.3.15

The corresponding lemma for local state variables is immediate:

Lemma 5.3.16. *(Equality of values for local state variables, right-to-left)*
Let $\pi_i \in Traces(S_i.OZ)$, and let π be defined according to Equations 5.5, 5.6 and 5.7. Let $j \neq i$. Then:

$$\forall n \geq 1, \forall x \in V_i, \forall op^i \in S_i.Op \bullet x \in (V_i \setminus V_j) \Rightarrow s_n.x = s_n^i.x.$$

Proof. The property holds based on Equation 5.7. □

Note that the previous property does *not* hold for $n = 0$, as the initial states do not correspond on the set $\{z_1, \ldots, z_l\}$. Next, we show that π is an element of $Traces(S.OZ)$:

Theorem 5.3.17. *(Correctness of the decomposition: Object-Z part, second part)*
Let π be defined according to Equations 5.5, 5.6 and 5.7. Then, $\pi \in Traces(S.OZ)$.

Proof. Again, the proof is conducted by induction on the length of π.

Induction Base:

$s_0 \models S.\texttt{Init}$ directly follows by Equation 5.6 and Lemma 5.3.9.

Induction Step:

Again, let $e_k = op_k.in_k.sim_k.out_k$. By assumption,

$$\texttt{enable\_}op_k^i(s_k^i, in_k^i, sim_k^i) \wedge \texttt{effect\_}op_k^i(s_k^i, in_k^i, sim_k^i, out_k^i, s_{k+1}^i)$$

holds. By using Lemma 5.3.15, we know that $s_k.x = s_k^i.x$ holds for all *referenced* variables within op_k^i.

Guard is Satisfied: In order to show $\texttt{enable\_}op_k(s_k, in_k^i, sim_k^i)$, we start with $op_k \in Op_i$, that is, op_k is a non-cut operation: $\texttt{enable\_}op_k = \texttt{enable\_}op_k^i$ holds according to Definition 4.3.10. We deduce:

$$
\begin{aligned}
& \texttt{enable\_}op_k^i(s_k^i, in_k^i, sim_k^i) \\
\overset{(1)}{\Rightarrow}\ & \texttt{enable\_}op_k(s_k \oplus s_k^i, in_k^i, sim_k^i) \\
\overset{(2)}{\Rightarrow}\ & \texttt{enable\_}op_k(s_k \oplus (s_k \restriction V_i), in_k^i, sim_k^i) \\
\overset{(3)}{\Rightarrow}\ & \texttt{enable\_}op_k(s_k, in_k^i, sim_k^i).
\end{aligned}
$$

Implication (1) is due to $\texttt{enable\_}op_k^i = \texttt{enable\_}op_k$ and the fact that only variables of V_i are referenced in $\texttt{enable\_}op_k^i$. The second implication follows by the induction hypothesis and Lemma 5.3.15, again using that non-referenced variables within $\texttt{enable\_}op_k$ do not affect the truth-value of the associated predicate. The last implication is immediate.

If we assume $op_k \in Op_{\mathbf{C}_i}$, the equation $\texttt{enable\_}op_k = \texttt{enable\_}op_k^i$ holds as well, and we proceed accordingly.

Operation is Executable with Compatible Successor State: Next, we show

$$\texttt{effect\_}op_k(s_k, in_k^i, sim_k^i, out_k^i, s_{k+1}),$$

by distinguishing the two cases

1. $op_k \in Op_i$ and
2. $op_k \in Op_{\mathbf{C}_i}$.

$op_k \in Op_i$**:** Here, $\texttt{effect\_}op_k = \texttt{effect\_}op_k^i$ and $op_k^j = \texttt{noev}$, $j \neq i$, according to Definition 4.3.10. We get:

$$
\begin{aligned}
& \texttt{effect\_}op_k^i(s_k^i, in_k^i, sim_k^i, out_k^i, s_{k+1}^i) \\
\overset{(1)}{\Rightarrow}\ & \texttt{effect\_}op_k(s_k \oplus s_k^i, in_k^i, sim_k^i, out_k^i, s_k \oplus s_{k+1}^i) \\
\overset{(2)}{\Rightarrow}\ & \texttt{effect\_}op_k(s_k \oplus (s_k \restriction V_i), in_k^i, sim_k^i, out_k^i, s_k \oplus s') \\
\overset{(3)}{\Rightarrow}\ & \texttt{effect\_}op_k(s_k, in_k^i, sim_k^i, out_k^i, s_k \oplus s'),
\end{aligned}
$$

for

$$s' := (s_{k+1}^i \restriction (ref(op_k) \cup mod(op_k))).$$

The first implication is analogous to the considerations for the `enable`-schema, and the last implication is obvious. For implication (2), the value $s_{k+1}^i.x$ can solely be used in case that either x is correspondingly modified by op_k and op_k^i or the identical pre-state values is kept. For the remaining variables, the value $s_k.x$ needs to be used.

The state $s_{k+1} = s_k \oplus s'$ satisfies the conditions of Equation 5.7 for any $x \in V$: the sole case of s_{k+1} and Equation 5.7 differing is $x \in (ref(op_k^i) \setminus mod(op_k^i))$. But then, $s_k.x = s_k^i.x = s_{k+1}^i$.

$op_k \in Op_{\mathbf{C}_i}$**:** Again, $\texttt{effect\_}op_k = \texttt{effect\_}op_k^i$, according to Definition 4.3.10. In addition, $\texttt{effect\_}op_k^j$ for $j \neq i$ solely comprises $\bigwedge_{w \in CV_j} w' = tr_w?$. The proof is corresponding to the previous case, except for the fact that now, $x \in mod(op_k^i) \cap mod(op_k^j)$ is possible. However, in this case, the modification is *identical* based on Definition 4.3.10, which corresponds to Equation 5.7, where we choose the modification from op_k^1. □

Step 3: As $\pi \in Traces(S.OZ)$ and $\pi{\restriction}E_S = tr$ hold, we immediately deduce $tr \in traces(S.OZ)$. This completes the proof of the right-to-left implication and thus, the correctness proof of the decomposition of the Object-Z part.

5.4 Correctness of the Renaming for the Decomposition

The previous sections showed correctness for the decomposition of both, the CSP part and the Object-Z part of S. Preservation of control flow and data flow can only be achieved by the introduction of additional parameters. One drawback of these parameters is the required modification of the types of operations from S: equivalence of S and $S_1 \parallel S_2$ can only be shown modulo a renaming of events.

According to Section 4.3.4, the addition of parameters requires a channel renaming f. As the interface of a specification declares the types of operations, the set of additional parameters is identical for the CSP part and the Object-Z part. However, according to Section 4.3, transmission parameters are solely restricted by the Object-Z part, whereas the restriction of address parameters is limited to the CSP part.

For the definition of the CSP parts of S_1 and S_2, we already introduced two renaming relations

$$\mathtt{R}_1^{\mathtt{C}} : E_S \to E_{S_1} \text{ and } \mathtt{R}_2^{\mathtt{C}} : E_S \to E_{S_2}.$$

These relations determine the possible events the CSP parts of S_i can communicate, and they fix the values for the address parameters, whereas transmission parameters remain unrestricted. Subsequently, in case that no restriction on either the transmission parameters or address parameters is conducted, we write $?tr$ and $?add$, respectively. If the number of additional parameters is irrelevant, we write $op.x.t.a$ to denote an event of $E_{S'}$, according to Section 5.2. Note that none of these parameters have to exist. We recall the definitions of $\mathtt{R}_1^{\mathtt{C}}$ and $\mathtt{R}_2^{\mathtt{C}}$:

$$\mathtt{R}_{\mathtt{i}}^{\mathtt{C}}(op.x) := \begin{cases} op.x.i, & op \in (Op_1 \cap Op_2) \setminus (Op_{\mathbf{C}_1} \cup Op_{\mathbf{C}_2}), \\ op.x?tr.a, & op \in Op_{\mathbf{C}} \wedge |l^{-1}(op)| > 1, \\ op.x?tr, & op \in Op_{\mathbf{C}} \wedge |l^{-1}(op)| = 1, \\ op.x, & \text{otherwise.} \end{cases}$$

The definition of $S_i.OZ$ implicitly defines two renaming relations for the Object-Z parts as well. The roles for restricting the different types of added parameters are switched: for an event $op.x \in E_S$, the Object-Z part of S_i is able to communicate any event $op.x.t?add$,

as address parameters are unrestricted for the Object-Z part. Precisely, we get a renaming relation for the Object-Z part, given as:

$$\mathtt{R}_\mathtt{i}^\mathtt{O}(op.x) := \begin{cases} op.x?add, & op \in (Op_1 \cap Op_2) \setminus (Op_{\mathbf{C}_1} \cup Op_{\mathbf{C}_2}), \\ op.x.t?add, & op \in Op_\mathbf{C} \wedge |l^{-1}(op)| > 1, \\ op.x.t, & op \in Op_\mathbf{C} \wedge |l^{-1}(op)| = 1, \\ op.x, & \text{otherwise.} \end{cases}$$

The renaming needs to be considered, when it comes to showing trace equivalence between the original system and the decomposition. We use several notations for the combinations of the four renaming relations $\mathtt{R}_1^\mathtt{C}$, $\mathtt{R}_2^\mathtt{C}$, $\mathtt{R}_1^\mathtt{O}$ and $\mathtt{R}_2^\mathtt{O}$:

- $\mathtt{R}_\mathtt{i}$ denotes the union of $\mathtt{R}_\mathtt{i}^\mathtt{C}$ and $\mathtt{R}_\mathtt{i}^\mathtt{O}$.
- $\mathtt{R}^\mathtt{C}$ denotes the union of $\mathtt{R}_1^\mathtt{C}$ and $\mathtt{R}_2^\mathtt{C}$ and, accordingly, $\mathtt{R}^\mathtt{O}$ the union of $\mathtt{R}_1^\mathtt{O}$ and $\mathtt{R}_2^\mathtt{O}$.
- Finally, $\mathtt{R}$ denotes the union of *all* renaming relations, that is, the union of $\mathtt{R}^\mathtt{C}$ and $\mathtt{R}^\mathtt{O}$ or, accordingly, $\mathtt{R}_1$ and $\mathtt{R}_2$.

For achieving a comparison between both, the original system and its decomposition, we consider the – in regard of $\mathtt{R}$ – *inverse* relation $\mathtt{R}'$, which removes the additional transmission parameters and address parameters. More precisely, $\mathtt{R}' : E_{S'} \to E_S$ and

$$\mathtt{R}'(op.x.t.a) := op.x.$$

We are now able to relate S to $(S_1 \parallel_{E_\mathbf{C}} S_2)$ by means of $\mathtt{R}'$. Correctness for the decomposition of the Object-Z part and the CSP part was carried out modulo $\mathtt{R}'$, and we showed the equivalences

- $S.\mathtt{main} =_T (S_1.\mathtt{main} \parallel_{E_\mathbf{C}} S_2.\mathtt{main})[\![\mathtt{R}']\!]$ and
- $S.OZ =_T (S_1.OZ \parallel_{E_\mathbf{C}} S_2.OZ)[\![\mathtt{R}']\!]$ for the set of traces conforming to the CSP part.

In order to facilitate reasoning about the individual component's parts $S_i.\mathtt{main}$ and $S_i.OZ$, we show that $\mathtt{R}'$ can be *distributed* over the parallel composition operator. Precisely, we show the following equivalence in the semantic domain of the CSP trace model:

$$(S_1.\mathtt{main} \parallel_{E_\mathbf{C}} S_2.\mathtt{main})[\![\mathtt{R}']\!] \parallel_{E_S} (S_1.OZ \parallel_{E_\mathbf{C}} S_2.OZ)[\![\mathtt{R}']\!] =_T \\ ((S_1.\mathtt{main} \parallel_{E_\mathbf{C}} S_2.\mathtt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_\mathbf{C}} S_2.OZ))[\![\mathtt{R}']\!].$$

Here, the crucial aspect is that the synchronisation alphabet for the outer parallel composition changes from E_S to $E_{S'}$, as the right hand side now synchronises over events *after* the renaming took place.

We start the proof by showing a property about the composition of a general relation and its inverse: if a relation $\mathtt{Re}$ is total and injective, the composition $\mathtt{Re}^{-1} \circ \mathtt{Re}$ is the identity relation.

Lemma 5.4.1. *(Composition law for injective and total relations)*
Let $\mathtt{Re} \subseteq \mathcal{A} \times \mathcal{B}$ *be a relation, and let*

$$\mathtt{Re}^{-1} := \{(b,a) \mid (a,b) \in \mathtt{Re}\}$$

be its inverse relation. If Re *is total and injective, then*

$$(\mathtt{Re}^{-1} \circ \mathtt{Re}) = \mathsf{Id}_{\mathcal{A}}.$$

Proof. We recall the definitions for a relation being total and injective and the one for the composition of two relations:

- Re is total, if, and only if, $\forall a \in \mathcal{A}\ \exists (a,b) \in \mathcal{A} \times \mathcal{B} \bullet (a,b) \in \mathtt{Re}$.
- Re is injective, if, and only if, $\forall (a,b),(a',b') \in \mathtt{Re} \bullet b = b' \Rightarrow a = a'$.
- $\mathtt{Re}^{-1} \circ \mathtt{Re} = \{(x,y) \in \mathcal{A} \times \mathcal{A} \mid \exists z \in \mathcal{B} \bullet (x,z) \in \mathtt{Re} \wedge (z,y) \in \mathtt{Re}^{-1}\}$.

Based on the fact that Re is total, $\mathsf{Id}_{\mathcal{A}} \subseteq (\mathtt{Re}^{-1} \circ \mathtt{Re})$ holds. Now assume some $(x,y) \in (\mathtt{Re}^{-1} \circ \mathtt{Re})$ and $x \neq y$. But then, there exists $z \in \mathcal{B}$, such that $(x,z) \in \mathtt{Re}$ and $(z,y) \in \mathtt{Re}^{-1}$. The definition of $\mathtt{Re}^{-1}$ yields $(x,z),(y,z) \in \mathtt{Re}$, contradiction to Re being injective. □

For an application of this property on our renaming relation, we show that R, $\mathtt{R}^{\mathtt{C}}$ and $\mathtt{R}^{\mathtt{O}}$ are both, total and injective:

Lemma 5.4.2. *(Properties of event renaming)*
The following properties for R, $\mathtt{R}^{\mathtt{C}}$ *and* $\mathtt{R}^{\mathtt{O}}$ *are satisfied:*

a) R, $\mathtt{R}^{\mathtt{C}}$ *and* $\mathtt{R}^{\mathtt{O}}$ *are total.*

b) R, $\mathtt{R}^{\mathtt{C}}$ *and* $\mathtt{R}^{\mathtt{O}}$ *are injective.*

c) $(\mathtt{R}' \circ \mathtt{R}) = (\mathtt{R}' \circ \mathtt{R}^{\mathtt{C}}) = (\mathtt{R}' \circ \mathtt{R}^{\mathtt{O}}) = \mathsf{Id}_{E_S}$ *and for any CSP process Q:*

- $Q = Q'[\![\mathtt{R}]\!]$ *for some process* Q' *implies* $Q[\![(\mathtt{R} \circ \mathtt{R}')]\!] = Q$.
- $Q = Q'[\![\mathtt{R}^{\mathtt{C}}]\!]$ *for some process* Q' *implies* $Q[\![(\mathtt{R}^{\mathtt{C}} \circ \mathtt{R}')]\!] = Q$.
- $Q = Q'[\![\mathtt{R}^{\mathtt{O}}]\!]$ *for some process* Q' *implies* $Q[\![(\mathtt{R}^{\mathtt{O}} \circ \mathtt{R}')]\!] = Q$.

Proof.

a) As the renaming relations are defined with respect to the whole set E_S, we immediately deduce this property. ✓

b) Immediately follows by the implication

$$(op_1.x_1.t_1.a_1 = op_2.x_2.t_2.a_2) \Rightarrow (op_1.x = op_2.x).\ \checkmark$$

c) Based on Lemma 5.4.2, the combination of a) and b) yields

$$(\mathtt{R}' \circ \mathtt{R}) = (\mathtt{R}' \circ \mathtt{R}^{\mathtt{C}}) = (\mathtt{R}' \circ \mathtt{R}^{\mathtt{O}}) = \mathsf{Id}_{E_S}.$$

Thus, $(\mathtt{R} \circ \mathtt{R}' \circ \mathtt{R}) = \mathtt{R}$, $(\mathtt{R}^{\mathtt{C}} \circ \mathtt{R}' \circ \mathtt{R}^{\mathtt{C}}) = \mathtt{R}^{\mathtt{C}}$ and $(\mathtt{R}^{\mathtt{O}} \circ \mathtt{R}' \circ \mathtt{R}^{\mathtt{O}}) = \mathtt{R}^{\mathtt{O}}$. ✓ □

We use these properties in the following theorem, showing the already mentioned distributivity law for the inverse renaming. The core idea for its proof is the following: both additional parameter types are either restricted by the decomposition's Object-Z part (transmission parameters) or by its CSP part (address parameters), but neither of them by *both*. Therefore, if a synchronisation of some $op.x$ between $(S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main})$ and $(S_1.OZ \parallel_{E_C} S_2.OZ)$ is possible *after* removing the additional parameters, there exists some $op.x.t.a$ on which both parts can synchronise *beforehand*: the intersection of the newly constructed event sets of the CSP part and the Object-Z part is non-empty.

Theorem 5.4.3. *(Distributivity law for inverse renaming)*
The inverse renaming relation $\mathtt{R}'$ *distributes over the parallel composition* $\parallel_{E_S}$*, that is:*

$$(S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main})[\![\mathtt{R}']\!] \parallel_{E_S} (S_1.OZ \parallel_{E_C} S_2.OZ)[\![\mathtt{R}']\!] =_T ((S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_C} S_2.OZ))[\![\mathtt{R}']\!].$$

Proof. First, note that $E_S = \mathtt{R}'(E_{S'})$ holds as $\mathtt{R}$ is total (Lemma 5.4.2, a)) and thus, $\mathtt{R}'$ is surjective. Let $P := S_1.\texttt{main} \parallel_{E_C} S_2.\texttt{main}$ and $Q := S_1.OZ \parallel_{E_C} S_2.OZ$. We prove the theorem by showing that $(P \parallel_{E_{S'}} Q)[\![\mathtt{R}']\!]$ and $P[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} Q[\![\mathtt{R}']\!]$ are the initial states of a strong bisimulation [Mil89]

$$\mathcal{R} := \{(A,B) \mid A = (C \parallel_{E_{S'}} D)[\![\mathtt{R}']\!], B = C[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} D[\![\mathtt{R}']\!]\},$$

where C depicts any reachable state within the labelled transition system of P, and D denotes any reachable state within the labelled transition system of Q. Based on the definition of bisimulation, we need to show two directions:

(1) If $(A,B) \in \mathcal{R}$ and $B \xrightarrow{e} B'$ for $e \in (E_S \cup \{\tau\})$, then there exists some A', such that $A \xrightarrow{e} A'$ and $(A',B') \in \mathcal{R}$.

(2) If $(A,B) \in \mathcal{R}$ and $A \xrightarrow{e} A'$ for $e \in (E_S \cup \{\tau\})$, then there exists some B', such that $B \xrightarrow{e} B'$ and $(A',B') \in \mathcal{R}$.

Based on the firing rules for CSP, renaming has no effect on τ-transitions [Ros98]. Thus, for the τ-case, both directions are immediate.

(1): Let $(A,B) \in \mathcal{R}$ and $B \xrightarrow{op.x} B'$. Since $\mathtt{R}'(E_{S'}) = E_S$, both processes $C[\![\mathtt{R}']\!]$ and $D[\![\mathtt{R}']\!]$ need to synchronise on $op.x$. Based on the operational semantics of CSP, there exist B'_1, B'_2, such that $B' = B'_1 \parallel_{\mathtt{R}'(E_{S'})} B'_2$ and

$$C[\![\mathtt{R}']\!] \xrightarrow{op.x} B'_1 \text{ and } D[\![\mathtt{R}']\!] \xrightarrow{op.x} B'_2.$$

From 5.4.2, c), we deduce $B'_1 = B'_1[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{C}})]\!]$ and $B'_2 = B'_2[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{O}})]\!]$. Thus,

$$C[\![\mathtt{R}']\!] \xrightarrow{op.x} B'_1[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{C}})]\!] \text{ and } D[\![\mathtt{R}']\!] \xrightarrow{op.x} B'_2[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{O}})]\!].$$

By applying R and the firing rule for relational renaming from [Ros98], we get

$$C \xrightarrow{op.x.t_1.a_1} B'_1[\![\mathtt{R}^{\mathtt{C}}]\!] \text{ and } D \xrightarrow{op.x.t_2.a_2} B'_2[\![\mathtt{R}^{\mathtt{O}}]\!]$$

for all $op.x.t_1.a_1 \in \mathtt{R}^{\mathtt{C}}(op.x)$ and $op.x.t_2.a_2 \in \mathtt{R}^{\mathtt{O}}(op.x)$. Here, we again apply Lemma 5.4.2, c), as $C[\![(\mathtt{R}^{\mathtt{C}} \circ \mathtt{R}')]\!] = C$ and $D[\![(\mathtt{R}^{\mathtt{O}} \circ \mathtt{R}')]\!] = D$ holds. The following observation is the crucial point in this proof: the intersection of $\mathtt{R}^{\mathtt{C}}(op.x)$ and $\mathtt{R}^{\mathtt{O}}(op.x)$ is *non-empty*, since the CSP part solely restricts the address parameters, whereas the Object-Z part solely restricts the transmission parameters of an operation. Precisely,

- $\mathtt{R}^{\mathtt{C}}(op.x) = \{op.x.?tr.a \mid a \text{ addressing extension for } op\}$ and
- $\mathtt{R}^{\mathtt{O}}(op.x) = \{op.x.t?add \mid t \text{ transmission parameters for } op\}$.

Thus, there exists some $op.x.t.a \in (\mathtt{R}^{\mathtt{C}}(op.x) \cap \mathtt{R}^{\mathtt{O}}(op.x))$ on which C and D can synchronise on. We deduce

$$(C \parallel_{E_{S'}} D) \xrightarrow{op.x.t.a} (B_1'[\![\mathtt{R}^{\mathtt{C}}]\!] \parallel_{E_{S'}} B_2'[\![\mathtt{R}^{\mathtt{O}}]\!]).$$

Again applying the firing rule for relation renaming, we get

$$(C \parallel_{E_{S'}} D)[\![\mathtt{R}']\!] \xrightarrow{op.x} (B_1'[\![\mathtt{R}^{\mathtt{C}}]\!] \parallel_{E_{S'}} B_2'[\![\mathtt{R}^{\mathtt{O}}]\!])[\![\mathtt{R}']\!],$$

based on $\mathtt{R}'(op.x.t.a) = op.x$. Finally,

$$(A', B') = ((B_1'[\![\mathtt{R}^{\mathtt{C}}]\!] \parallel_{E_{S'}} B_2'[\![\mathtt{R}^{\mathtt{O}}]\!])[\![\mathtt{R}']\!], (B_1'[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{C}})]\!] \parallel_{\mathtt{R}'(E_{S'})} B_2'[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{O}})]\!])) \in \mathcal{R}.$$

The bisimulation diagram for this case is given next.

$$\begin{array}{lccc}
B = & C[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} D[\![\mathtt{R}']\!] & \xrightarrow{op.x} & B_1'[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{C}})]\!] \parallel_{\mathtt{R}'(E_{S'})} B_2'[\![(\mathtt{R}' \circ \mathtt{R}^{\mathtt{O}})]\!] = B' \\
& \mid & & \vdots \\
& \mathcal{R} & & \mathcal{R} \\
& \mid & & \vdots \\
A = & (C \parallel_{E_{S'}} D)[\![\mathtt{R}']\!] & \xrightarrow{op.x} & (B_1'[\![\mathtt{R}^{\mathtt{C}}]\!] \parallel_{E_{S'}} B_2'[\![\mathtt{R}^{\mathtt{O}}]\!])[\![\mathtt{R}']\!] = A'
\end{array}$$

(**2**)**:** For the second implication, assume that $(A, B) \in \mathcal{R}$ and $A \xrightarrow{op.x} A'$, that is,

$$(C \parallel_{E_{S'}} D)[\![\mathtt{R}']\!] \xrightarrow{op.x} A'.$$

Again, based on Lemma 5.4.2, we have the identity $(\mathtt{R} \circ \mathtt{R}')$, yielding

$$(C \parallel_{E_{S'}} D) \xrightarrow{op.x.t.a} A'[\![\mathtt{R}]\!]$$

for any $op.x.t.a \in \mathtt{R}(op.x)$. Based on the operational semantics of CSP, there need to exist some A_1' and A_2', such that

$$A'[\![\mathtt{R}]\!] = A_1' \parallel_{E_{S'}} A_2'.$$

Following up, we get

$$C \xrightarrow{op.x.t.a} A_1' \text{ and } D \xrightarrow{op.x.t.a} A_2'.$$

Application of $\mathtt{R}'$ leads to

$$C[\![\mathtt{R}']\!] \overset{op.x}{\rightarrow} A_1'[\![\mathtt{R}']\!] \text{ and } D[\![\mathtt{R}']\!] \overset{op.x}{\rightarrow} A_2'[\![\mathtt{R}']\!]$$

and finally,

$$C[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} D[\![\mathtt{R}']\!] \overset{op.x}{\rightarrow} A_1'[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} A_2'[\![\mathtt{R}']\!].$$

This concludes the left-to-right implication, as

$$(A', B') = ((A_1' \parallel_{E_{S'}} A_2')[\![\mathtt{R}']\!], A_1'[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} A_2'[\![\mathtt{R}']\!]) \in \mathcal{R}$$

holds. □

$$\begin{array}{lllll} A = & (C \parallel_{E_{S'}} D)[\![\mathtt{R}']\!] & \overset{op.x}{\longrightarrow} & (A_1' \parallel_{E_{S'}} A_2')[\![\mathtt{R}']\!] & = A' \\ \mid & & & \vdots & \\ \mathcal{R} & & & \mathcal{R} & \\ \mid & & & \vdots & \\ B = & C[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} D[\![\mathtt{R}']\!] & \overset{op.x}{\longrightarrow} & A_1'[\![\mathtt{R}']\!] \parallel_{\mathtt{R}'(E_{S'})} A_2'[\![\mathtt{R}']\!] & = B' \end{array}$$

The previous theorem showed that the renaming relation can be distributed over the parallel composition $\parallel_{E_S}$, allowing us to reason about $S_i.\mathtt{main}$ and $S_i.OZ$ and its parallel composition, without considering the renaming relation.

5.5 CSP Laws for Parallel Composition

The last step in the chain of proof steps is the easiest one: we need to show that within the parallel composition

$$(S_1.\mathtt{main} \parallel_{E_C} S_2.\mathtt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_C} S_2.OZ),$$

$S_1.OZ$ and $S_2.\mathtt{main}$ can be redistributed, such that the resulting parallel composition constitutes the assembly of S_1 and S_2. In particular, the respective synchronisation alphabets need to be correctly adapted.

The following lemma shows a generalisation of this property for arbitrary processes with certain restrictions on their alphabets. Afterwards, we instantiate the lemma for our specific case:

Lemma 5.5.1. *(Redistribution of CSP processes, alphabetised parallel)*
Let P_i, Q_i be CSP processes and A_i, B_i alphabets. Then,

$$(P_1 \; {}_{A_1}\!\|_{A_2} \; P_2) \; {}_{A_1 \cup A_2}\!\|_{B_1 \cup B_2} \; (Q_1 \; {}_{B_1}\!\|_{B_2} \; Q_2) = (P_1 \; {}_{A_1}\!\|_{B_1} \; Q_1) \; {}_{A_1 \cup B_1}\!\|_{A_2 \cup B_2} \; (P_2 \; {}_{A_2}\!\|_{B_2} \; Q_2).$$

Proof. We use rules (2.4) $\ {}_X\|_Y\ -$ `sym`

$$P\ {}_X\|_Y\ Q = Q\ {}_Y\|_X\ P$$

and (2.5) $\ {}_X\|_Y\ -$ `assoc`

$$(P\ {}_X\|_Y\ Q)\ {}_{X\cup Y}\|_Z\ R = P\ {}_X\|_{Y\cup Z}\ (Q\ {}_Y\|_Z\ R)$$

from [Ros98] and incrementally deduce the equation:

$$\begin{aligned}
& (P_1\ {}_{A_1}\|_{A_2}\ P_2)\ {}_{A_1\cup A_2}\|_{B_1\cup B_2}\ (Q_1\ {}_{B_1}\|_{B_2}\ Q_2) \\
= \ & P_1\ {}_{A_1}\|_{A_2\cup B_1\cup B_2}\ (P_2\ {}_{A_2}\|_{B_1\cup B_2}\ (Q_1\ {}_{B_1}\|_{B_2}\ Q_2)) && ({}_X\|_Y\ -\ \texttt{assoc}) \\
= \ & P_1\ {}_{A_1}\|_{B_1\cup B_2\cup A_2}\ ((Q_1\ {}_{B_1}\|_{B_2}\ Q_2)\ {}_{B_1\cup B_2}\|_{A_2}\ P_2) && ({}_X\|_Y\ -\ \texttt{sym}) \\
= \ & P_1\ {}_{A_1}\|_{B_1\cup A_2\cup B_2}\ (Q_1\ {}_{B_1}\|_{B_2\cup A_2}\ (Q_2\ {}_{B_2}\|_{A_2}\ P_2)) && ({}_X\|_Y\ -\ \texttt{assoc}) \\
= \ & P_1\ {}_{A_1}\|_{B_1\cup A_2\cup B_2}\ (Q_1\ {}_{B_1}\|_{A_2\cup B_2}\ (P_2\ {}_{A_2}\|_{B_2}\ Q_2)) && ({}_X\|_Y\ -\ \texttt{sym}) \\
= \ & (P_1\ {}_{A_1}\|_{B_1}\ Q_1)\ {}_{A_1\cup B_1}\|_{A_2\cup B_2}\ (P_2\ {}_{A_2}\|_{B_2}\ Q_2) && ({}_X\|_Y\ -\ \texttt{assoc})
\end{aligned}$$

□

In order to use the previous lemma in our context, we have to apply it with respect to *interface* parallel. This can only be achieved, if all participating processes never communicate outside their respective synchronisation alphabets:

Corollary 5.5.2. *(Redistribution of CSP processes, interface parallel)*
Let P_i, Q_i be CSP processes and A_i, B_i their respective alphabets, that is, P_i never communicates outside of A_i, and Q_i never communicates outside of B_i, respectively. Then:

$$\begin{aligned}
& (P_1\ \|_{A_1\cap A_2}\ P_2)\ \|_{(A_1\cup A_2)\cap(B_1\cup B_2)}\ (Q_1\ \|_{B_1\cap B_2}\ Q_2) = \\
& (P_1\ \|_{A_1\cap B_1}\ Q_1)\ \|_{(A_1\cup B_1)\cap(A_2\cup B_2)}\ (P_2\ \|_{A_2\cap B_2}\ Q_2).
\end{aligned}$$

Proof. Follows immediately from Lemma 5.5.1 and the fact that $P\ \|_{X\cap Y}\ Q = P\ {}_X\|_Y\ Q$ holds„ if P, Q never communicate outside X and Y ([Ros98]). □

In the following section, the corollary will be instantiated by setting

- $P_i := S_i.\texttt{main}$,
- $Q_i := S_i.OZ$,
- $A_i := E_{S_i}$ and
- $B_i := E_{S_i}$.

5.6 Proof of the Main Theorem

Finally, we show Theorem 4.3.25 by subsuming the results of the previous sections:

Theorem 5.6.1. *(Correctness of the decomposition)*
Let S be a specification, and let $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$ be a cut, yielding a decomposition into S_1 and S_2, according to Definition 4.3.24. Then, the following holds:

$$S =_T (S_1 \parallel_{E_\mathbf{C}} S_2)[\![\mathtt{R}']\!],$$

where $\mathtt{R}' : E_{S'} \to E_S$ is defined as

$$\mathtt{R}'(op.x.t.a) := op.x,$$

with x depicting the original parameter values, t denoting the valuation for the possible transmission *parameters and a the valuation for the possible* address *parameters.*

Proof.

$$\begin{aligned}
& S \\
=_T\ & S.\mathtt{main} \parallel_{E_S} S.OZ && \text{(Def. of } S\text{)} \\
=_T\ & (S_1.\mathtt{main} \parallel_{E_\mathbf{C}} S_2.\mathtt{main})[\![\mathtt{R}']\!] \parallel_{E_S} (S_1.OZ \parallel_{E_\mathbf{C}} S_2.OZ)[\![\mathtt{R}']\!] \\
& && \text{(Theorem 5.2.4, Theorems 5.3.14 and 5.3.17)} \\
=_T\ & [(S_1.\mathtt{main} \parallel_{E_\mathbf{C}} S_2.\mathtt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_\mathbf{C}} S_2.OZ)][\![\mathtt{R}']\!] && \text{(Theorem 5.4.3)} \\
=_T\ & [(S_1.\mathtt{main} \parallel_{E_{S_1} \cap E_{S_2}} S_2.\mathtt{main}) \parallel_{E_{S'}} (S_1.OZ \parallel_{E_{S_1} \cap E_{S_2}} S_2.OZ)][\![\mathtt{R}']\!] && \text{(Lemma 4.3.19)} \\
=_T\ & [(S_1.\mathtt{main} \parallel_{E_{S_1}} S_1.OZ) \parallel_{E_{S_1} \cap E_{S_2}} (S_2.\mathtt{main} \parallel_{E_{S_2}} S_2.OZ)][\![\mathtt{R}']\!] && \text{(Corollary 5.5.2)} \\
=_T\ & [(S_1.\mathtt{main} \parallel_{E_{S_1}} S_1.OZ) \parallel_{E_\mathbf{C}} (S_2.\mathtt{main} \parallel_{E_{S_2}} S_2.OZ)][\![\mathtt{R}']\!] && \text{(Lemma 4.3.19)} \\
=_T\ & (S_1 \parallel_{E_\mathbf{C}} S_2)[\![\mathtt{R}']\!] && \text{(Def. of } S_1 \text{ and } S_2\text{)}
\end{aligned}$$

□

Note that an application of Lemma 5.5.2 is indeed possible, as $S_i.\mathtt{main}$ and $S_i.OZ$ never communicate outside of E_{S_i}.

This completes the proof of the main result of this thesis, Theorem 4.3.25. The theorem allows us to apply the assume-guarantee-based proof rules from Chapter 3: as S and $S_1 \parallel_{E_\mathbf{C}} S_2$ are trace equivalent modulo renaming, we can safely replace $S_1 \parallel S_2$ by S in an application of **(B-AGR)** and **(P-AGR)**.

After showing *correctness* of our decomposition approach, the next chapter will deal with the question on how to identify *reasonable* decompositions, that is, correct decompositions, which most likely result in efficient compositional verification.

6 Finding Reasonable Decompositions

Contents

6.1 Decomposition Heuristics 168
6.1.1 First Heuristic: Cut Size 169
6.1.2 Second Heuristic: Even Distribution 170
6.1.3 Third Heuristic: Few Transmission 170
6.1.4 Fourth Heuristic: Few Addressing 172
6.2 Evaluation of Decomposition Heuristics 172
6.3 Candy Machine Revisited: Evaluation of Cuts 174
6.4 Case Study: Two Phase Commit Protocol 175
6.5 Discussion 180
6.6 Related Work 181

The overall goal of our decomposition technique is an application within the compositional verification framework, as introduced in Chapter 3. So far, we showed the *correctness* of our approach: the decomposition does not change the behaviour of the specification in terms of our semantic domain. This allows us to apply assume-guarantee-based proof rules with respect to the decomposed system and to infer a global result for the original system.

However, for compositional verification to have a practical impact, the technique needs to provide an advantage over monolithic, that is, non-compositional verification. Therefore, it is essential to evaluate the *effectiveness* of the decomposition as, in general, compositional verification does not automatically result in comparatively small time and memory consumption during model checking.

Example 6.0.2. *Recall the specification of a candy machine from Figure 2.3. The set* $\mathbf{C} = \{term\}$ *defines a valid (single) cut of the specification. A decomposition with respect to* $\mathbf{C}$ *is impractical, as it results in* $S_1 =_T S$*. Yet, even though we consider compositional verification, the need to deal with the full state space of S remains.*[1]

The question is how to describe and detect *reasonable* decompositions. In [CAC06], the authors investigate the usefulness of assume-guarantee reasoning by evaluating all possible decompositions on five different case studies. The overall results are not very encouraging as, in terms of the size of the explored state spaces, monolithic verification often succeeds over compositional verification. This leads the authors to the following statements:

[1] From now on, as a valid cut uniquely defines a decomposition, we will synonymously use both terms.

> *Deciding how to partition the subsystems into S_1 and S_2 is not easy and can have a significant impact on the time and memory needed for verification. [...] Thus, randomly selecting decompositions would likely not yield a decomposition better than monolithic verification.* *[CAC06]*

As a possible solution to this problem, the authors recommend to investigate *heuristics* to guide the software engineer towards the *best* possible decompositions: in this case, assume-guarantee-based verification most often outperforms non-compositional verification.

These considerations motivate the following strategy: in order to evaluate the valid decompositions which our technique generates, we define several context-specific heuristics, focusing on the underlying verification framework and the definition of the decomposition itself. These heuristics serve as the basis for a classification of all correct decompositions: those, which are *unreasonable* or *dominated* by other ones (see Section 6.2), are no longer considered - the remaining decompositions can be further compared by prioritising specific heuristics.

This chapter is organised as follows: in Section 6.1, we motivate and define our context-specific heuristics. The following section discusses an evaluation of the results by giving a very brief introduction into the topic of multi-valued optimisation [Ehr00]. In Sections 6.3 and 6.4, respectively, we illustrate and apply the heuristics and evaluate them for the candy machine specification and a second, slightly bigger, case study. The final sections discuss the approach and related work.

6.1 Decomposition Heuristics

Several factors influence the effectiveness of compositional reasoning in general. In [dRHH$^+$01], the authors elaborate on the question of when to use a *compositional* style of proof and when to use a *non-compositional* one. For instance, they argue that compositional verification becomes infeasible, if a system is tightly-coupled, that is, any decomposition will result in a lot of common elements and shared behaviour, or if the system comprises global invariants, which cannot be split up.

Choosing the most effective decompositions cannot be established in an automatic manner. Due to the context-specific verification frameworks, the usage of different model checkers or the structure of the specifications and verification properties, there is no universally optimal decomposition. However, one particular issue exerts the dominating influence on the efficiency of compositional verification and model checking in general: the size of the state space, which needs to be explored. Thus, according to [CAC06], we state that one decomposition is better than another one, if *the number of states explored during model checking is comparatively smaller*. We need to define heuristics, favouring decompositions with a relatively small state space.

The evaluation of our approach will use an implementation of the learning-based framework from Section 3.2.3 and compare it to direct model checking of the original system. We derive our heuristics from the following two requirements:

Small Interface: The size of $E_{\mathbf{C}}$ within the system $S_1 \parallel_{E_{\mathbf{C}}} S_2$ should be small. In general,

the smaller the interface between both components, the less shared behaviour and fewer communication between them, and the looser the components are coupled. This results in a smaller number of states, which have to be explored during model checking. In the context of AGR and according to [CS07], the smaller the assumption alphabet, the more efficient the L* algorithm in the learning-based framework from Section 3.2.3. More precisely, the number of L* membership queries directly depends on the assumption alphabet, which itself closely depends on the size of $E_{\mathbf{C}}$.

Equal Size of Components: The size of the components S_1 and S_2 within $S_1 \parallel_{E_{\mathbf{C}}} S_2$ should approximately be the same. The question of how to find a good partitioning of a system is discussed in [Nam07]. The author argues that an even distribution of the number of system variables over the components leads to a more effective compositional verification. Moreover, in [GMF07], the authors state that the execution time of the L* algorithm is exponential in the size of S_1 and S_2. If we assume s_1 to denote the size of S_1 and s_2 the size of S_2, $2^{s_1} + 2^{s_2}$ is minimal for $s_1 = s_2$, in case that $s = s_1 + s_2$ is fixed. This justifies the requirement that both components should have about the same size.

Based on these two requirements, we derive four different heuristics, with one of them related to *Equal Size of Components* and the remaining three based on *Small Interface*. These heuristics will be given as functions, mapping a specific decomposition on a certain value within the natural numbers. A comparatively better decomposition has a lower value, that is, we aim at a *minimisation* of the function values. For each heuristic, we start by stating the principal characteristic and give an intuitive description. Subsequently, we introduce the mathematical definition.

6.1.1 First Heuristic: Cut Size

The first heuristic, which we call **cut size**, is related to the requirement that the interface between both components should be relatively small. A small number of nodes within the cut is preferable, as the size of $E_{\mathbf{C}}$ depends on the number of corresponding operation nodes, that is,

$$E_{\mathbf{C}} = \{\mid Op_{\mathbf{C}} \mid\} = \{\mid l[\mathbf{C}_1] \cup l[\mathbf{C}_2] \mid\}$$

holds. This leads to the following objective for the first heuristic:

$$h_{\mathbf{CS}}\textit{: Minimise the number of cut nodes.}$$

The fewer nodes contained in the cut, the smaller the common elements to both specification parts and thus, the smaller the shared behaviour and the assumption alphabet. A mathematical definition for this heuristic obviously maps a cut on its number of elements. We summarise the first heuristic in Table 6.1.

Notation	Name of Heuristic	Description	Motivation
$h_{\mathbf{CS}}$	**cut size**	Minimise number of cut nodes.	*Small Interface*.
Mathematical Definition			
$h_{\mathbf{CS}}(\mathbf{C}) := \#\mathbf{C}$			

Table 6.1: Heuristic $h_{\mathbf{CS}}$: **cut size**

6.1.2 Second Heuristic: Even Distribution

The second heuristic, **even distribution**, targets the *Equal Size of Components*. In order to measure the size of a component, we count the number of operation nodes corresponding to S_1 and S_2, leading to the following objective:

$h_{\mathbf{ED}}$*: Minimise the difference between the number of operation nodes corresponding to* S_1 *and* S_2.

Based on Definition 4.3.1, we get

- $\mathbf{Ph}_1 \cup \mathbf{Ph}_3 \cup \mathbf{C}_1 \cup \mathbf{C}_2$ for the set of nodes according to S_1 and
- $\mathbf{Ph}_2 \cup \mathbf{C}_1 \cup \mathbf{C}_2$ for the set of nodes according to S_2.

The mathematical definition for the second heuristic from Table 6.2 computes the absolute value of the difference between these sets. As the set of cut nodes is contained in both of them, it can be neglected.

Notation	Name of Heuristic	Description	Motivation
$h_{\mathbf{ED}}$	**even distribution**	Minimise size difference between both components.	*Equal Size of Components*.
Mathematical Definition			
$h_{\mathbf{ED}}(\mathbf{C}) :=\mid \#(\mathbf{Ph}_1 \cup \mathbf{Ph}_3) - \#\mathbf{Ph}_2 \mid$			

Table 6.2: Heuristic $h_{\mathbf{ED}}$: **even distribution**

6.1.3 Third Heuristic: Few Transmission

The final two heuristics are again related to the requirement *Small Interface*. In Section 4.3.2, we introduced the concept of transmission parameters to ensure a preservation of a specification's data flow within the decomposition. These parameters are required to

ensure the correctness of the technique. Unfortunately, they increase the set of cut events, that is, the set $E_{\mathbf{C}}$.

In order to measure the additional amount of cut events, we need to refer to the *types* of these parameters: simply counting the number of parameters would be too coarse. For instance, one additional parameter of type $\{1, \ldots, 10\}$ causes the size of $E_{\mathbf{C}}$ to be increased by the factor of 10, whereas two parameters of type $\{1, 2\}$ only increase it by the factor of 4. In order to define the third heuristic $h_{\mathbf{FT}}$, we proceed as follows:

- The number of elements of $\{\mid op \mid\}$ (see Definition 2.2.5) increases by the amount of possible parameter extensions with respect to all transmission parameters. Thus, for any cut operation *op*, we compute the product over the cardinality of each transmission parameter type.
- An operation can have multiple occurrences within the cut. Even though this is not reflected in the size of $E_{\mathbf{C}}$, we still need to deal with it by multiplying the previous result with the number of cut-occurrences of the operation.
- Finally, we compute the sum over the results for all cut operations.

Henceforth, *tr_i* denote *transmission* parameters, and ty_p^{op} depicts the type of the parameter *p* of the operation *op*. We illustrate the weight computation for the third heuristic with an example:

Example 6.1.1. *Let* $\mathbf{C} = \{op_1, op_2\}$ *be a valid cut for some specification S, such that* op_1 *occurs once, and* op_2 *occurs twice within S*.main. *Let* op_1 *comprise two transmission parameters of types* $\mathbb{B}$ *and* $\{1, 2, 3\}$. *Furthermore, let* op_2 *comprise one transmission parameter of type* $\mathbb{P}(\mathbb{B})$. *For the first operation, we get* $\#ty_{tr\_1}^{op_1} * \#ty_{tr\_2}^{op_1} = 2 * 3 = 6$. *Moreover,* $\#ty_{tr\_1}^{op_2} = 2^2 = 4$. *As* op_2 *occurs twice within S*.main, *we multiply the second value by* 2, *which results in the overall weight of* $h_{\mathbf{FT}}(\mathbf{C}) = 6 + 8 = 14$ *for the third heuristic.*

Another question is how to deal with *infinite* data types. One solution could be the definition $h_{\mathbf{FT}}(\mathbf{C}) := \infty$ in case that there *exists at least one* operation within **C** with one transmission parameter of infinite type. However, in this case, any number of transmission parameters of infinite types would result in the same value for the given heuristic. During model checking, infinite data types need to be abstracted to some finite subset - either by the model checker or the user itself. Therefore, we follow a different approach: in our cardinal arithmetic, we assume that ∞ can be mapped to some bound **MaxInf**. Based on the actual cardinality of ∞ for the model checker, **MaxInf** can appropriately be instantiated.

Subsuming, we require:

$h_{\mathbf{FT}}$: Minimise the amount and the type cardinality of the transmission parameters.

The third heuristic is summarised in Table 6.3. According to the previous considerations, $\#ty_p^{op} =$ **MaxInf** is possible.

Notation	Name of Heuristic	Description	Motivation
$h_{\mathbf{FT}}$	**few transmission**	Minimise amount and type cardinality of transmission parameters.	*Small Interface*.

Mathematical Definition
$h_{\mathbf{FT}}(\mathbf{C}) := \textbf{let } T_{op} := (\#l^{-1}(op) * \#ty^{op}_{tr\_1} * \cdots * \#ty^{op}_{tr\_n}) \textbf{ in } \sum_{op \in l[\mathbf{C}]} T_{op}$

Table 6.3: Heuristic $h_{\mathbf{FT}}$: **few transmission**

6.1.4 Fourth Heuristic: Few Addressing

In correspondence to transmission parameters, Section 4.3.3 introduced the concept of address parameters to preserve the control flow within the decomposition. Again, these parameters increase the size of $E_{\mathbf{C}}$.

Contrary to transmission parameters, address parameters never have an infinite type. Thus, we can precisely determine the weight for these parameters, motivating separate measurements for both parameter types. We introduce a new heuristic, which mainly corresponds to the previous one, and we set the following objective:

$h_{\mathbf{FA}}$*: Minimise the amount and the type cardinality of the address parameters.*

Notation	Name of Heuristic	Description	Motivation
$h_{\mathbf{FA}}$	**few addressing**	Minimise amount and type cardinality of address parameters.	*Small Interface*.

Mathematical Definition
$h_{\mathbf{FA}}(\mathbf{C}) := \textbf{let } A_{op} := (\#l^{-1}(op) * \#ty^{op}_{ad\_1} * \cdots * \#ty^{op}_{ad\_n}) \textbf{ in } \sum_{op \in l[\mathbf{C}]} A_{op}$

Table 6.4: Heuristic $h_{\mathbf{FA}}$: **few addressing**

The mathematical definition for this heuristic corresponds to the one for **few transmission**, except that now $\#ty^{op}_{p} = \mathbf{MaxInf}$ is impossible. In the definition, *ad_i* denote *address* parameters.

6.2 Evaluation of Decomposition Heuristics

The previous section introduced several individually defined heuristics, which are possibly conflicting with each other. In order to evaluate the set of valid decompositions (or

solutions, as we will call them in the context of this chapter), the joint application of all heuristics is required. This obviously results in a trade-off between the specific requirements for *good* decompositions: for instance, assigning a high priority to heuristic $h_{\mathbf{ED}}$ will result in a set of cuts with potentially high value for heuristic $h_{\mathbf{CS}}$. The general problem is well known as the task of *multi-objective optimisation* [Ehr00, Zel74].

Besides this trade-off and despite allowing the specific heuristics to be scaled and thus prioritised, some decompositions or solutions can be neglected entirely. These are the ones for which one resulting component is on the scale of the original specification: here, compositional verification needs to deal with at least the same state space as non-compositional one.

Definition 6.2.1. *(Unreasonable decomposition)*
Let S be a CSP-OZ class specification, and let $\mathcal{C}$ denote the set of all valid *cuts of S. We say that $\mathbf{C} \in \mathcal{C}$ is* unreasonable*, if, and only if,*

$$\mathbf{Ph}_1 \cup \mathbf{Ph}_3 \cup \mathbf{C} = \mathtt{op}(N) \text{ or } \mathbf{Ph}_2 \cup \mathbf{C} = \mathtt{op}(N).$$

Regarding our heuristics, we immediately deduce that a decomposition is unreasonable, if, and only if, the sum over the values for the heuristics $h_{\mathbf{CS}}$ and $h_{\mathbf{ED}}$ is equal to the size of all operation nodes of the DG:

Lemma 6.2.2. *(Connection between unreasonable decompositions and heuristics)*
Let S be a CSP-OZ class specification, and let $\mathcal{C}$ denote the set of all valid *cuts of S. $\mathbf{C} \in \mathcal{C}$ is unreasonable, if, and only if,*

$$h_{\mathbf{CS}}(\mathbf{C}) + h_{\mathbf{ED}}(\mathbf{C}) = \#\mathtt{op}(N).$$

Proof. Immediate: any operation node is uniquely assigned to one of the sets $\mathbf{Ph}_1$, $\mathbf{Ph}_2$, $\mathbf{Ph}_3$ and $\mathbf{C}$. Furthermore, $h_{\mathbf{CS}}(\mathbf{C}) = \#\mathbf{C}$ and $h_{\mathbf{ED}}(\mathbf{C}) = |\#(\mathbf{Ph}_1 \cup \mathbf{Ph}_3) - \#\mathbf{Ph}_2|$ holds. □

Unreasonable decompositions will generally not be considered within our evaluation. For the further restriction of the set of valid cuts, we reason about *dominated* decompositions. Intuitively, they are outmatched by some other decomposition in *any* heuristic. In the context of multi-objective optimisation, the remaining solutions are called *Pareto-optimal* [Par71]. We will introduce the definition for our context, where we refer to the one from [DW04]. Let $h_1, \ldots, h_4$ denote the heuristics, as introduced in Section 6.1:

Definition 6.2.3. *(Weakly dominated decomposition [DW04])*
Let S be a CSP-OZ class specification, and let $\mathcal{C}$ denote the set of all valid *cuts of S. We say that $\mathbf{C} \in \mathcal{C}$ is* (weakly) dominated by $\mathbf{C}' \in \mathcal{C}$ *(with respect to $\{h_1, h_2, h_3, h_4\}$), if, and only if,*

$$(\forall\, i \in \{1,2,3,4\} \bullet h_i(\mathbf{C}') \leq h_i(\mathbf{C})) \wedge (\exists\, i \in \{1,2,3,4\} \bullet h_i(\mathbf{C}') < h_i(\mathbf{C})).$$

We illustrate the definition by an example.

Example 6.2.4. *Recall the candy machine specification from Section 2.2.1. Both, $\mathbf{C} = \{switch, abort\}$ and $\mathbf{C}' = \{switch\}$ denote valid cuts. For the evaluation of the different heuristics, we get:*

$$
\begin{array}{lcllcl}
h_{\mathsf{CS}}(\mathbf{C}) & = & 2 & h_{\mathsf{CS}}(\mathbf{C}') & = & 1 \\
h_{\mathsf{ED}}(\mathbf{C}) & = & 2 & h_{\mathsf{ED}}(\mathbf{C}') & = & 1 \\
h_{\mathsf{FT}}(\mathbf{C}) & = & \mathbf{MaxInf} & h_{\mathsf{FT}}(\mathbf{C}') & = & \mathbf{MaxInf} \\
h_{\mathsf{FA}}(\mathbf{C}) & = & 0 & h_{\mathsf{FA}}(\mathbf{C}') & = & 0
\end{array}
$$

Here, $h_{\mathsf{FT}}(\mathbf{C}) = h_{\mathsf{FT}}(\mathbf{C}') = \mathbf{MaxInf}$*, due to one transmission parameter of type* $\mathbb{N}$*. As* $h_i(\mathbf{C}') \leq h_i(\mathbf{C})$ *for any of the four heuristics and as* strictly smaller *holds for the first two,* $\{switch, abort\}$ *is weakly dominated by* $\{switch\}$*.*

Independent of the scaling, a weakly dominated cut never achieves the relatively best results. In the implementation of our decomposition approach, solutions dominated by other ones will thus accordingly be marked and can be suppressed. Note that even if highly unlikely, a dominated solution might still be the most efficient one. This is due to the respective property under interest, the specific characteristics of the model checker and the general nature of a heuristic approach, which is experience-based and only points the direction.

For the remaining *near-optimal* solutions, no general classification is possible. Yet, an elimination of all dominated cuts results in a smaller set of possible decompositions, which can then be further interpreted, according to the priority for each heuristic.

6.3 Candy Machine Revisited: Evaluation of Cuts

Next, we are interested in the evaluation of the set of all valid cuts of a specification based on our heuristics. We recall the case study of a candy machine from Chapter 2. Here, we restrict ourselves to the special case of a *single* cut from Definition 4.2.10 due to two reasons:

- The set of all possible general cuts is too large for an effective comparison.
- Defining two different cut sets is impractical, as the specification does not comprise any outer recursion.

Subsuming, there are 26 valid (single) cuts, which are depicted in Table 6.5. We additionally denote if the respective cut is, according to Definitions 6.2.1 and 6.2.3, non-reasonable or weakly dominated by another one.

No.	Cut	Reasonable?	Non-Dominated?
1	$\{abort, deliver, order, pay, payout, select, switch\}$	No	No
2	$\{abort, deliver, order, payout, select, switch, term\}$	No	Yes
3	$\{abort, deliver, order, payout, select, switch\}$	Yes	No
4	$\{abort, deliver, order, payout, select, term\}$	No	Yes
5	$\{abort, deliver, order, select, switch, term\}$	No	Yes
6	$\{abort, deliver, order, select, switch\}$	Yes	No
7	$\{abort, deliver, order, select, term\}$	No	Yes
8	$\{abort, deliver, payout, term\}$	No	Yes
9	$\{abort, deliver, term\}$	No	Yes

10	$\{abort, order, pay, payout, select, switch\}$	No	No
11	$\{abort, order, payout, select, switch\}$	Yes	No
12	$\{abort, order, select, switch\}$	Yes	Yes
13	$\{abort, pay, payout, switch\}$	No	No
14	$\{abort, payout, switch\}$	Yes	No
15	$\{abort, payout, term\}$	No	Yes
16	$\{abort, payout\}$	No	Yes
17	$\{abort, switch\}$	Yes	No
18	$\{abort, term\}$	No	Yes
19	$\{abort\}$	No	Yes
20	$\{deliver, order, select, switch, term\}$	No	Yes
21	$\{deliver, order, select, switch\}$	Yes	No
22	$\{deliver, order, select, term\}$	No	Yes
23	$\{deliver, term\}$	No	Yes
24	$\{order, select, switch\}$	Yes	No
25	$\{switch\}$	Yes	Yes
26	$\{term\}$	No	Yes

Table 6.5: Set of valid cuts for the candy machine

Even though the set of valid cuts is rather large, only *two* solutions are reasonable *and* non-dominated. These are $\{abort, order, select, switch\}$ and $\{switch\}$. In Chapter 7, we will compare both cuts.

6.4 Case Study: Two Phase Commit Protocol

In order to further illustrate and exemplify our decomposition technique, we introduce a second case study: a specification of the *Two-Phase-Commit Protocol* (TPCP) [BHG87, dRHH+01]. The purpose of the protocol is to guarantee consistency of N local sites (or *pages*) of a distributed database. Instructed by a coordinator process, the protocol results in either all pages committing their transaction or all pages aborting it. The basic system structure and communication is illustrated in Figure 6.1. As the name says, the protocol works in two phases:

Phase 1 - Commit-Request: The protocol starts with the coordinator process informing all participating pages about a *request* to commit the current transaction. Next, all pages *execute* the transaction and send a *vote* to the coordinator, dependent on whether the local transaction succeeded (*YES*) or failed (*NO*). The coordinator collects the votes and *decide*s to either *COMMIT*, in the case that all votes agree on *YES*, or to *ABORT* the transaction. Figure 6.2 illustrates the workflow of phase one for the coordinator and, for simplification, for one instance of *Page*.

Phase 2 - Commit: The coordinator *informs* all pages about the decision. All participating sites behave accordingly: an *abort* leads to an *undo* of the transaction, while a commit leads to *complet*ion. In any case, the sites output the *result* and send an *acknowledge*ment to the coordinator. An illustration is given in Figure 6.3.

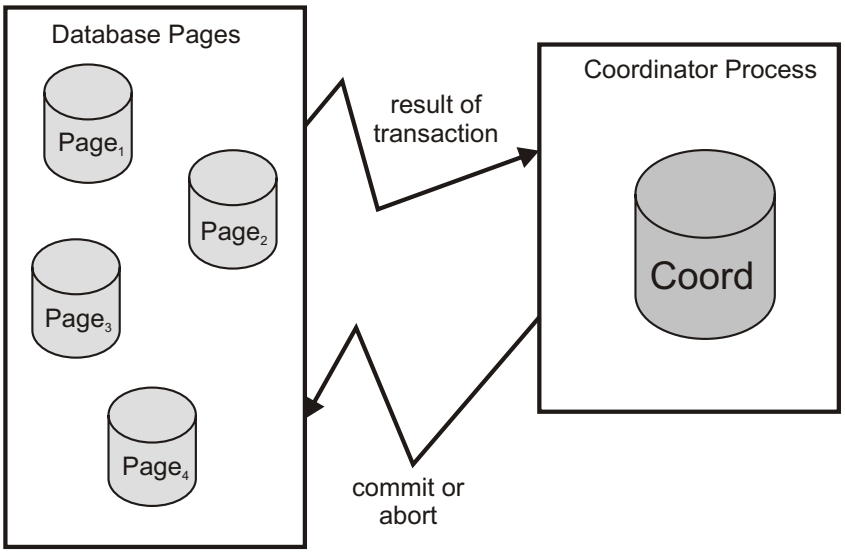


Figure 6.1: Illustration of the Two Phase Commit Protocol

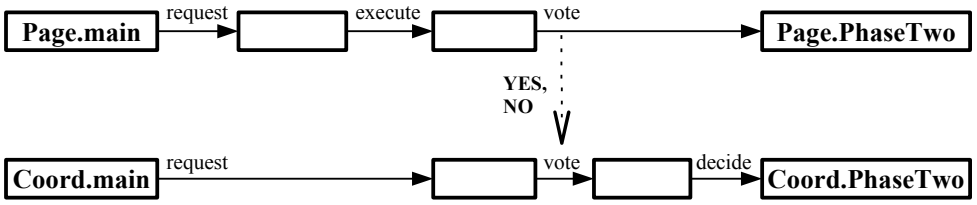


Figure 6.2: Phase one of the Two Phase Commit Protocol

Let N be the number of pages participating in the protocol, and let *Votes* and *Trans* be the following two base types:

$$Votes == \{YES, NO\}$$
$$Trans == \{COMMIT, ABORT\}$$

Here, *Votes* represents the possible votes of the pages, dependent on whether the transaction succeeded or not, whereas *Trans* describes the actual decision to either commit or abort the transaction.

The specification, as given in Figure 6.4, is the CSP-OZ class for the central coordinator. The ordering of events within the CSP part corresponds to Figures 6.2 and 6.3.

For the Object-Z part, the class' state space comprises two variables: *decC* of type *Trans*, for holding the final decision and *votes* of type $\mathbb{P}\, Votes$, for storing the votes of the different pages. The operation *vote* comprises an input parameter of type *Votes*, and its value is

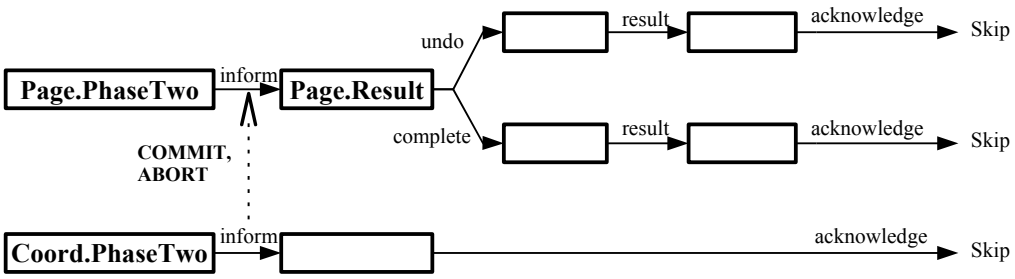


Figure 6.3: Phase two of the Two Phase Commit Protocol

Coord

`chan` *request* `chan` *vote* : [*vo*? : *Votes*] `chan` *decide*
`chan` *inform* : [*in*! : *Trans*] `chan` *acknowledge*

$$\texttt{main} \stackrel{c}{=} |||_{0<i\leq N}(request \rightarrow \texttt{Skip}); |||_{0<i\leq N}(vote?vo \rightarrow \texttt{Skip}); decide \rightarrow PhaseTwo$$

$$PhaseTwo \stackrel{c}{=} |||_{0<i\leq N}(inform?in \rightarrow \texttt{Skip}); |||_{0<i\leq N}(acknowledge \rightarrow \texttt{Skip})$$

decC : *Trans*
votes : $\mathbb{P}$ *Votes*

`Init`
$decC = ABORT$

`effect_`*request*
$\Delta(votes)$
$votes' = \varnothing$

`effect_`*vote*
$\Delta(votes)$
vo? : *Votes*
$votes' = votes \cup \{vo?\}$

`effect_`*inform*
in! : *Trans*
$in! = decC$

`effect_`*decide*
$\Delta(decC)$
if $(NO \in votes)$ **then** $decC' = ABORT$ **else** $decC' = COMMIT$

Figure 6.4: Two Phase Commit Protocol: *Coord* specification

added to the set *votes*. *decide* evaluates the set by assigning *decC* to *ABORT* in case that at least one page votes with *NO* and to *COMMIT* otherwise. Finally, *inform* sends the evaluation result to all participating pages by using an output parameter of type *Trans*.

The class *Coord* operates in parallel with *N* instantiations of the class *Page*, as given in Figure 6.5. The state space of the Object-Z part of *Page* holds two variables *decP* of type *Trans*, corresponding to *Coord.decC*, and *stable* of type $\mathbb{B}$, for representing a successful (*true*) or unsuccessful (*false*) execution of the transaction. *execute* nonderministically

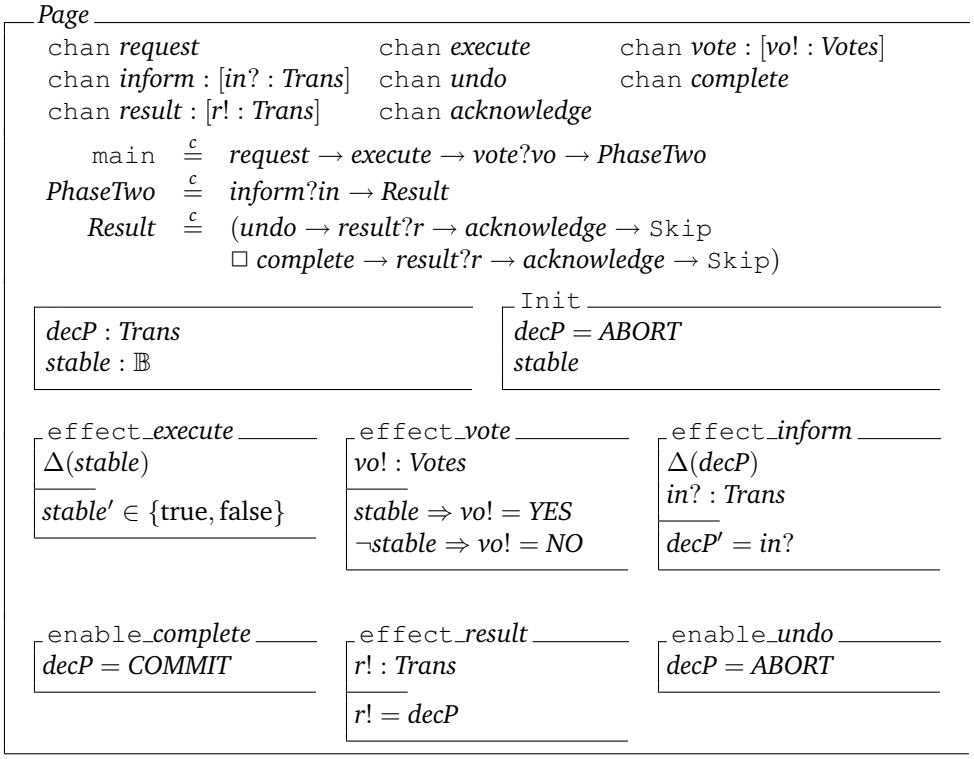


Figure 6.5: Two Phase Commit Protocol: *Page* specification

assigns a value to *stable*, dependent on which *vote* decides to either vote *YES* or *NO*. *inform* receives the decision to commit or abort the transaction, after which the specification either conducts a rollback (*undo*) or a permanent write (*complete*). Finally, the *result* of the transaction is communicated.

The full system is specified as

$$System = Coord \|_I (\ |||_{0<i\leq N} Page),$$

where $I = \{|\ request, vote, inform, acknowledge\ |\}$ denotes the synchronisation alphabet for both classes.

Again, we are interested in an evaluation of the set of all valid cuts. For simplicity, we again solely deal with single cuts. Independent of the number of pages, 42 valid cuts can be identified. These are given in Table 6.6, where an operation name is abbreviated by its first four letters. Whether a certain cut is dominated by another one depends on the value of N. Thus, within the following table, we assume $N \geq 3$. Overall, there exist 9 reasonable and non-dominated cuts for the specification of the Two Phase Commit Protocol.

No.	Cut	Reasonable?	Non-Dominated?
1	$\{ack, comp, deci, exec, info, resu, undo, vote\}$	No	No
2	$\{ack, comp, deci, info, resu, undo, vote\}$	No	No
3	$\{ack, comp, deci, info, resu, undo\}$	No	No
4	$\{ack, comp, info, resu, undo\}$	No	No
5	$\{ack, comp, resu, undo\}$	No	No
6	$\{ack, comp, resu\}$	No	No
7	$\{ack, resu, undo\}$	No	No
8	$\{ack, resu\}$	No	Yes
9	$\{ack\}$	No	Yes
10	$\{comp, deci, exec, info, requ, resu, undo, vote\}$	No	No
11	$\{comp, deci, exec, info, requ, undo, vote\}$	No	No
12	$\{comp, deci, exec, info, requ, vote\}$	No	No
13	$\{comp, deci, exec, info, resu, undo, vote\}$	Yes	No
14	$\{comp, deci, exec, info, undo, vote\}$	Yes	No
15	$\{comp, deci, exec, info, vote\}$	Yes	No
16	$\{comp, deci, info, resu, undo, vote\}$	Yes	Yes
17	$\{comp, deci, info, resu, undo\}$	Yes	Yes
18	$\{comp, deci, info, undo, vote\}$	Yes	No
19	$\{comp, deci, info, undo\}$	Yes	Yes
20	$\{comp, deci, info, vote\}$	Yes	No
21	$\{comp, deci, info\}$	Yes	No
22	$\{comp, info, resu, undo\}$	Yes	Yes
23	$\{comp, info, undo\}$	Yes	Yes
24	$\{comp, info\}$	Yes	Yes
25	$\{deci, exec, info, requ, undo, vote\}$	No	No
26	$\{deci, exec, info, requ, vote\}$	No	No
27	$\{deci, exec, info, undo, vote\}$	Yes	No
28	$\{deci, exec, info, vote\}$	Yes	No
29	$\{deci, exec, requ, vote\}$	No	No
30	$\{deci, exec, vote\}$	Yes	No
31	$\{deci, info, undo, vote\}$	Yes	No
32	$\{deci, info, undo\}$	Yes	No
33	$\{deci, info, vote\}$	Yes	No
34	$\{deci, info\}$	Yes	No
35	$\{deci, vote\}$	Yes	Yes
36	$\{exec, requ, vote\}$	No	No
37	$\{exec, requ\}$	No	No
38	$\{exec, vote\}$	Yes	No
39	$\{info, undo\}$	Yes	Yes
40	$\{info\}$	Yes	Yes
41	$\{requ\}$	No	No
42	$\{vote\}$	Yes	No

Table 6.6: Set of valid cuts for the TPCP

Some of the valid cuts are unreasonable. For instance, the decomposition corresponding to $\{\mathit{acknowledge}\}$ results in an equal size of the first component and the original specification.

It is interesting to note that some cuts, which one would intuitively expect to result in a decomposition effective for compositional reasoning, are dominated and thus ruled out. One example is the cut $\{\mathit{vote}\}$, which is dominated by $\{\mathit{inform}\}$. Dependent on the number of pages N, we get

$$\begin{array}{lcllcl} h_{\mathbf{CS}}(\{\mathit{vote}\}) & = & 2*N & h_{\mathbf{CS}}(\{\mathit{inform}\}) & = & 2*N \\ h_{\mathbf{ED}}(\{\mathit{vote}\}) & = & 6*N+1 & h_{\mathbf{ED}}(\{\mathit{inform}\}) & = & 2*N-1 \\ h_{\mathbf{FT}}(\{\mathit{vote}\}) & = & 8*N & h_{\mathbf{FT}}(\{\mathit{inform}\}) & = & 4*N \\ h_{\mathbf{FA}}(\{\mathit{vote}\}) & = & 2*N^3 & h_{\mathbf{FA}}(\{\mathit{inform}\}) & = & 2*N^3 \end{array}$$

The values for $h_{\mathbf{FA}}$ and $h_{\mathbf{CS}}$ are identical. However, $\{\mathit{inform}\}$ results in a distribution of the set of operation nodes closer to an *even distribution* than the one for $\{\mathit{vote}\}$. Additionally, $\{\mathit{vote}\}$ requires one transmission parameter of cardinality $\#\,\mathbb{P}\,\mathit{Votes} = 4$, reflecting the variable *Coord.votes*, whereas $\{\mathit{inform}\}$ sufficiently uses one additional parameter of cardinality $\#\mathit{Trans} = 2$, corresponding to *Page.decP* within the decomposition.

In contrast to the specification of a candy machine, the evaluation does not yield a small set of possible solutions. A thorough evaluation and comparison of the remaining set of reasonable and non-dominated cuts will be conducted in Chapter 7, where we introduce our implementation framework and give the experimental results for both case studies.

6.5 Discussion

As the name implies, a heuristic approach, setting up context-specific *rules-of-thumb*, cannot be expected to precisely and completely cover all aspects of the underlying problem, neither can it generate a single optimal solution. Hence, we keep the approach as least restrictive as possible by still guiding the engineer to head into the right direction.

First, our aim for introducing the described heuristics is a classification of the set of valid cuts or decompositions of a specification. Even though the implementation of our approach focusses on the model checker `FDR2`, we do not define the heuristics by exploiting its specific characteristics. By doing so, we keep the approach *independent* of a specific *model checker*.

Second, in contrast to the slicing technique, as introduced in [Brü08], we do not consider the property under interest. As the alphabet of the generated assumption during learning not only depends on the set of cut events but also on the set of events occurring in the verification property, it could be reasonable to integrate the alphabet of the property as well. However, we choose not to do so, as we want to keep the decomposition approach *independent* of a certain *verification property*.

Finally, the previously introduced heuristics can be applied in any compositional verification setting – they are not limited to the learning based framework, which we consider. This is due to the fact that we try to keep the state space of the decomposition (and thus the interdepences between both components) small, which is a reasonable strategy, *independent* of any *compositional verification framework*.

Yet, the following question remains: why do the previously defined heuristics most likely result in a set of *practical* solutions?

We investigated the different possibilities, causing a large state space, which needs to be explored during model checking. Here, we referred to two certain paradigms, which are generally valid for compositional verification [CAC06, dRHH+01, GL91, CGP03]: a strong connection between both components results in a high memory consumption and an increased run-time during verification. In addition, large components cause a large state space, which needs to be built up during model checking. The previous heuristics are closely related to both paradigms, as they investigate the definition of our decomposition technique and evaluate different possibilities to keep the cohesion between the components and their individual state spaces relatively small.

The different heuristics cannot be seen as equally important for any kind of specification. In addition, they conflict with each other. For instance, often, the larger the cut size for a decomposition, the smaller the size difference between both components, simply because the cut is neglected for the second heuristic.

Therefore, the actual evaluation of the set of valid decompositions must not be restricted to the specific values, given by the mathematical definitions of the heuristics. In fact, as we will see in Chapter 7, our implementation framework allows the user to *prioritise* certain heuristics by computing the weighted sum over all values.

However, in order to not mislead the user, several solutions can be neglected. We discussed this topic in Section 6.2: an evaluation of valid decompositions comparatively worse than other ones, that is, weakly dominated ones, is unnecessary. The same applies to unreasonable decompositions.

Summarising, the approach presented in this chapter automatically restricts the set of valid solutions as much as possible. This is done by eliminating those decompositions, which are impractical with respect to our heuristics or the generated state space size. Due to the nature of a heuristics-based approach, human intervention is still required for an evaluation of the remaining set of valid decompositions. However, this set is comparatively small in relation to the set of all valid cuts.

6.6 Related Work

Several works from different areas investigate heuristic approaches to cope with the state explosion problem during model checking. The work closest to ours is presented in [Nam07]. For learning-based compositional verification for models, described as symbolic transition systems (STS), the author chooses to partition a given system into several components, based on an algorithm for *hypergraph partitioning* [KK99]. The approach follows the general idea for an even distribution of the state variables of the STS and also aims at a minimisation of the interdependences between the components. The decomposition is performed fully automatically, not allowing the user to guide the framework to a potentially better partitioning, not complying to the static requirements. In addition, the author does not consider the control flow or a dependence analysis, and the approach does not take the alphabet of the generated assumptions into account.

In order to cope with the state explosion problem during model checking of systems already composed of several components, in [SLU89], the authors present several alternative heuristic rules to reduce the state space of the system, focusing on the LTS semantics of a system. The work presented in [TJ02] follows a similar approach by, for instance, developing heuristics to fusion states or transitions or eliminating redundant states.

In the context of the L^* algorithm, in [GP09], the authors present a strategy for interface generation of software components. They implemented their approach for *Java PathFinder* (JPF) [NAS], a verification framework for Java byte code. Based on their learning framework for interface specifications, the authors also implemented assume-guarantee reasoning in JPF. JPF itself uses different search heuristics for an effective identification of possible bugs, eventually complementing compositional verification.

Further away from our approach, Dirks and Olderog [OD08] investigate the specification and the model checking of *real-time* systems. In their semantic domain, the first author developed an approach for heuristics-based planning and model checking [Die05]. Another heuristics-based approach, in order to more efficiently direct a model checker to potential counterexamples, is *directed* model checking [ESB+09]. Edelkamp et. al investigate directed model checking for `SPIN` [Hol03].

Multicriteria optimisation is an extensively researched area with a lot of different textbooks and articles giving a profound overview and insight on the topic [Ehr00, DW04, SNT85]. We concentrate on the definition of *Pareto-optimality* which was introduced in [Par71] and we restrict ourselves to *discrete* optimisation, that is, we do not consider real values within our heuristics.

This concludes the current chapter. The next chapter will introduce the implementation of our approach, including the *modelling*, the *heuristics-based decomposition* of a system and a subsequent *direct* or *learning-based compositional verification*. In addition, we evaluate the non-dominated and reasonable decompositions for both case studies and provide some significant experimental results.

7 Implementation and Experimental Results

Contents

7.1 Syspect 184
7.1.1 Class Diagrams 184
7.1.2 State Machines 186
7.1.3 Component Diagrams 187
7.1.4 Export to CSP-OZ 187
7.2 Decomposition Framework for Syspect 188
7.2.1 Decomposition Plug-In 189
7.2.2 Mass Validation 191
7.2.3 Model Checking with FDR2 and the CSPLChecker 192
7.2.4 Counterexample Analysis 196
7.2.5 Overall Workflow 198
7.3 Experimental Results 200
7.3.1 Overview 200
7.3.2 Verification Results for the Candy Machine 201
7.3.3 Verification Results for the Two Phase Commit Protocol 204
7.3.4 Verification Results for the Number Swapper 207
7.3.5 Discussion 207

The previous chapters introduced an approach for the decomposition of formal specifications, allowing for an application of compositional verification. Furthermore, we presented several heuristics for a classification of all valid decompositions. In order to substantiate our method and to measure its effectiveness, the technique has been implemented, and several case studies have been evaluated.

The present chapter describes the implementation framework for the theory of the previous chapters. Section 7.1 introduces *Syspect* [Sys06], a graphical modelling environment for CSP-OZ specifications, developed by the research group "Correct System Design" in Oldenburg. By using one of our case studies, we give a short overview on Syspect's different diagram types for modelling different aspects of a specification. The following Section 7.2 presents our context-specific extensions, realised to integrate the decomposition approach into Syspect. In the last section, the experimental results for three case studies, the candy machine from Section 2.2, the Two Phase Commit Protocol from Section 6.4 and the number swapper from Section 4.4, are given. We measure the different optimal and reasonable cuts by comparing direct model checking with `FDR2` and compositional (learning-based) model checking. Finally, we discuss the results and

draw some conclusions: some context-specific characteristics for *good* decompositions are pointed out, and we comment on when the application of our technique most likely results in a speed-up of model checking.

7.1 Syspect

The underlying platform for the implementation of our decomposition approach is the **Sy**stem **Spec**ification **T**ool (Syspect, [Sys06]). Syspect is a graphical and UML-based modelling environment for specifications, written in the integrated formalism CSP-OZ-DC [Hoe06]. By extending the language of CSP-OZ with the formalism *Duration Calculus (DC)* [ZH04], CSP-OZ-DC additionally allows to reason about *real time aspects* of a software model. Within this thesis, we do not consider DC. However, as CSP-OZ is naturally embedded into CSP-OZ-DC by simply declaring the DC-part to be empty, we can use Syspect to model CSP-OZ specifications as well.

Syspect has been developed within a student project, carried out at the research group "Correct System Design" in Oldenburg. The basis for their work is a specific UML profile for CSP-OZ, described in [MORW08]. A UML model can then be translated into a CSP-OZ specification. One focus for the definition of the UML model is the choice of a suitable subset of the UML, which is expressive enough to represent a significant part of CSP-OZ. In order to achieve this, the profile uses three different diagrams of the UML, namely

- class diagrams,
- state machines and
- component diagrams.

Next, we will shortly introduce the Syspect representation of the different diagram types by modelling the specification of the *Two Phase Commit Protocol* from Section 6.4. For a more detailed introduction into Syspect and the underlying UML profile, we refer to [Sys06, MORW08, Brü08].

7.1.1 Class Diagrams

In order to describe the *static* behaviour of a system specification, UML *class diagrams* [Obj05] can be used. Such a diagram comprises the specification's classes including their attributes: data variables (according to the state variables of the Object-Z part of a class) and methods (corresponding to the operations of the CSP-OZ class). Additionally, the definition of relationships between classes is possible: for the purpose of connecting classes and class-interfaces, different associations, such as aggregation or composition, can be used. These relationships represent the specification's composition- and synchronisation structure.

For the specification of the *Two Phase Commit Protocol*, the class diagram contains both classes *Coord* and *Page*. An interface *ISyncCoordPage* describes the set of synchronised

operations of both classes. One additional class *System* is defined, representing the composition of *Coord* and *Page*, without defining additional attributes.

Figure 7.1 displays a screenshot of Syspect, showing the class diagram of the TPCP within the Syspect class diagram editor.

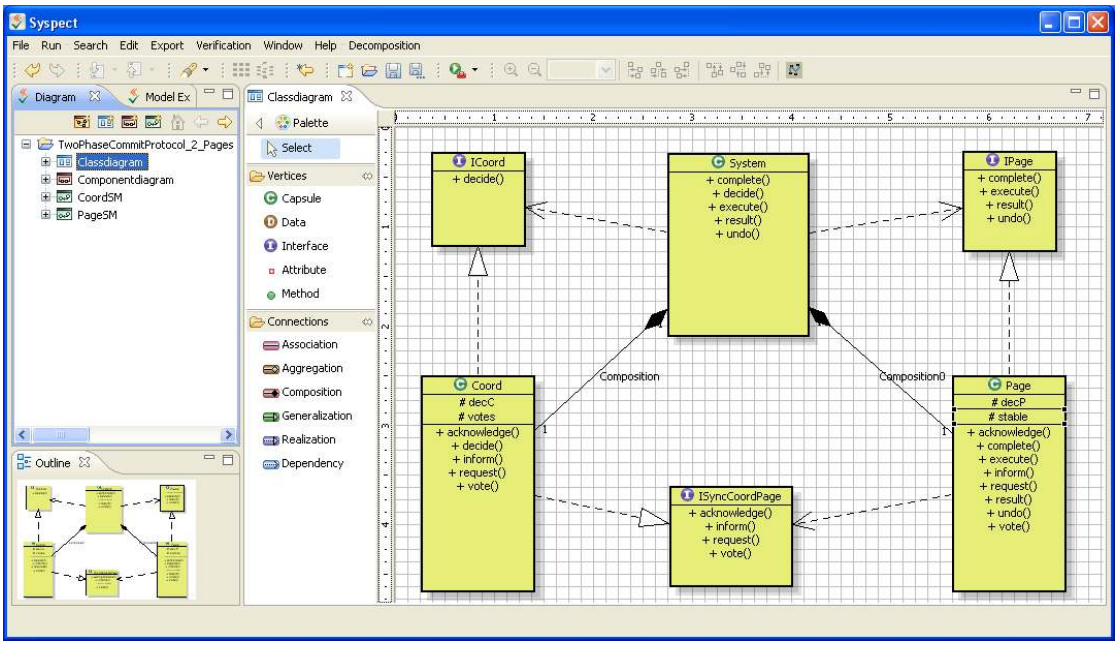


Figure 7.1: Syspect class diagram for the TPCP

Within a certain class, its set of variables and operations can be defined. The types of the variables and the behaviour of an individual operation can be described within the associated property view. In our example, both base types *Votes* and *Trans* are represented by $\mathbb{B}$, the set of boolean variables. Figure 7.2 shows the property view, associated with the operation *Page.inform*.

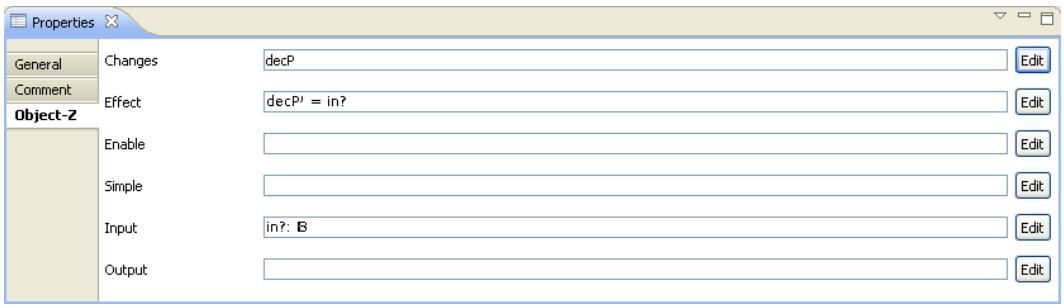


Figure 7.2: Syspect property view for the operation *Page.inform*

7.1.2 State Machines

UML *state machines* are defined for representing the CSP parts of the individual classes of a CSP-OZ specification. Transitions of a state machine are labelled with an event corresponding to the associated class, or they are unlabelled for representing non-determinism. States are either

- ordinary states, representing a CSP process,
- initial states, representing the specific *initial* process `main`,
- final states, representing successful *termination*, that is, the process `Skip`, or
- complex states, containing a number of regions for modelling concurrency, that is, *interleaving* of several processes in terms of CSP.

In order to describe (non-deterministic- or deterministic-) choice, branching can be used.

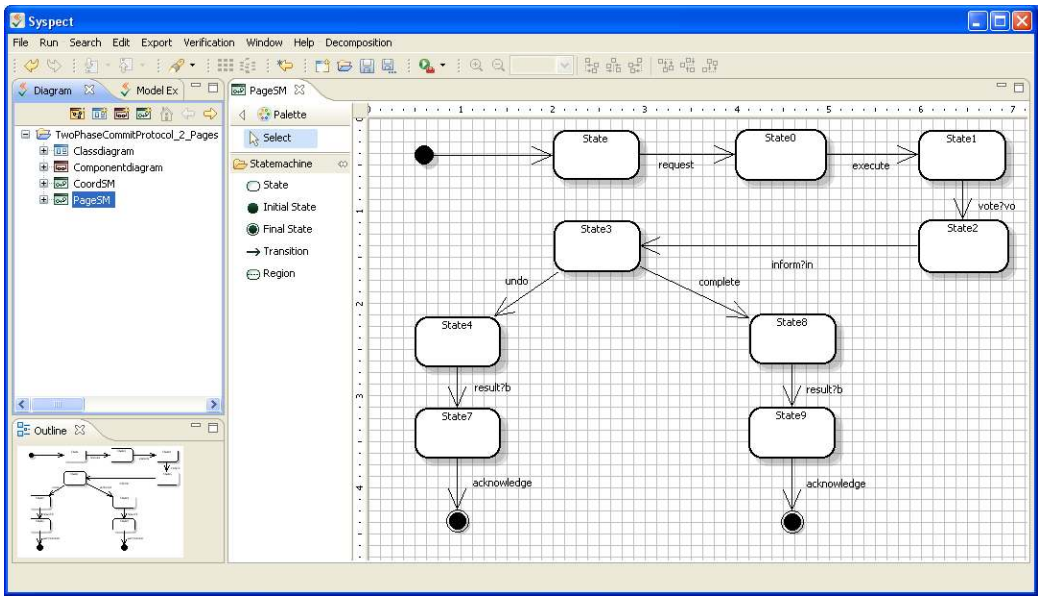


Figure 7.3: Syspect state machine for the class *Page* of TPCP

For the TPCP, there are two state machines, one corresponding to *Page*.`main` and one describing *Coord*.`main`. Figure 7.3 shows the state machine for *Page*.`main`. As the process *Coord*.`main` comprises interleaving of several processes, complex states are required. Here, we set $N := 2$, that is, the specification comprises two instances of class *Page*. Therefore, two regions are used, corresponding to the processes

$$|||_{i \in \{1,2\}} (op \rightarrow \texttt{Skip})$$

for $op \in \{request, vote, inform, acknowledge\}$. The according state machine is given in Figure 7.4.

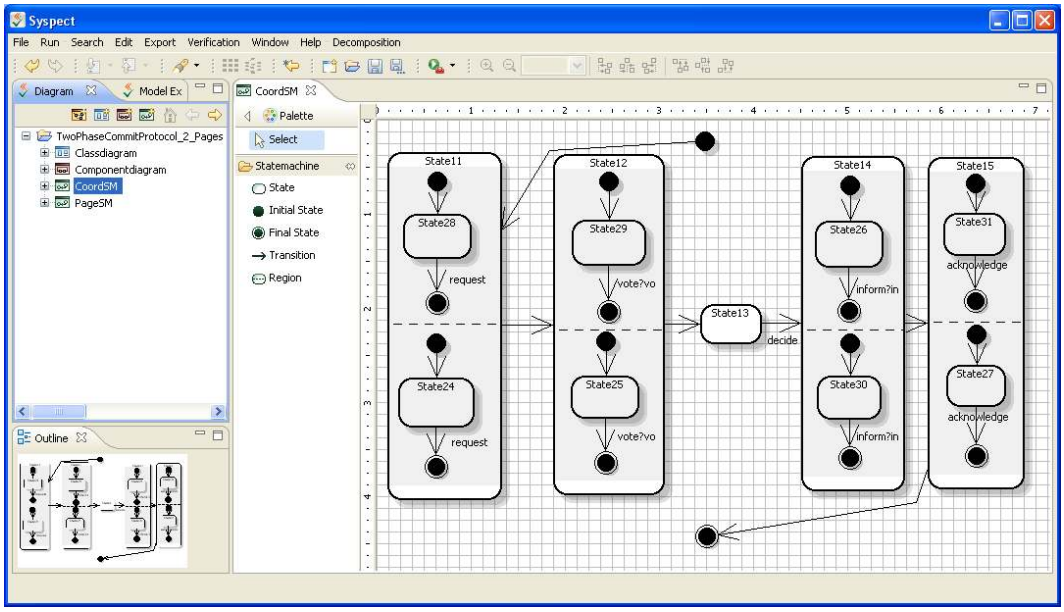


Figure 7.4: Syspect state machine for the class *Coord* of TPCP

7.1.3 Component Diagrams

The component diagram of a specification complements the class diagram and describes the *composition and instantiations* of its different constituents. Intuitively, it represents the overall system composition, that is,

$$System = Coord \|_I (\|\|_{i \in \{1,2\}} Page)$$

for the overall specification of the TPCP in the case of $N = 2$. Complementary to the class diagram, the number of instances of *Page* is specified, along with the associations between all class instances, based on the interface connections.

Figure 7.5 shows the component diagram for the *Two Phase Commit Protocol*. It describes that both instances of *Page* synchronise with the sole instance of *Coord* via the interface *ISyncCoordPage*. Conjointly, this synchronisation yields the *System*-class.

7.1.4 Export to CSP-OZ

Syspect provides an export functionality for the translation of an UML model into a CSP-OZ representation of the model. Here, a translation into various formats can be carried out. In this thesis, we are solely concerned with the LaTeX-export of a CSP-OZ specification: Syspect allows the generation of LaTeX mark-up, conforming to [ISO00] and the style file `csp-oz.sty`, as documented in [Fis99]. Within our verification framework, the export will be further processed and translated into the input language of the model checker `FDR2`.

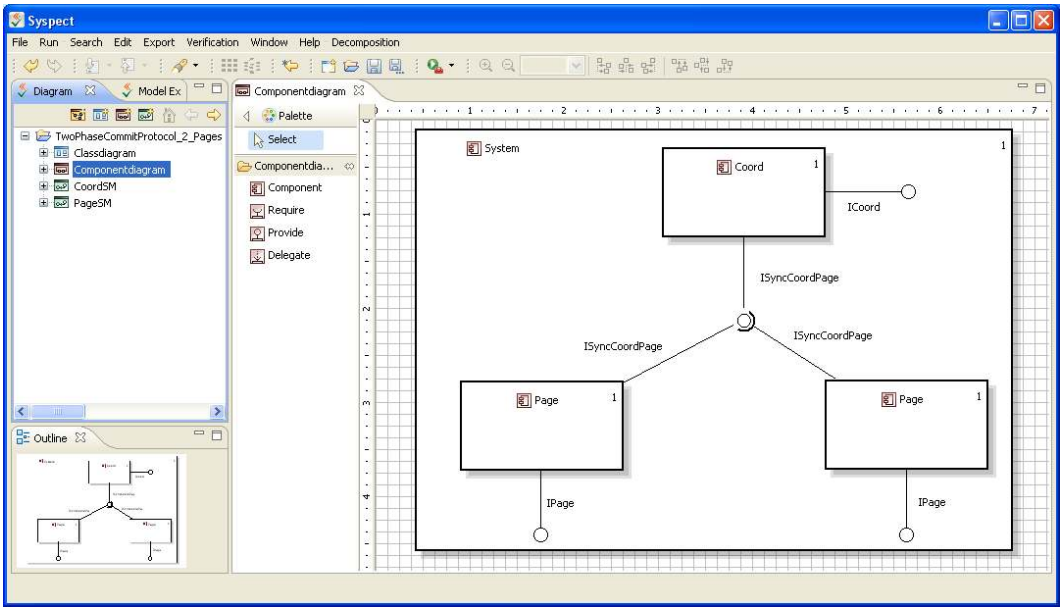


Figure 7.5: Syspect component diagram for the TPCP

7.2 Decomposition Framework for Syspect

After a brief introduction into Syspect, we will now describe the various additional features and extensions of Syspect, which have been developed in order to provide tool support for our decomposition technique. Namely, we present

- an implementation for the *decomposition* of a specification, based on the selection and validation of a (single) cut, the fragmentation of the specification's dependence graph and the subsequent decomposition of the specification itself, both according to Chapter 4 (Section 7.2.1),
- a *mass validation* framework to efficiently compute the set of *all* valid cuts, sorting out unreasonable and (weakly) dominated decompositions and scaling the remaining decompositions, based on the definitions from Chapter 6 (Section 7.2.2),
- an integration of the model checker `FDR2` [For05] into Syspect, including a *compiler* from the Syspect export to the input language of `FDR2`, along with an interface to an implementation of the *learning-based compositional verification framework* as presented in Chapter 3 (Section 7.2.3) and
- a *counterexample analysis* for visualising error traces, possibly generated by `FDR2`, within the Syspect model (Section 7.2.4).

Figure 7.6 sketches the overall workflow. Given a Syspect specification of the software model, a user can choose

- to generate the dependence graph of a specification, manually select a cut and – in case of a valid cut – accordingly decompose the specification or

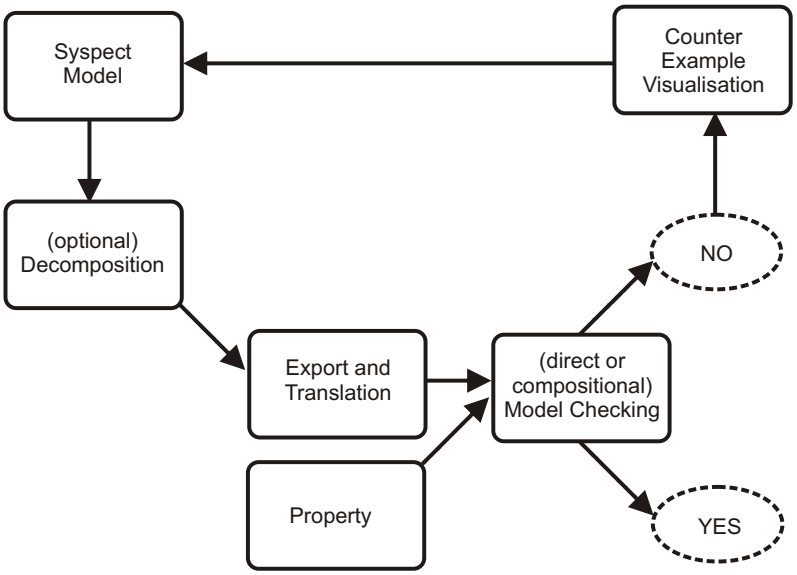


Figure 7.6: Toolchain for the verification framework

- initiate a computation of the set of all valid cuts, evaluate them with respect to our heuristics, select a specific cut and accordingly decompose the specification or
- not to decompose the specification at all.

As a next step, an export and subsequent compilation of the (decomposed) specification to the input language of the model checker `FDR2` can be carried out. Afterwards, either compositional model checking or direct model checking can be applied, additionally requiring the definition of the verification property under interest. A possible counterexample is visualised within the Syspect model.

Next, we give a survey over the specific extensions of Syspect. We close this section with a more detailed description of the verification framework by using UML activity diagrams [Obj05].

7.2.1 Decomposition Plug-In

In Chapter 4, we defined a cut of a dependence graph, yielding a decomposition of the underlying specification. The foundation for the corresponding integration into Syspect is the *decomposition plug-in*, developed by Klaus Herbold as part of his diploma thesis [Her09]. The plug-in can optionally be used within Syspect, allowing the user to visualise the dependence graph of a specification and select a set of operation nodes. Afterwards, the selected cut-candidate is validated against the different correctness criteria from Section 4.2.2. To this end, the decomposition plug-in comprises the selection and validation of a *single* cut. In case of an invalid cut, the responsible set of violations is displayed. Otherwise, the user can proceed to decompose the specification. The decomposition is

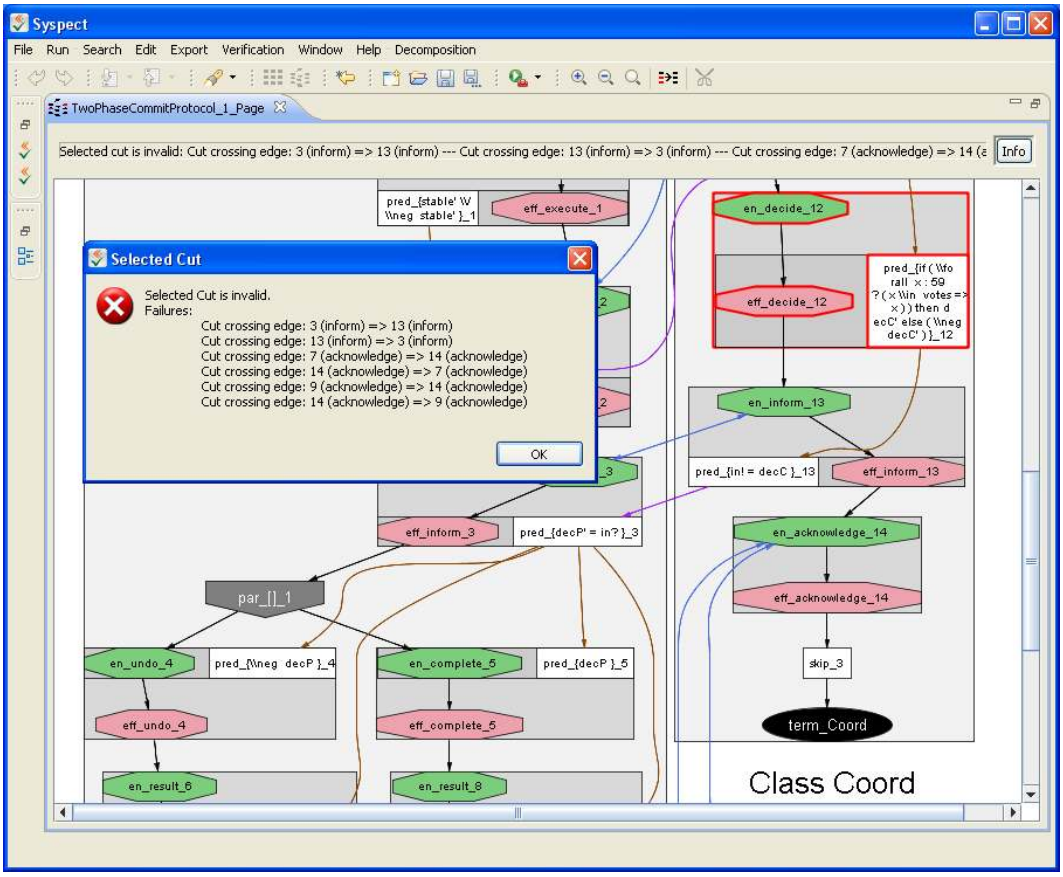


Figure 7.7: Screenshot of a selected invalid cut

carried out by applying the definitions from Section 4.3 and the addressing algorithm from Section 5.1.

Figure 7.7 shows a screenshot of an excerpt of the DG of the Two Phase Commit Protocol, as it is displayed within Syspect. For illustration purposes, we refer to the specification solely comprising *one* instance of the class *Page*. The visualisation builds up on the implementation of Brueckner's definition of the DG, which was carried out within the *slicing plug-in*, developed by Sven Linker [Brü08]. The graph is defined according to our modifications of the DG, as given in Section 2.3.4.

In general, a user can interact with the displayed DG and select a set of operation nodes. In the example, as displayed in the screenshot, the single node *Coord.decide* is selected. According to the correctness criteria, {*Coord.decide*} does not define a valid (single) cut: no nodes within the DG of *Page* are selected. Thus, its set of operation nodes is assigned to $\mathbf{Ph}_1$, and several synchronisation edges, connecting nodes from the DGs of *Page* and *Coord*, violate the correctness criterion **no crossing**. In line with the selected cut, these violations are indicated.

In the event of the selection of a valid cut, the validity is displayed, and a further decomposition with respect to the cut can be carried out. Here, several options are

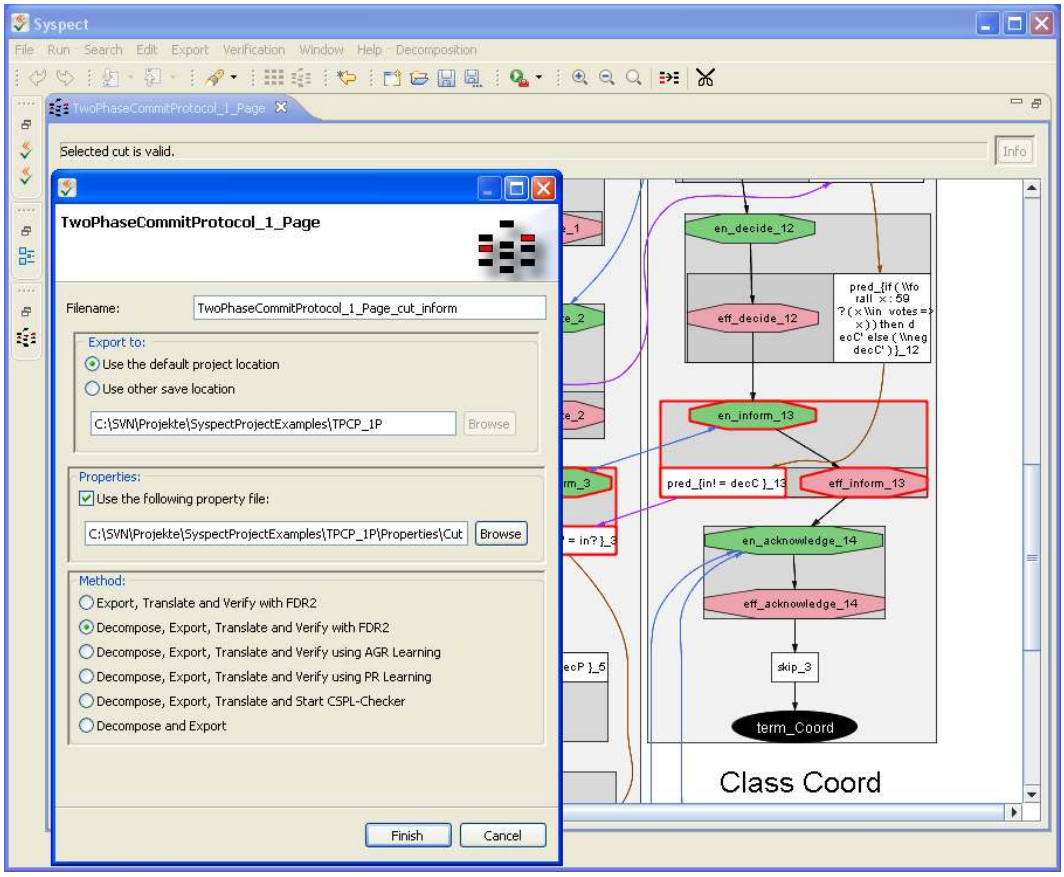


Figure 7.8: Screenshot of the decomposition options after selection of a valid cut

possible. Figure 7.8 displays the dialogue box after the selection of the correct single cut $\{Coord.inform, Page.inform\}$. The various options regarding the export and model checking will be explained in Section 7.2.3. A decomposition of S results in LaTeX mark-up for $S_1 \parallel S_2$, which can then further be processed. In order to generate a valid decomposition, transmission parameters and addressing parameters are added.

Within the plug-in, several features and optimisations are implemented. For instance, according to the correctness criterion **all-or-none** from Section 4.2.2, either all or no nodes with the same operation name have to be contained in a cut. Thus, to facilitate a cut selection, in case that a user picks an operation node, all correspondingly named nodes are automatically selected. For more details on the implementation, we refer to [Her09].

7.2.2 Mass Validation

The previous section introduced the general functionality of the Syspect decomposition plug-in. A user can select and deselect operation nodes within a specification's DG, until he identified a valid cut, for which he chooses to carry out a decomposition.

The larger the DG of a specification, the more tedious becomes a manual search for a cut. Moreover, according to Chapter 6, a valid cut must not automatically yield a decomposition suitable for an application of compositional reasoning.

In order to facilitate the choice of a valid cut and to guide the user to a decomposition most likely outmatching the original model in terms of model checking run-times, Meik Piepmeyer developed an extension of the decomposition plug-in, called the *mass validation* framework. Within his diploma thesis [Pie10], he mainly investigated the following question.

Given the DG of a specification, how can the set of all valid cuts efficiently be computed?

This question particularly becomes relevant if the DG comprises a large set of operation nodes: assuming $\#\mathtt{op}(N) = k$, the number of cut-candidates is 2^k. Piepmeyer showed that the general problem of identifying all valid cuts is *NP-complete* [Coo71]. However, he developed and implemented several strategies and algorithms to efficiently validate a set of operation nodes against the various correctness criteria. One of his strategies uses a SAT solver [PBG05].

Additionally, Piepmeyer implemented the different heuristics from Chapter 6, along with the identification of all unreasonable and dominated cuts. In the latter case, one of the dominating cuts is displayed. Unreasonable cuts and dominated cuts can be removed from the set of all valid cuts, and the remaining cuts can be scaled according to the different heuristics.

Figure 7.9 shows a screenshot of the mass validation framework after choosing to compute all valid cuts for the TPCP for three instances of *Page*. Unreasonable cuts are marked with a minus, whereas a plus signalises reasonable cuts. In addition, optimal cuts are indicated by a small histogram. Furthermore, the amounts of valid cuts, optimal cuts and reasonable cuts are displayed. According to the results of Chapter 6, the amount of cuts which are both, optimal and reasonable, is equal to 9.

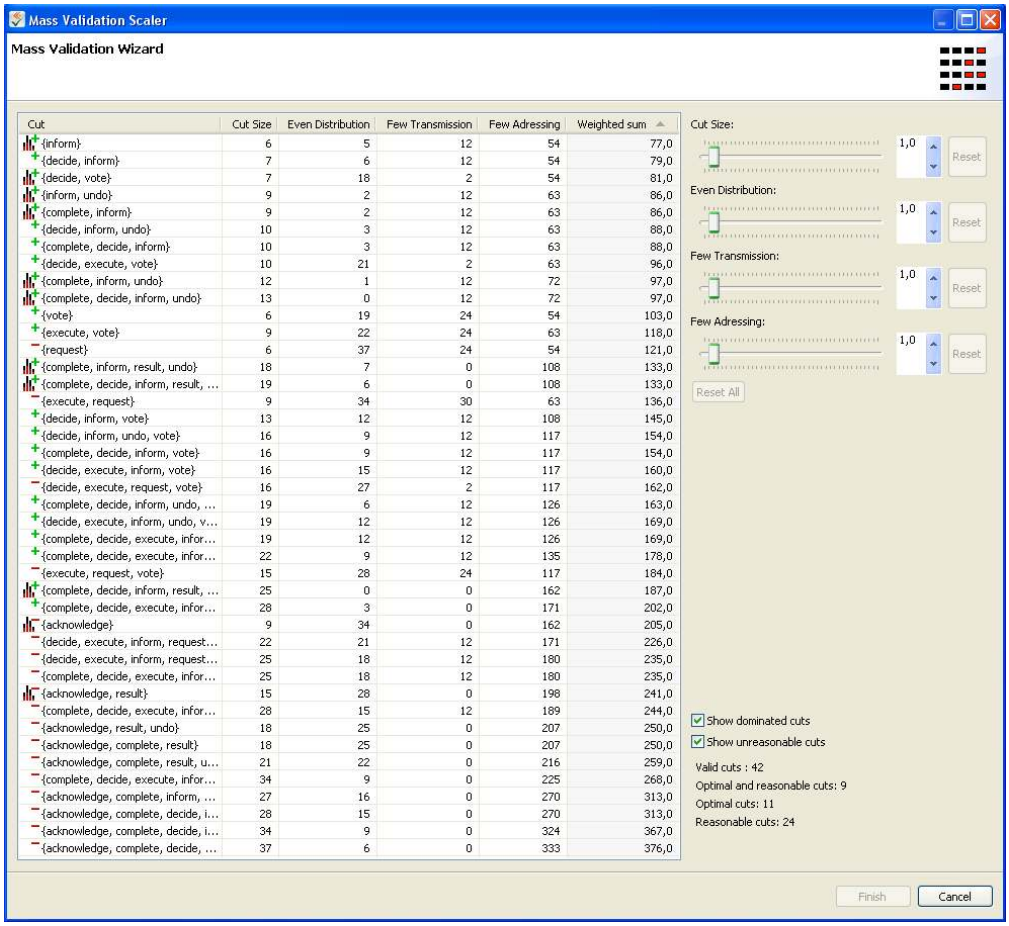
In order to further classify the set of optimal and reasonable cuts, to this end, the *weighted sum* over all heuristics is used as the scaling function. In the example, in case that all heuristics are equally rated, $\{inform\}$ obtains the smallest value.

After scaling the different heuristics to identify a subjectively optimal cut, the user can proceed to decompose the specification and model check the result. For more details on the mass validation, see [Pie10].

7.2.3 Model Checking with FDR2 and the CSPLChecker

Following up on the selection of a valid cut and a corresponding decomposition of the model, we aim at an evaluation of direct model checking in comparison to the compositional one. The decomposition approach of this thesis is not restricted to a particular model checker. Yet, we require an existing translation from CSP-OZ to the respective input language.

In order to evaluate the effectiveness of our theory, we choose the CSP model checker `FDR2` (**F**ailure **D**ivergence **R**efinement [For05]), developed by Formal Systems (Europe) Ltd. Several reasons substantiate this choice. First, `FDR2` is the most commonly used



Cut	Cut Size	Even Distribution	Few Transmission	Few Adressing	Weighted sum
{inform}	6	5	12	54	77,0
{decide, inform}	7	6	12	54	79,0
{decide, vote}	7	18	2	54	81,0
{inform, undo}	9	2	12	63	86,0
{complete, inform}	9	2	12	63	86,0
{decide, inform, undo}	10	3	12	63	88,0
{complete, decide, inform}	10	3	12	63	88,0
{decide, execute, vote}	10	21	2	63	96,0
{complete, inform, undo}	12	1	12	72	97,0
{complete, decide, inform, undo}	13	0	12	72	97,0
{vote}	6	19	24	54	103,0
{execute, vote}	9	22	24	63	118,0
{request}	6	37	24	54	121,0
{complete, inform, result, undo}	18	7	0	108	133,0
{complete, decide, inform, result, ...	19	6	0	108	133,0
{execute, request}	9	34	30	63	136,0
{decide, inform, vote}	13	12	12	108	145,0
{decide, inform, undo, vote}	16	9	12	117	154,0
{complete, decide, inform, vote}	16	9	12	117	154,0
{decide, execute, inform, vote}	16	15	12	117	160,0
{decide, execute, request, vote}	16	27	2	117	162,0
{complete, decide, inform, undo, ...	19	6	12	126	163,0
{decide, execute, inform, undo, v...	19	12	12	126	169,0
{complete, decide, execute, infor...	19	12	12	126	169,0
{complete, decide, execute, infor...	22	9	12	135	178,0
{execute, request, vote}	15	28	24	117	184,0
{complete, decide, inform, result, ...	25	0	0	162	187,0
{complete, decide, execute, infor...	28	3	0	171	202,0
{acknowledge}	9	34	0	162	205,0
{decide, execute, inform, request...	22	21	12	171	226,0
{decide, execute, inform, request...	25	18	12	180	235,0
{complete, decide, execute, infor...	25	18	12	180	235,0
{acknowledge, result}	15	28	0	198	241,0
{complete, decide, execute, infor...	28	15	12	189	244,0
{acknowledge, result, undo}	18	25	0	207	250,0
{acknowledge, complete, result}	18	25	0	207	250,0
{acknowledge, complete, result, u...	21	22	0	216	259,0
{complete, decide, execute, infor...	34	9	0	225	268,0
{acknowledge, complete, inform, ...	27	16	0	270	313,0
{acknowledge, complete, decide, i...	28	15	0	270	313,0
{acknowledge, complete, decide, i...	34	9	0	324	367,0
{acknowledge, complete, decide, ...	37	6	0	333	376,0

Figure 7.9: Screenshot of the mass validation framework

CSP model checker: at the time of writing his book [Ros98], Roscoe called it the *chief proof and analytic tool for CSP*, and this fact did not change over the recent years. Second, Wonisch [Won08] implemented the assume-guarantee-based learning framework based upon `FDR2`. Finally, `FDR2` is well suited for our purpose, due to an already existing translation from CSP-OZ to the input language of `FDR2` [FW99].

The tool inputs process specifications, written in a machine-readable dialect of CSP, called CSP_M. As the underlying verification concept, `FDR2` uses *refinement checks*: given two CSP-processes S and *Prop*, the refinement $\mathit{Prop} \sqsubseteq S$ can be evaluated within CSP's different semantic models. In this thesis, we are solely concerned with checking *trace* inclusion and do not consider any other refinement checks. For more details on `FDR2`, we refer to [Ros98, For05], with the latter reference comprising a full documentation of `FDR2` along with the syntax of CSP_M.

Already, several works investigated a translation from either CSP-OZ or a related formalism to the input language of `FDR2`. For instance, in [MS01], the authors present a translation from CSP-Z to CSP_M. In her diploma thesis and simultaneously to the development of Syspect, Stamer [Sta06] investigated a translation to CSP_M for models

```
N = {1,2}

pages = card(N)

PROP  = PC(pages)

PC(0) = ||| x:N @ (complete -> SKIP)
PC(i) = vote.true -> PC(i-1)
     [] vote.false -> PU(i-1)

PU(0) = ||| x:N @ (undo -> SKIP)
PU(i) = vote?j -> PU(i-1)

SPEC  = (S_1
        [ {|request,execute,vote,decide,inform|} ||
          {|inform,undo complete,result,acknowledge|} ]
         S_2)
        \ {|request,execute,decide,inform,result,acknowledge|}

assert PROP [T= SPEC
```

Figure 7.10: Correctness requirement for the TPCP in terms of CSP_M

specified in the UML profile for CSP-OZ [MORW08]. Obviously, such a translation has certain limits: it is clearly restricted by the expressiveness of the input language of the model checker. Moreover, as CSP-OZ exemplary allows to use infinite and underspecified data types, such as basic type definitions, a translation is limited to a subset of the integrated formalism.

In our context, we target the translation of the Syspect-LaTeX-export to CSP_M. As part of his work as a student assistant, Wonisch implemented a corresponding compiler. The translation builds up on the definition from [FW99] and thus, the CSP_Z semantics, as given in Section 2.2.4. Some of the accomplishments in the compiler development include the support to translate finite sequences and various mathematical tool kit functions, along with axiomatic definitions.

Some restrictions have to be applied in order to model the different case studies within Syspect and to allow for a translation to CSP_M. For the case study of a candy machine, we require sequences of finite length, for which we define a corresponding constant. For specifying the Two Phase Commit Protocol, both base types are mapped to $\mathbb{B}$. For a translation of the Syspect export, a user needs to specify the maximal value for an element of $\mathbb{Z}$. By default, this value is set to 5, meaning that $\mathbb{Z}$ is mapped onto the set $\{-5,\ldots,5\}$ within the CSP_M-script (and, accordingly, $\mathbb{N}$ mapped onto $\{0,\ldots,5\}$). The maximal integer is consistently used within the mass validation framework, where we implicitly set **MaxInf** to this specific value.

Besides the actual specification, model checking additionally requires a *verification property*. Currently, the user manually needs to define this property in terms of CSP_M.

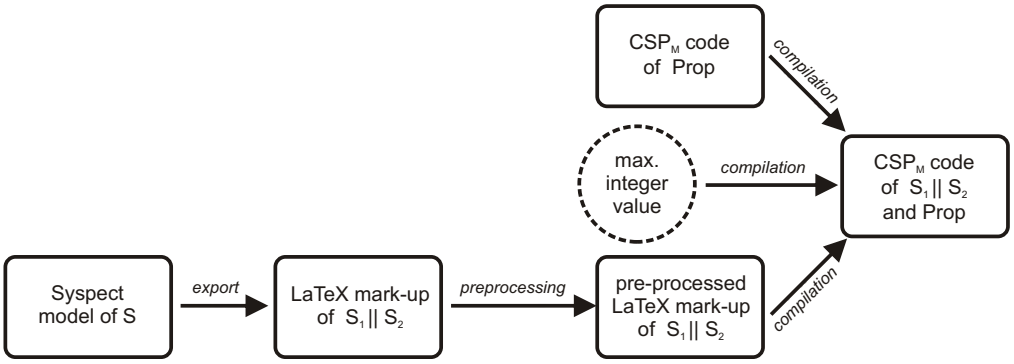


Figure 7.11: Compilation from LaTeX to CSP_M

Moreover, he needs to declare an *assertion*, specifying the individual trace refinement under investigation. Figure 7.10 defines a CSP process *Prop*, specifying a correctness requirement *Prop* for the TPCP. Intuitively, the property states:

" If, and only if, at least one page votes *NO*, all pages will *undo* the transaction. "

More technically, the process $PC(i)$ allows for i votes (and thus *Prop* for N votes, where $N = 2$) and - if the control flow has not left the process before - an amount of N subsequent events *complete*. As soon as one vote has the value *NO*, *PC* switches to a process *PU*. $PU(i)$ also allows for i votes, independent of the parameter value, but always terminates with *undo*s. Thus, as soon as one event *vote.NO* occurs, the final events of *PROP* will be *undo*.

Subsequently to the definition of the property, we define an assertion, stating that the individual specification *SPEC* refines the property *PROP*. Here, we additionally need to consider the respective decomposition we are dealing with: in the example, we evaluate the decomposition with respect to the cut $\{inform\}$. Thus, *SPEC* needs to be accordingly defined.

Figure 7.11 illustrates the compilation framework. First, the export is preprocessed, mainly to adapt CSP-OZ-DC specific syntax according to the one for CSP-OZ. Subsequently, the CSP_M-code for the specification, including the one for the verification property, is generated.

Recall the different options for exporting and verifying a specification against a certain requirement, as displayed in Figure 7.8. In order to apply *non-compositional* model checking, the first option can be selected, yielding a direct verification of $Prop \sqsubseteq_T S$. In this case, S is not decomposed at all. In any of the following options, the specification is decomposed according to the selected cut. Here, the second option again initiates direct model checking, this time to prove or contradict $Prop \sqsubseteq_T (S_1 \| S_2)$, whereas the last option solely exports the resulting specification to CSP_M.

For the remaining options, compositional verification based on the learning-based approaches from [CGP03] and [BGP03] can be carried out. As part of his bachelor's thesis, Wonisch [Won08] implemented the approach by using the CSP model checker `FDR2` as the teacher. The tool is called *CSPLChecker* [Won] and supports an assumption

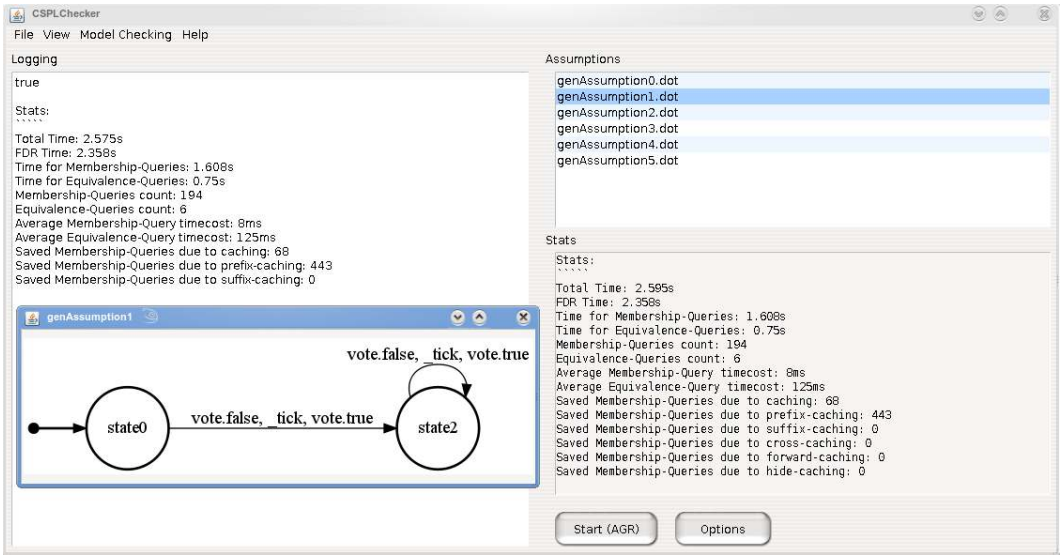


Figure 7.12: Screenshot of the CSPLChecker

generation, according to the learning frameworks for both assume-guarantee proof rules, **(B-AGR)** and **(P-AGR)** (see Section 3.3). Various optimisations as, for instance, different caching strategies, are implemented. During and after model checking, several statistics, such as the amount of membership queries or equivalence queries, can be displayed. For direct model checking with `FDR2`, the CSPLChecker is likewise called, forwarding the `FDR2` output and showing several statistics.

The screenshot of the CSLChecker from Figure 7.12 shows the verification result for the decomposition of the TPCP. The decomposition is carried out with respect to the cut $\{inform\}$, model checking refers to the property from Figure 7.11. At run-time, six intermediate assumptions are generated, the second one is displayed on the bottom left hand side. Overall, model checking takes approximately two seconds. The tool can freely be downloaded [Won], a more detailed description can be found in [Won08].

7.2.4 Counterexample Analysis

Independent of a direct call of `FDR2` or a compositional verification, model checking a specification against a requirement possibly results in a counterexample. Such an error trace comprises a sequence of events, constituting a violation of the respective verification property. As part of the CSPLChecker output, this trace is displayed within the Syspect console. However, a purely textual representation of a counterexample is difficult to analyse. In particular, recovering the counterexample within the model can become tedious.

In order to guide the user to the specific behaviour of the model which violates the verification property, Micus [Mic10] developed an additional extension to Syspect, the *countertrace* plug-in. By evaluating the textual representation of a counterexample and linking it back to the specification's state machines, the error trace is visualised within

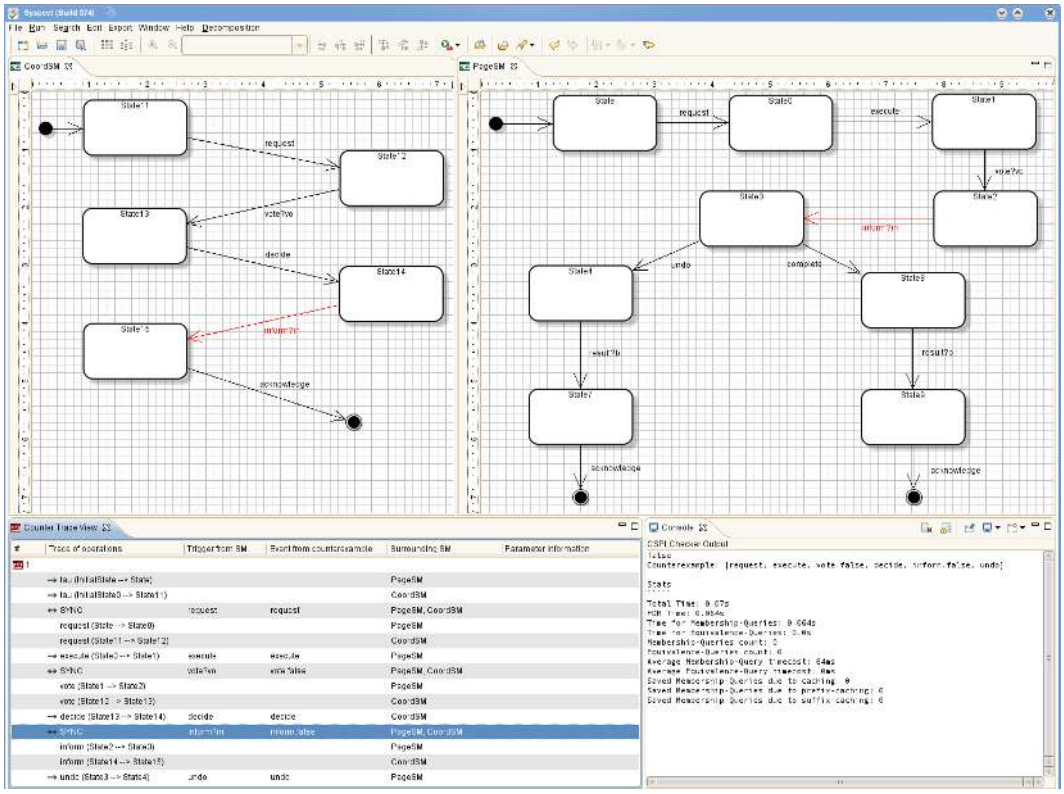


Figure 7.13: Screenshot of the counterexample visualisation

the Syspect model.

Consider a modification of the verification property from Figure 7.10: if we replace `pages = card(N)` by `pages = card(N)+1`, for $N = 1$, a verification will result in the following error trace:[1]

$$tr = \langle request, execute, vote.false, decide, inform.false, undo \rangle.$$

This is due to the fact that *Prop* requires the execution of two *votes* before the first *undo*, which is clearly impossible for the TPCP with one instance of *Page*.

tr comprises events, solely executed by one class, along with synchronised events between the classes *Coord* and *Page*. A visualisation of *tr* thus requires its events to be distributed over both state machines.

Figure 7.13 shows a screenshot of the visualisation of the error trace *tr* within both state machines. Here, the synchronised operation *inform* is selected, yielding the corresponding state machine triggers to be highlighted in red. Along with this, a visualisation of the complete error trace is possible.

Up to now, only the CSP part of the specification is analysed. In case there is more than one visualisation of the error trace, all of them are displayed. More details on the countertrace plug-in can be found in [Mic10].

[1] Recall, that *Votes* and *Trans* are mapped onto $\mathbb{B}$.

7.2.5 Overall Workflow

After introducing the several context-specific extensions of Syspect, we concludingly assemble and summarise them. Figure 7.14 shows the decomposition framework by using an UML *activity diagram* [Obj05].

Given a specification S and a property P, a user can either choose to apply direct model checking or compositional verification. In the first case, S is exported to LaTeX-mark-up (Syspect *export* plug-in, Section 7.1.4) and both, S and P are translated into a CSP_M-script (CSP_M-export, Section 7.2.3). Here, P is simply forwarded, as it is already specified in the input language of `FDR2`. In the latter case, S is decomposed into some $S_1 \parallel S_2$ - either by using the manual choice of a cut (Section 7.2.1) or the mass validation framework (Section 7.2.2), both realised within the Syspect *decomposition* plug-in. The property P possibly needs to be adapted to some property P', according to the respective decomposition: P' needs to comply to the specification in terms of transmission parameters and address parameters. Along with that, a modified assertion now refers to the decomposed system. Again, a compilation of $S_1 \parallel S_2$ and P' can be carried out, resulting in corresponding CSP_M-code.

Next, the actual model checking takes place. Direct, `FDR2`-based, model checking with respect to S or $S_1 \parallel S_2$ is generally possible. Compositional verification using the *CSPLChecker* (Section 7.2.3) requires the system to be composed of two components. This is clearly the case for $S_1 \parallel S_2$ and, if S itself is already assembled of two parts, for S as well. In any case, if the model checking yields an error trace, the counterexample is visualised within the Syspect model (Syspect *countertrace* plug-in, Section 7.2.4).

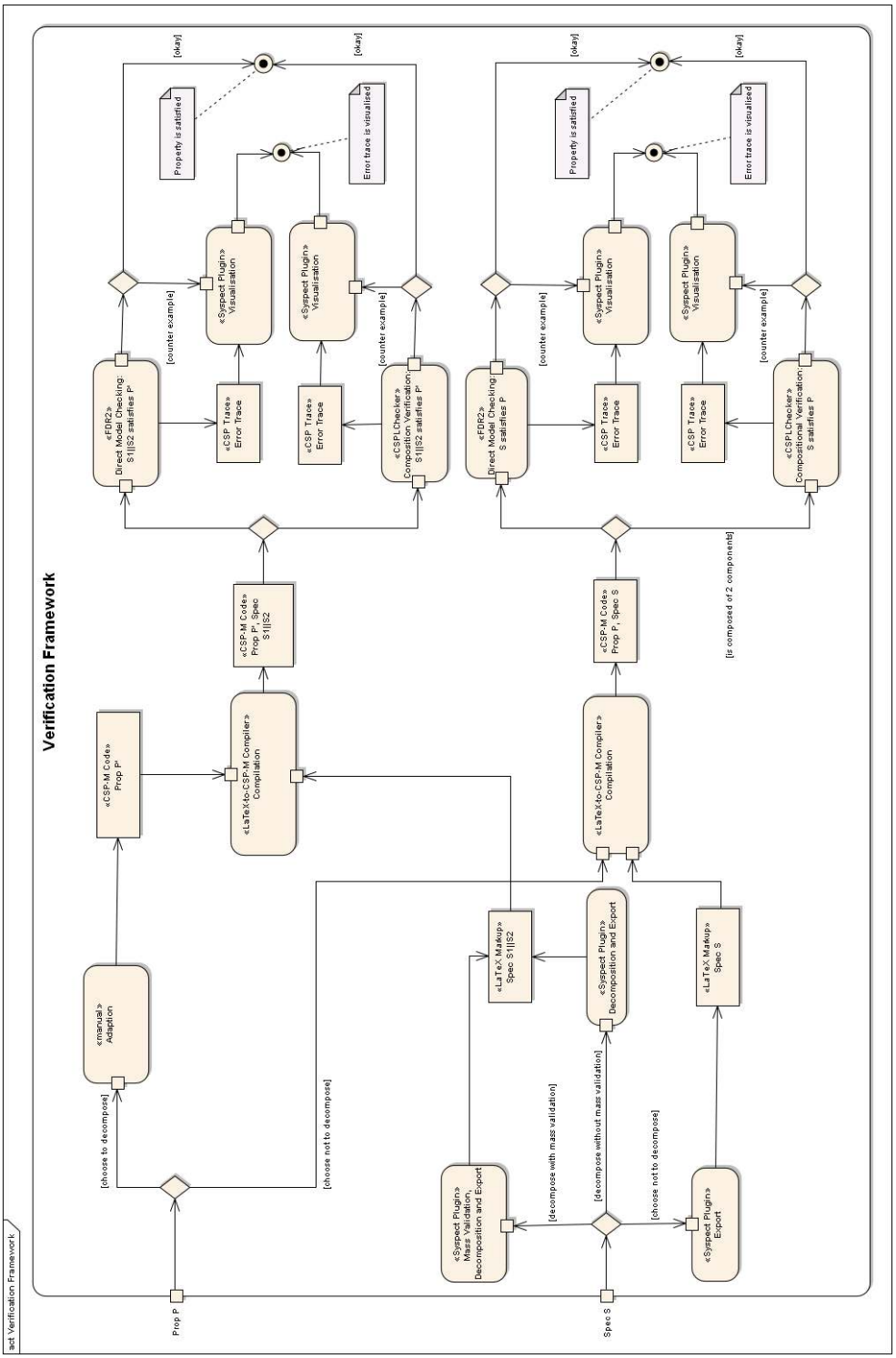


Figure 7.14: Verification framework

7.3 Experimental Results

Within this thesis, we specified several case studies, serving as an illustration of the main concepts, definitions and algorithms. The present section provides the experimental results for three specifications: the candy machine from Section 2.2, the Two Phase Commit Protocol from Section 6.4 and the number swapper from Section 4.4. In order to evaluate our approach, the examples have been specified within Syspect, decomposed and exported, after which the run-times during model checking were investigated. For all three case studies, we stepwise enlarged the size of the specification, namely, by increasing the maximal value for $\mathbb{Z}$ and, for the Two Phase Commit Protocol, the amount of participating pages. This allows us to estimate how the approach scales with an increasing size of the model.

We start this section with an overview on the technical conditions for our evaluation. Afterwards, we separately analyse the candy machine, the TPCP and the number swapper, and we draw some first conclusions. Finally, we discuss the evaluation results.

7.3.1 Overview

In order to accomplish an experimental evaluation of our approach, we analysed our case studies on a Dell PC, equipped with an Intel Core 2 Duo CPU, 4 GB RAM and openSUSE Linux 11.1. Besides that, we used Syspect version 1.4.0 with an integration of the various extensions, as described in Section 7.2. For model checking, we employed `FDR` in version 2.83.

All case studies have been modelled within Syspect. The tool is available from its public subversion directory [Cor], the various extensions and the case studies along with the corresponding exports are freely accessible from [Res].

We provide some more background information on the conducted experimental studies.

- Up to now, the implementation of our approach within Syspect does not allow for a decomposition with respect to a *general* cut. Thus, for the case study of the number swapper, we manually decomposed the model, before proceeding with model checking. For the remaining case studies, we evaluated those sets of decompositions, which correspond to the set of optimal and reasonable single cuts, as given in Chapter 6.

- According to the two different proof rules, **(B-AGR)** and **(P-AGR)**, we used two learning strategies, which will from now on be called *basic reasoning* (**BR**) and *parallel reasoning* (**PR**). In general, an assertion must be formed as $\mathit{Prop} \sqsubseteq_T (S_1 \parallel S_2)$, where S_1 denotes the first component of the decomposed system and S_2 the second.

- Some manual modifications of the export were necessary to achieve a fair comparison between the model checking results for the different systems. For instance, parameters were ordered such that the original ones are denoted first, followed by address parameters and transmission parameters.

- Instead of comparing the sizes of the generated state spaces during model checking, we choose to compare verification run-times along with the amount of equivalence queries and membership queries during learning. This is owed to the model checker `FDR2`, not allowing for the computation of the size and visualisation of the overall state space, generated during model checking. In fact, it is possible to display the state space of the *final* generated transition system. However, in order to compare the amount of states constructed and visited during model checking, the transition systems of the *intermediate* processes would have to be considered as well. In our context, comparing run-times along with the amount of the different L^*-queries, is a satisfactory way of an evaluation.
- In general, we specified verification properties, which turn out to be *valid* for the respective specification. Model checking is carried out, until either the system ran out of memory or the verification result is *true*.

Besides direct model checking of the original system and compositional, learning-based verification of the decomposition, we also evaluated direct model checking with respect to our generated decompositions. Therefore, our evaluation will investigate run-times for three different systems:

Original System: Given a specification S and a requirement *Prop*, we consider direct model checking of $\mathit{Prop} \sqsubseteq_T S$.

Decomposed System, no AGR: For a valid decomposition of S into $S_1 \parallel S_2$, we investigate direct model checking of $\mathit{Prop} \sqsubseteq_T (S_1 \parallel S_2)$.

Decomposed System, AGR based on Learning: For a valid decomposition of S into $S_1 \parallel S_2$, model checking of $\mathit{Prop} \sqsubseteq_T (S_1 \parallel S_2)$ with respect to the proof rules **(B-AGR)** and **(P-AGR)** is examined.

Section 7.3.5 will comment on the most likely reasons for the verification results. Next, we present the experimental results for the three investigated examples in detail.

7.3.2 Verification Results for the Candy Machine

Our experimental evaluation starts with the specification of a candy machine, as defined in Section 2.2. In Chapter 6, we already filtered all valid (single) cuts according to the criteria *unreasonable* (Definition 6.2.1) and *dominated* (Definition 6.2.3). The remaining two cuts are

- $\mathbf{C}_1 := \{\mathit{switch}\}$ and
- $\mathbf{C}_2 := \{\mathit{abort}, \mathit{order}, \mathit{select}, \mathit{switch}\}$.

Therefore, we will investigate three different systems: the undissected candy machine specification and two decompositions, according to the single cuts $\mathbf{C}_1$ and $\mathbf{C}_2$. Direct model checking is conducted for all three systems. In addition, for both decomposed systems, we consider basic reasoning and parallel reasoning.

```
PROP          = Paying(0)

Paying(i)     = (if (i+2 <= MAX) then P(i) else Collecting(i))

P(i)          = [] j:Coins @ (pay.j -> Paying(i+j))

Collecting(i) = (if i >= 0 then D(i) else STOP)

D(i)          = C(i) [] Terminate(i)

C(i)          = deliver.CHOC   -> Collecting(i-1)
              [] deliver.COOKIE -> Collecting(i-2)
              [] deliver.CRISPS -> Collecting(i-3)

Terminate(i)  = term.i -> SKIP

SPEC = (S_1
       [ {|abort,pay,payout,switch|} ||
         {|deliver,order,select,switch,term|} ]
        S_2)
       \ {|payout,abort,switch,select,order|}

assert PROP [T= SPEC
```

Figure 7.15: Correctness requirement for the candy machine in terms of CSP_M

The verification property, which we consider, is the one defined in Figure 2.6. Rephrased in terms of CSP_M, the property is denoted in Figure 7.15. Here, we additionally need to to disallow the usage of integer values greater than *MAX* and smaller than 0, which is specified within the property. The definition also contains an assertion for the decomposed system with respect to the cut $\{switch\}$.

In the following, we scale the size of the evaluated model by increasing the maximal integer value **MaxInf** within the CSP_M-code. Precisely, **MaxInf** equal to n means that $\mathbb{Z}$ is mapped to $\{-n,\ldots,n\}$ whereas $\mathbb{N}$ is mapped to $\{0,\ldots n\}$. The value for **MaxInf** determines a corresponding value for the constant *Max* (see Section 2.2.1).

We denote the run-times in seconds and, in case of learning, the membership queries and equivalence queries. The amount of equivalence queries is identical to the number of generated intermediate assumptions during learning.

The symbol `(*)` indicates that the memory limit was exceeded during model checking, causing `FDR2` to cancel the verification process with the message `std::bad_alloc`. In addition, `(-)` denotes that the respective verification was not conducted, as model checking for the same system already failed for a smaller value of **MaxInf**. Finally, `n/a` denotes that compositional verification was not applicable, as the original system is not composed of two components.

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	<1	n/a	n/a	n/a	n/a	n/a	n/a
{*switch*}	<1	<1	1	8	1	8	6
{*abort*, *order*, *select*, *switch*}	<1	<1	1	20	5	16	2000

(a) Results for **MaxInf** $= 1$

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	17	n/a	n/a	n/a	n/a	n/a	n/a
{*switch*}	<1	2	3	25	7	18	448
{*abort*, *order*, *select*, *switch*}	53	62	2	188	1916	92	156K

(b) Results for **MaxInf** $= 2$

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	(⋆)	n/a	n/a	n/a	n/a	n/a	n/a
{*switch*}	12	107	5	88	162	23	944
{*abort*, *order*, *select*, *switch*}	(⋆)	(⋆)	(−)	(−)	(⋆)	(−)	(−)

(c) Results for **MaxInf** $= 3$

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	(−)	n/a	n/a	n/a	n/a	n/a	n/a
{*switch*}	183	(⋆)	(−)	(−)	3044	25	1527
{*abort*, *order*, *select*, *switch*}	(−)	(−)	(−)	(−)	(−)	(−)	(−)

(d) Results for **MaxInf** $= 4$

Table 7.1: Experimental results for the candy machine

The improvement from Section 4.3.7, allowing for a neglect of specific initial data dependences, was not yet implemented in Syspect. Therefore, the cut {*switch*} is not indicated as a valid cut. In order to cope with this problem, we removed the initial predicate *items* $= \langle\,\rangle$ from the model and manually re-added it within the CSP_M-code.

Finally, we give the experimental evaluation for the candy machine specification. Table 7.1 displays the results for **MaxInf** $= 1$ to **MaxInf** $= 4$. Most importantly, in case that the machine did not exceed its memory limit, we denote the amount of seconds until the verification terminated with the result *true*. **DC** indicates direct model checking, and, as already mentioned, **BR** and **PR** indicate basic reasoning and parallel reasoning. The number of equivalence queries and membership queries are given in the columns marked with EQ and MQ, respectively. For **MaxInf** $= 5$, the two remaining evaluations for the cut {*switch*} lead to an out-of-memory exception.

It turns out that direct model checking of the original system can only be carried out for **MaxInf** $\in \{1, 2\}$. The same applies for the cut $\mathbf{C}_2 = \{abort, order, select, switch\}$, independent of monolithic or compositional verification. For this specific cut, run-times are even worse compared to model checking of the undissected model. The best results are achieved for the cut $\mathbf{C}_1 = \{switch\}$. Quite surprisingly, direct model checking outperforms the compositional one.

Summing up, due to the decomposition of the model, we can verify the investigated property on larger systems. Even though learning-based reasoning already outperforms monolithic verification of the original system, the best results are achieved for *direct* model checking of the decomposed systems according to one specific cut. This particularly shows that effective model checking for a decomposed system not automatically requires compositional, assume-guarantee-based strategies. We will further elaborate on these particular results in Section 7.3.5.

7.3.3 Verification Results for the Two Phase Commit Protocol

The next case study under investigation is the Two Phase Commit Protocol, specified in Section 6.4. Again, we only consider the set of optimal and reasonable cuts, as given in Table 6.6. In order to refer to the different cuts, we occasionally use the according numbers from the respective table. We verify the system against the property, given in Figure 7.10.

According to Section 6.4, there are nine optimal and reasonable cuts. As already argued, a heuristics-based approach solely points the direction, but it does not automatically determine *the* (set of) qualified cut(s). Instead of comparing all nine cuts, we filter the set by further analysing its elements:

- The cuts numbered as 24 and 39 result in two symmetric decompositions, which only differ in the two different operation names *undo* and *complete*. Therefore, we select one of these cuts for the evaluation, namely the one numbered as 39, that is, $\{inform, undo\}$.
- The sole reason why the cut $\{complete, inform, result, undo\}$ does not dominate the cuts numbered as 16 and 17, is the value for the heuristic **even distribution**. However, as the first cut is a subset of the latter two cuts, the difference appears simply due to a shift of node(s) into the cut. Clearly, this does not improve the decomposition, and we solely consider the first of these three cuts.
- The same argument applies in case we compare the cuts numbered as 23 and 19 with $\{inform, undo\}$.

The remaining four (single) cuts will be evaluated. These are

- $\mathbf{C}_1 := \{inform\}$,
- $\mathbf{C}_2 := \{vote, decide\}$,
- $\mathbf{C}_3 := \{inform, undo\}$ and

- $\mathbf{C}_4 := \{complete, inform, result, undo\}$.

Along with them, we will also consider the *dominated* cut $\mathbf{C}_5 := \{vote\}$: even though this cut seems to be a sensible one, our heuristics reject it. In order to draw some further conclusions on the plausibility of the heuristics, we exemplify an evaluation of a dominated cut on $\mathbf{C}_5$.

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	<1	4	9	464	5	33	973
$\{inform\}$	<1	2	6	194	16	28	4050
$\{vote, decide\}$	<1	2	5	230	3	16	464
$\{inform, undo\}$	<1	76	11	5011	29	28	6611
$\{complete, inform, result, undo\}$	<1	308	16	18K	2	5	416
$\{vote\}$	<1	4	3	226	7	13	976

(a) Results for two pages

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	<1	19	14	1071	25	52	3248
$\{inform\}$	7	10	8	567	192	52	38K
$\{vote, decide\}$	<1	11	7	933	9	23	1875
$\{inform, undo\}$	7	1459	21	38K	567	61	97K
$\{complete, inform, result, undo\}$	5	8449	23	148K	8	6	1403
$\{vote\}$	1	43	4	1218	40	19	4265

(b) Results for three pages

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	17	319	18	1839	6807	67	46K
$\{inform\}$	3657	42	10	1251	1970	83	205K
$\{vote, decide\}$	47	52	9	2672	74	29	5040
$\{inform, undo\}$	3142	$(\star)$	$(-)$	$(-)$	$(\star)$	$(-)$	$(-)$
$\{complete, inform, result, undo\}$	1796	$(\star)$	$(-)$	$(-)$	781	7	3614
$\{vote\}$	57	336	6	4721	161	20	10K

(c) Results for four pages

Table 7.2: Experimental Results for the TPCP, first part

Tables 7.2 and 7.3 show the evaluation results for the TPCP, comprising two to seven pages. The table is correspondingly configured to the one for the candy machine, and it uses the same symbol to indicate an out-of-memory failure during model checking. As the specification itself is composed of two components *Coord* and *Pages*, we can apply compositional verification for the original system as well.

The evaluation yields the following results: first of all, direct verification for the undissected system and for the decompositions according to $\mathbf{C}_2$ and $\mathbf{C}_5$ can be carried out for an amount of five pages, before the memory limit exceeded. For the remaining decompositions, direct verification is only possible for an amount of four pages.

Compositional verification results in comparatively worse run-times for two and three pages. However, the larger the model, the more effective becomes compositional reasoning and, in particular, basic reasoning. The best results are achieved for the decomposition according to $\mathbf{C}_2$, that is, $\{vote, decide\}$. Here, model checking can be carried out for up to seven pages, before the memory limit exceeds. Regarding the dominated cut $\{vote\}$, it is outmatched by basic reasoning with respect to the cuts $\{inform\}$ and $\{vote, decide\}$. Thus, even though one might intuitively assume $\{vote\}$ to declare a better decomposition than $\{vote, decide\}$, the heuristics prove this conjecture wrong. A more detailed discussion will be part of Section 7.3.5.

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	926	8696	23	2926	(⋆)	(−)	(−)
$\{inform\}$	(⋆)	571	12	2342	(⋆)	(−)	(−)
$\{vote, decide\}$	3584	241	11	6174	11K	35	11K
$\{inform, undo\}$	(⋆)	(−)	(−)	(−)	(−)	(−)	(−)
$\{complete, inform, result, undo\}$	(⋆)	(−)	(−)	(−)	(⋆)	(−)	(−)
$\{vote\}$	4220	1877	7	11K	10K	17	20K

(a) Results for five pages

Cut	**DC**	**BR**			**PR**		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	(⋆)	(⋆)	(−)	(−)	(−)	(−)	(−)
$\{inform\}$	(−)	(⋆)	(−)	(−)	(−)	(−)	(−)
$\{vote, decide\}$ (6 pages)	(⋆)	1738	13	12K	(⋆)	(−)	(−)
$\{vote, decide\}$ (7 pages)	(−)	5846	15	22K	(−)	(−)	(−)
$\{inform, undo\}$	(−)	(−)	(−)	(−)	(−)	(−)	(−)
$\{complete, inform, result, undo\}$	(−)	(−)	(−)	(−)	(−)	(−)	(−)
$\{vote\}$	(⋆)	(⋆)	(−)	(−)	(⋆)	(−)	(−)

(b) Results for six and seven pages

Table 7.3: Experimental Results for the TPCP, second part

Summing up, contrary to the evaluation results for the candy machine, direct verification of the decomposed system results in higher run-times than basic reasoning. Moreover, basic reasoning performs significantly better than parallel reasoning. The example shows that assume-guarantee-based compositional verification can indeed lead to a significant speed-up during model checking.

```
PROP = [] j: Nat @ (input.j -> result.1 -> P(j))
P(j) = [] k: Nat @ (input.k -> result.j -> P(k))

SPEC = (S_1
       [ {|input,storeB,result|} ||
         {|storeB,moveA,moveB,result|} ]
        S_2)
       \ {|moveA,moveB,storeB|}

assert PROP [T= SPEC
```

Figure 7.16: Correctness requirement for the number swapper in terms of CSP_M

7.3.4 Verification Results for the Number Swapper

The final case study under investigation is the (extended version of the) number swapper from Section 4.4, defined in Figure 4.26. Here, due to the specific recursive structure of the CSP part, a decomposition with respect to a single cut is impossible. Moreover, based on the different data dependences, there is only one reasonable general cut, namely $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$, for $\mathbf{C}_1 = \{store\_b\}$ and $\mathbf{C}_2 = \{result\}$. Thus, we manually decomposed the specification according to $\mathbf{C}$, and we compared run-times for direct verification of the original system and the decomposed system with the ones for compositional verification of the decomposed system.

In order to carry out the model checking, we refer to the verification property, as given in Figure 4.29. Rephrased in terms of CSP_M with an additional assertion in regard of the decomposed system, the property is specified in Figure 7.16. It states:

" The parameter value, received by *input*, corresponds to the output value of *result* in the next iteration of the protocol. "

According to the experimental evaluation of the candy machine, we scale the specification by stepwise increasing the maximal integer value **MaxInf**. The individual results are given in Tables 7.4 and 7.5, respectively.

Compositional reasoning, particularly with respect to the proof rule **(B-AGR)**, results in drastically worse run-times than non-compositional one. The comparison between direct verification of the original system and the decomposed one mainly yields a draw. Thus, for this case study, decomposing the specification does not yield an advantage over model checking of the undissected system.

7.3.5 Discussion

In this section, we evaluated three case studies, and we compared direct model checking with the compositional one with diverse results:

Cut	DC	BR			PR		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	<1	n/a	n/a	n/a	n/a	n/a	n/a
{*store_b*}, {*result*}	<1	26	22	2740	3	32	591

(a) Results for **MaxInf** = 1

Cut	DC	BR			PR		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	<1	n/a	n/a	n/a	n/a	n/a	n/a
{*store_b*}, {*result*}	<1	249	40	15K	13	57	2243

(b) Results for **MaxInf** = 2

Cut	DC	BR			PR		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	<1	n/a	n/a	n/a	n/a	n/a	n/a
{*store_b*}, {*result*}	<1	1888	66	56K	58	97	5883

(c) Results for **MaxInf** = 3

Cut	DC	BR			PR		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	1	n/a	n/a	n/a	n/a	n/a	n/a
{*store_b*}, {*result*}	1	11K	98	155K	230	147	12K

(d) Results for **MaxInf** = 4

Cut	DC	BR			PR		
	sec	sec	EQ	MQ	sec	EQ	MQ
None	3	n/a	n/a	n/a	n/a	n/a	n/a
{*store_b*}, {*result*}	3	62K	136	355K	779	207	25K

(e) Results for **MaxInf** = 5

Table 7.4: Experimental results for the (extended) number swapper, first part

1.) For the specification of a candy machine, direct model checking based on the cut {*switch*} outmatches learning-based verification along with direct verification of the original system.

2.) Regarding the Two Phase Commit Protocol, the learning-based method performs best, particularly for the decompositions according to the cuts {*vote*, *decide*} and {*inform*}.

3.) The evaluation of the final case study, the number swapper, showed that a decomposition of the system does not always improve the run-times during model checking in a significant way.

Cut	DC / PR **MaxInf** $= 6$	DC / PR **MaxInf** $= 7$	DC / PR **MaxInf** $= 8$	DC / PR **MaxInf** $= 9$
None	10	23	52	108
$\{store\_b\}, \{result\}$	10 / 2498	24 / 7209	52 / 19K	10 / (−)

(a) Results for **MaxInf** $\in \{6, 7, 8, 9\}$

Cut	DC **MaxInf** $= 10$	DC **MaxInf** $= 11$	DC **MaxInf** $= 12$
None	209	379	($\star$)
$\{store\_b\}, \{result\}$	207	372	($\star$)

(b) Results for **MaxInf** ≥ 10

Table 7.5: Experimental results for the (extended) number swapper, second part

A summary of the results is given in Table 7.6. They will lead us to some context-specific conjectures, which we will discuss next. In order to draw some conclusions and develop an intuition on when decomposing a system plus applying compositional verification might pay off, we investigate the specific model checker `FDR2` and the structure of the different case studies. Note that the following interpretations and considerations are mostly educated guesses and conjectures: neither can we precisely estimate the model checking procedure of `FDR2`, nor can we draw detailed and irrefutable conclusions from a heuristics-based technique.

Verification Technique	Case Study		
	Candy Machine	TPCP	Number Swapper
Direct, original system	-	-	+
Direct, decomposition	+	-	+
Compositional, decomposition	○	+	-

Table 7.6: Summary of the experimental results

General Conclusions

Based on the experimental results, we discuss some general observations. First, we experienced that the order of both components within the assertion is relevant for basic reasoning and – due to the nature of the symmetric proof rule – irrelevant for parallel reasoning. For basic reasoning, model checking of $Prop \sqsubseteq_T (S_1 \parallel S_2)$ generally performed better than the one of $Prop \sqsubseteq_T (S_2 \parallel S_1)$. The previous tables thus always refer to the case of $Prop \sqsubseteq_T (S_1 \parallel S_2)$.

Next, we observed that two particular criteria were most relevant for the measured

run-times of model checking. The first one is related to the additional address- and transmission parameters: parameters of high cardinality significantly increase the run-times. As the type of transmission parameters is arbitrary, decompositions without transmission parameters or, at least, transmission parameters of small type-cardinality should be favoured. As a second, closely related criterion, the amount of cut nodes highly influences the duration of model checking. For our case studies, we experienced that cuts with a size of more than two nodes generally lead to comparatively bad results. As both criteria determine the number of events, which have to be synchronised in the decomposition, both observations substantiate the claim from Section 6.1: the interface between both components needs to be small.

Another observation is related to the specific model checker we used for the evaluation: the behaviour of `FDR2` in the context of the learning based approach is generally non-deterministic, and it is nearly impossible to draw conclusions on how the amount of membership queries and equivalence queries can be reduced [Won08]. For instance, a reordering of the specification's parameters changes the number of intermediate assumptions. However, there is no general rule which orderings should generally be favoured.

Regarding the comparison of parallel reasoning and basic reasoning, basic reasoning mostly outmatched the parallel one. This might be related to the specific case studies which we investigated: the candy machine and the Two Phase Commit Protocol can be seen as *sequential* systems without outer recursion, thus favouring the specific sequential structure of the rule **(B-AGR)**. Yet, for the case study of the number swapper, even though parallel reasoning performed better that the basic one, run-times were significantly higher compared to direct model checking. This raises doubts on the usefulness of *parallel* reasoning in general.

Finally, we want to substantiate the claim that not only the final transition graph, generated during model checking, is relevant for the number of explored states. We illustrate this by an example.

Example 7.3.1. *Let us consider the following simple CSP-OZ specification.*

Simple

`chan` a, b, c

`main` $\stackrel{c}{=} a \rightarrow b \rightarrow c \rightarrow$ `Skip`

$a\_guard, b\_guard, c\_guard : \mathbb{B}$

Init

$a\_guard, b\_guard, c\_guard = true$

`enable_a`

$a\_guard = true$

`enable_b`

$b\_guard = true$

`enable_c`

$c\_guard = true$

`effect_a`

$\Delta(a\_guard)$

$a\_guard' = false$

`effect_b`

$\Delta(b\_guard)$

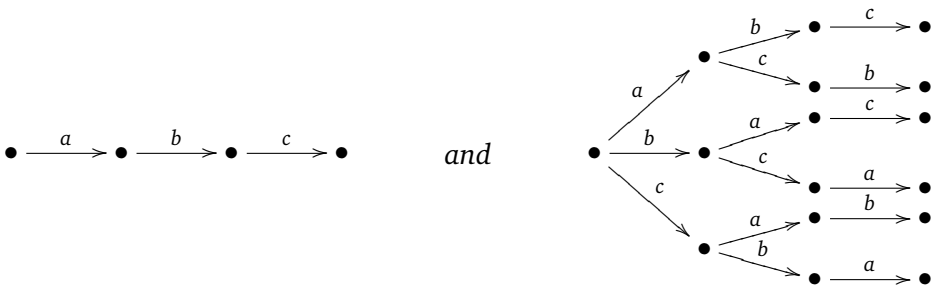
$b\_guard' = false$

`effect_c`

$\Delta(c\_guard)$

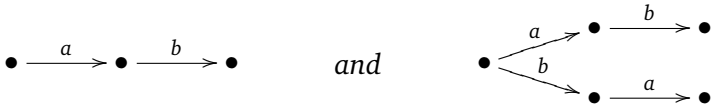
$c\_guard' = false$

The CSP process of the class solely allows for the trace $\langle a, b, c \rangle$. For the Object-Z part, any ordering of operations is possible, as long as each operation is only called once. Thus, in order to analyse the specification, the parallel composition of the two transition systems



needs to be computed (without denoting the state variables of the Object-Z part). Even though the transition system of the overall process is identical to the one for the CSP part, the much bigger transition system of the Object-Z part must be computed as well before the parallel composition can be carried out.

Now assume we decompose the specification based on the valid single cut $\mathbf{C} = \{b\}$. In this case, the transition system for the first component is a parallel composition of



For the transition system of the second component, the event b is simply replaced by the event c. The final transition systems for the original specification and the decomposed one are identical and according to the one of the original CSP part. However, the size of the intermediate system differs: for the original system, there are 19 states and 18 transitions, the decomposed systems needs to cope with only 16 states and 12 transitions. Thus, (direct) model checking with respect to the original model needs to explore more states than the compositional one.

The example particularly shows that direct model checking of a decomposed system can indeed outmatch direct model checking of the original specification. This can be the case if the decomposition results in smaller intermediate transition systems due to, for instance, a significant reduction of interleaving or an effective distribution of the set of state variables.

Evaluation Analysis: Candy Machine

We quote some further case-study-specific observations, and we start by analysing the results for the candy machine. The state space of the specification particularly comprises two sequences *paid* and *items*. Even though `FDR2` supports the specification of sequences, generating the set over all possible sequences of finite length n for some specific data type with cardinality k results in k^n elements. This is further substantiated by the fact

that FDR2 mainly applies *explicit* model checking techniques and generally needs to deal with the full state space of a system.

Consider the decomposition of the specification with respect to the cut $\{switch\}$, as given in Section 4.3.6. It results in a distribution of the state variables *paid* and *items* over both components – *paid* is assigned to $CandyMachine_1$, and *items* is assigned to $CandyMachine_2$. Thus, the individual state spaces of the Object-Z parts of the specification are significantly smaller than the state space of the original system. Hence, it is most likely that model checking with respect to $\mathbf{C}_1 = \{switch\}$ performs comparatively better than the one for the original system.

Regarding the cut $\mathbf{C}_2 = \{abort, order, select, switch\}$, the corresponding decomposition requires a transmission parameter of type $\mathbb{seq}\, Candies$. In addition, the number of cut nodes is equal to four. Thus, model checking with respect to $\mathbf{C}_2$ leads to comparatively poor results.

Yet, the question remains why direct model checking outperforms the compositional one. As already mentioned, the performance of learning-based compositional reasoning depends on the number of intermediate assumptions. According to [Won08] and due to the *black-box* character of FDR2, it seems quite difficult to pre-estimate this number.

Evaluation Analysis: Two Phase Commit Protocol

In [dRHH+01], the motivation for introducing and specifying the Two Phase Commit Protocol is its particular structure, allowing for an appliance of the *Communication-Closed-Layers law* (CCL) [EF82]. Our way of decomposing a specification is one particular way of adopting the CCL, which leads to a transformation of a specification with a *distributed* or *concurrent* structure – such as the parallel composition of several processes – to a *sequential* or *layered* structure, consisting of several phases.

The evaluation of this specific case study shows that the structure of the TPCP facilitates an application of compositional techniques. In particular, the protocol itself consists of two phases, which are nearly independent.

Quite surprisingly, the cut yielding the minimal run-times during model checking is $\mathbf{C} = \{vote, decide\}$. Figures 7.17 and 7.18 show the decomposition of the specification according to $\mathbf{C}$. In order to address specific instances of $Page_1$ and $Page_2$, we adopt CSP-OZ's concept of *constant* parameters ([Fis00]).

This specific cut reflects the loose connection between the commit-request-phase and the commit-phase: the corresponding decomposition only requires one transmission parameter of type $Trans = \{COMMIT, ABORT\}$ for the operation *decide*. This parameter represents the final decision to either commit or abort a transaction and thus, the point of intersection between both phases. As *decide* only occurs once within the specification, this parameter of type cardinality 2 is only used once as well. Contrary, the cut $\{vote\}$ requires a transmission parameter of type $\mathbb{P}\, Votes$ for the operation *vote*. Thus, the cardinality of the type of this operation is larger than the one for the parameter of *decide*. Moreover and more importantly, *vote* occurs once in each instance of *Page* and *N* times within *Coord*. This requires the transmission parameter to be added to all *N* occurrences within *Coord*. Therefore, even though the cut $\mathbf{C}_5 = \{vote\}$ only comprises one operation schema, model

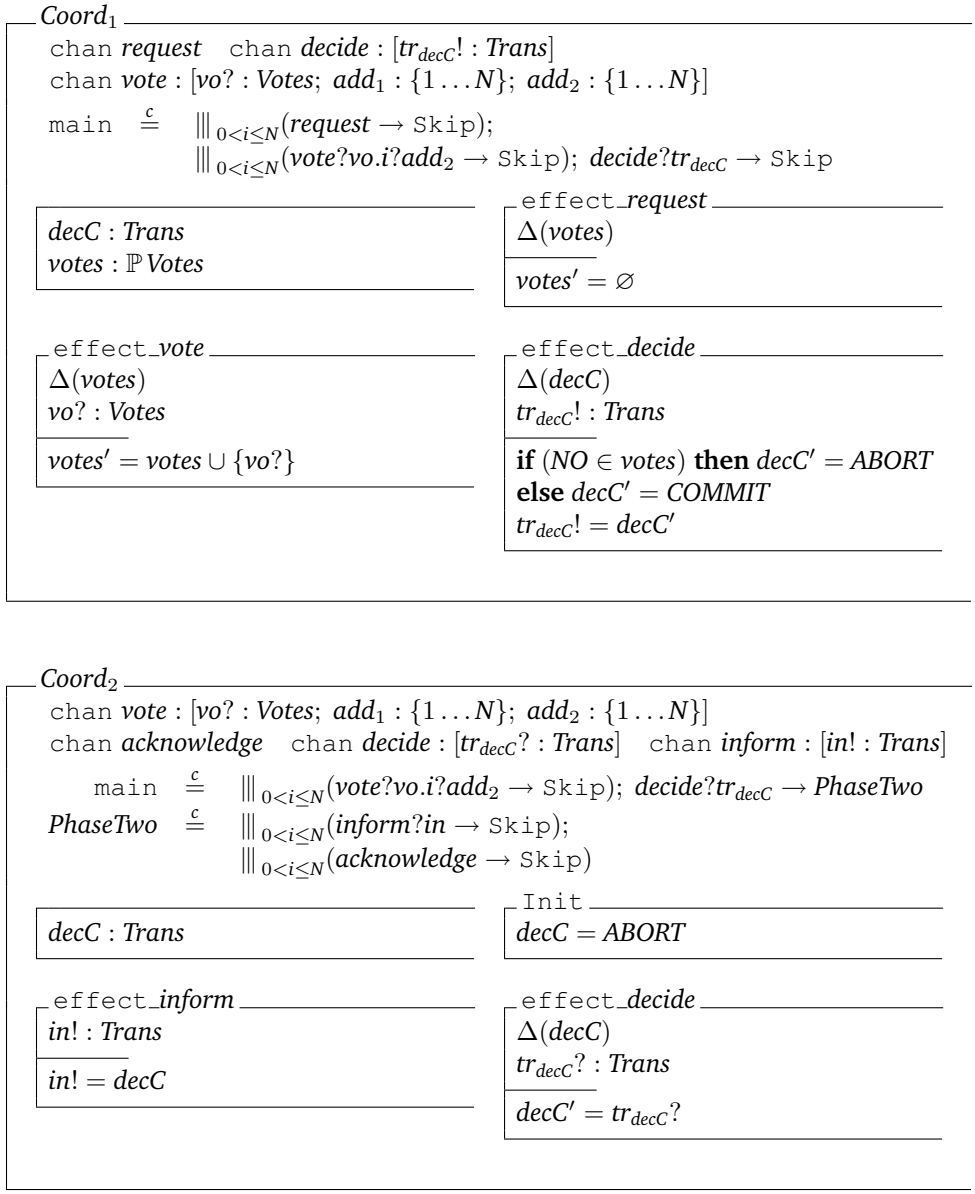


Figure 7.17: Decomposition of the TPCP: *Coord* specification

checking needs to cope with comparatively more events than the one of the decomposition according to $\{vote, decide\}$. Figure 7.19 illustrates the predominance of $\{vote, decide\}$.

Similarly, for the decomposition with respect to the cut $\mathbf{C}_1 = \{inform\}$, one additional transmission parameter of type *Trans* is required, and *inform* occurs multiple times within the specification. The larger the amount of pages, the more occurrences of *vote* and *inform* and thus, the more cut events for $\mathbf{C}_1$ and $\mathbf{C}_5$. This is reflected in the evaluation results: the larger the model, the better performs $\{vote, decide\}$ in comparison to the

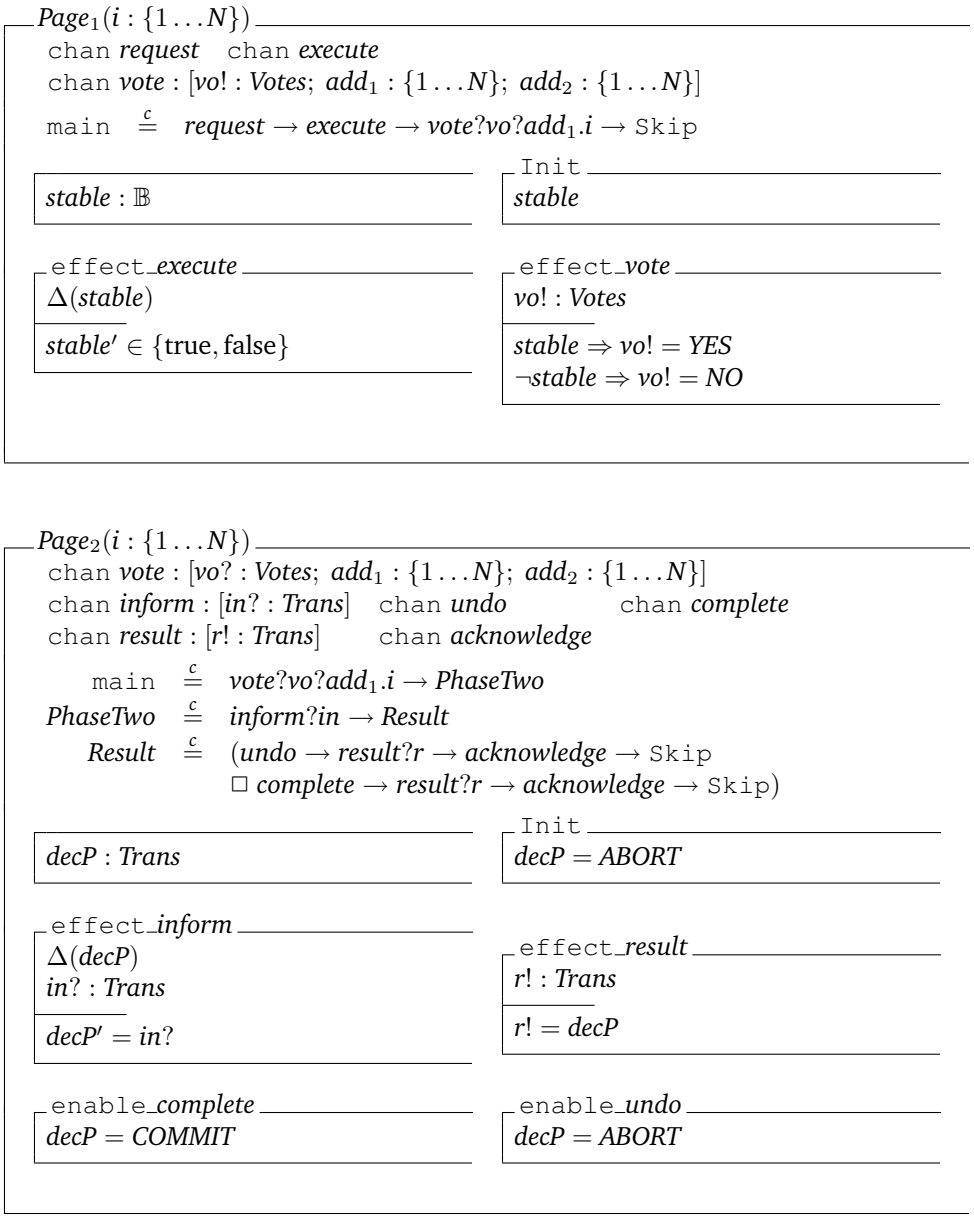


Figure 7.18: Decomposition of the TPCP: *Page* specification

other cuts.

Still, within the mass validation framework, the cut $\{inform\}$ receives the minimal value, if we set an equal weight for all heuristics. This is due to the comparatively smaller values for the heuristics **cut size** and **even distribution**. The example substantiates to offer a possibility of *scaling* the different heuristics: a higher weight for **few transmission** will cause $\{vote, decide\}$ to pass $\{inform\}$ in terms of the overall value.

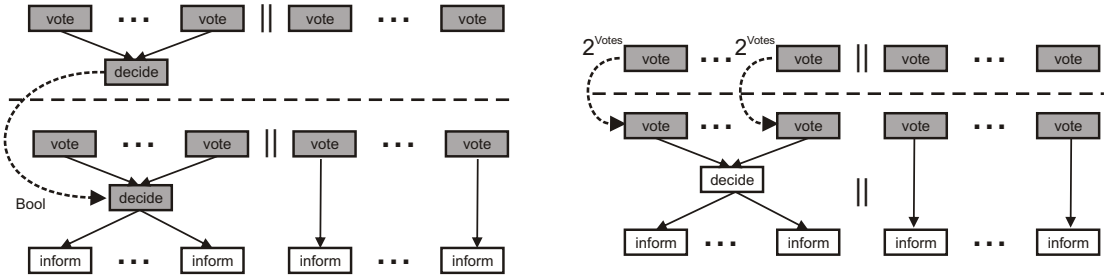


Figure 7.19: Justification for predominance of cut $\{vote, decide\}$

Evaluation Analysis: Number Swapper

The results for the number swapper showed that the decomposition of a system does not generally lead to a significant improvement in regard of model checking run-times. In this specific case, the structure of the system does not allow for a decomposition beneficial for compositional verification: the CSP part of the specification itself comprises five events and only allows for a general cut. Thus, one of the components inevitably comprises four events. Moreover, *store_b* requires an additional transmission parameter, increasing the size of the interface between both components. In conclusion, learning-based model checking results in poor run-times.

Still, by decomposing the number swapper, the structure of the specification is mainly maintained. There is no advantage in the application of direct model checking of the original system over direct model checking of the decomposed system.

Summary

The evaluation results for the different case studies of this thesis highly differ. Summing up, we can conclude that there is no universal best strategy which type of verification one should choose. Still, we identified some *rules-of-thumb* for when to apply which strategy: in general, applying the decomposition technique can be promising, if the underlying system can be distributed in a reasonable way. This can either mean a split-up of the CSP part into two phases without a large intersection between both parts (as, for instance, the Two Phase Commit Protocol and the cut $\{vote, decide\}$) or a reasonable distribution of its set of state variables (as, for instance, the candy machine and the cut $\{switch\}$).

The decomposition approach will not always be beneficial. In particular, if the system is tightly coupled, a decomposition might not significantly reduce run-times during (compositional) verification. However, the technique is generally applicable and for none of the case studies did direct model checking of the original system perform best. Even though one of our case studies represents a tightly coupled system, direct model checking of its decomposition results in run-times, which are comparable to the ones for the model checking of the undissected system.

8 Conclusion

Contents

8.1 Summary **217**
8.2 Future Work **219**

The present chapter concludes this thesis with a summary. Subsequently, we discuss some topics for future research.

8.1 Summary

Within this thesis, we introduced a decomposition technique for software models, specified in an integrated formalism. The primary motivation for this approach arose from the major challenge of automated software verification: the state explosion problem. In order to allow model checking to scale to complex systems, appropriate measures need to be taken. Compositional verification is one possible way of dealing with the state explosion. The technique follows a "divide and conquer" approach: instead of verifying the system as a whole, the components of the system are independently verified. An appropriate combination of the verification results yields the correctness of the system. Compositional reasoning avoids the state explosion problem to a certain extent, if the overall state space of the components is comparatively smaller than the one of the original system.

After a short introduction to the topic, Chapter 2 provided background information on the modelling and the analysis of software models. First, we surveyed the field of *integrated formal methods*. Next, we presented the syntax and the semantics of the underlying integrated formalism of this thesis, *CSP-OZ*, and we exemplified it by means of a case study. Furthermore, a *dependence analysis* for CSP-OZ specifications was introduced. Here, we defined the specification's *dependence graph*, reflecting the control flow and the data flow of a software model. The dependence graph provides the basis for a further analysis and, eventually, a decomposition of a specification.

The second background chapter, Chapter 3, focussed on strategies for the automated verification of a software model and, in particular, *compositional verification*. We provided an overview on related and complementary techniques to cope with the state explosion problem. The *assume-guarantee paradigm* of compositional reasoning was introduced, along with two inference proof rules. Both proof rules are applied within our implementation framework, and they use the L^* *algorithm* for an automatic detection of intermediate assumptions during model checking.

The first core chapter of this thesis, Chapter 4, described the actual *decomposition of a CSP-OZ specification*. The general idea for the approach is the definition of a *cut* of the

specification's dependence graph. In order to ensure the validity of the decomposition, that is, the semantic equivalence of the decomposed system and the original system, a cut needs to comply with four *correctness criteria*. We separated the general case from the specific case of a *single* cut, mainly to allow a certain class of systems to be decomposed in a more effective way. Subsequently, we defined a model's *decomposition* with respect to a valid cut. In order to guarantee the equivalence between the original and the decomposed system, additional modifications of the resulting components had to be introduced. Mainly, these modifications required the introduction of additional parameters in order to restore the specification's original control flow and data flow.

Chapter 5 showed *correctness* of the decomposition technique in terms of the trace equivalence of the original and the generated model. The proof employed the operational semantics of CSP-OZ. We compositionally showed the correctness of both, the decomposition of the CSP part and the one of the Object-Z part. For the CSP part, we showed *bisimilarity* of the considered CSP processes, taking into account the additional *address parameters* within the CSP part of the decomposition. For the correctness proof of the Object-Z part, we showed trace equivalence by explicitly constructing the respective transition paths and by using *transmission parameters*. Finally, both individual results were used to deduce the overall correctness of the decomposition, additionally requiring several CSP-related laws for renaming and for a redistribution of CSP processes.

As the validity of a decomposition does not automatically yield a system for which model checking can effectively be carried out, Chapter 6 introduced several context-specific *heuristics* to measure the quality of a decomposition. A classification of all valid cuts is carried out in two steps: first, all *unreasonable* and *dominated* cuts are sorted out. Second, the remaining cuts can be scaled, according to the heuristics.

Finally, Chapter 7 evaluated the approach on three case studies. As the underlying platform, we chose the UML-based modelling environment Syspect and the CSP model checker `FDR2`. In order to integrate the decomposition technique into the existing framework, several extensions to Syspect were carried out. We compared run-times for direct and compositional model checking for the original systems and the decompositions.

In summary, within this thesis, we developed a technique to effectively apply compositional verification for software models, specified in an integrated formalism. We mainly answered the following two questions:

1. How can we determine the set of all *valid* decompositions of a specification?

2. How can these decompositions be *classified and measured* regarding their effectiveness for compositional model checking?

We further implemented the approach by integrating it into an existing tool for the modelling and the analysis of software specifications. Based on the results obtained, we observed that the decomposition technique can lead to a considerable speed-up of both, compositional verification and direct verification.

8.2 Future Work

The present work opened several perspectives and ideas for future research, which we will discuss next. Particularly, we detail some further extensions and some possibilities on how to combine the approach of this thesis with complementary techniques.

Target Area of Application: The decomposition technique of this thesis has been carried out with respect to the integrated formal method CSP-OZ. Yet, the approach presents the major benefit of being independent from a specific formalism: the theory of Chapter 4 used the *(program) dependence graph (DG)* of a specification in order to decompose a software model. DGs are a commonly-used and language-independent way of representing a software system [HR92]. Thus, the technique can be transformed to fit to any language with an underlying dependence graph representation for its models. The general idea behind the decomposition of the model in terms of restoring the original control flow and the data flow can be used accordingly. The correctness criteria need to be adapted, according to the context specific semantic model and the equivalence requirements.

Semantic Model: As the semantics of CSP-OZ are given in terms of CSP alone [Fis00], our correctness proof referred to the semantic domain of CSP. Within the general context of learning-based model checking of *safety* properties, we were interested in analysing the observable behaviour of a specification. This allowed us to restrict ourselves to the semantic model of *traces*, that is, the sequences of communication events: the trace semantics is sufficient for showing the observable equivalence of two systems [CGP03, Weh00]. In order to analyse liveness properties as, for instance, deadlock or livelock freedom, the decomposition technique could as well be extended to the more discriminating *failures-divergences* model of CSP. This semantic model additionally takes the refusals of events and infinite sequences of internal actions into account. The extension would require a modification of the correctness proof and, possibly, some additional correctness criteria in order to ensure the failure-divergence equivalence of the original system and its decomposition.

Evaluation with ProB: An evaluation of the decomposition technique was carried out by using the CSP model checker `FDR2`. The choice was justified by several aspects, which were given in Section 7.2.3. In order to realise a more profound analysis, as a meaningful measure, the approach could be evaluated for a second model checker. Recently, ProB [Leu], an animator and model checker for the B-Method [Abr96], was extended to support CSP_M as the input language [LF08]. Thus, an implementation of the learning framework from Chapter 3 for ProB and a further comparison of evaluation results for ProB and `FDR2` could be advisable.

Weakening of Correctness Criteria: One correctness criterion for a valid fragmentation of the DG states that data dependences must not circumvent the set of cut operations. The criterion was justified by the fact that the influence from one specification part on the other one needs to be preserved. Our decomposition approach ensured this

by using transmission parameters. A possible weakening of the cut definition could be a neglect of this criterion. In this case, transmission parameters would have to be used for state variables which are modified before the cut as well. In general, these considerations result in a trade-off between, on the one hand, the growing amount of valid cuts and, on the other hand, the more complex evaluation of the set of all valid cuts. Yet, we observed that a large set of additional parameter values considerably slows down model checking, raising doubts on whether this strategy is a way to success.

Recursive Learning: Previous works [GGP07, PGB+08] extended the learning framework to systems with an arbitrary number of components. Here, the specification is stepwise decomposed, using a *recursive* application of the learning algorithm. Wonisch already integrated the method into the CSPLChecker [WW09]. His extension allows to recursively apply both assume-guarantee-based proof rules with respect to a specific split-ratio and systems which are parallel composed of n components. The theory of our thesis implicitly supports the recursive decomposition of a system, as the two resulting specification components are CSP-OZ specifications as well. In order to integrate this extension into Syspect, a *re-import* of the decomposed model and, in particular, a computation of the respective dependence graphs is necessary.

Arbitrary Amount of Cut Sets: According to Definition 4.2.8, a general cut refers to *two* cut sets, and it yields a fragmentation of the DG into two parts. Clearly, this approach might be extended to an arbitrary amount of lines of intersection, yielding a decomposition of a model into a corresponding number of components. However, within this thesis, we restricted ourselves to two cut sets: as previously explained, the approach supports a recursive decomposition, already allowing for a decomposition into an arbitrary amount of components without the need to generalise and further complicate Definition 4.2.8.

Combination with other Techniques: Another motivation for the re-import of a CSP-OZ class specification into Syspect is given by the possibility to combine two techniques, the slicing approach from [Brü08] and the decomposition method of this thesis: if the verification requirement is *at hand*, an obvious strategy is to first slice the original model with respect to the given requirement, re-import the slice and decompose it according to our technique. Moreover, the presented technique is generally compatible to other approaches to the state explosion problem.

Decomposition Implementation: Chapter 7 presented the implementation of the decomposition approach within the UML-based modelling tool Syspect. Here, several future extensions are possible, mainly for closing the gaps between the theory and the implementation and for facilitating the tool handling.

- *Implementation of General Cut Theory:* The implementation of the decomposition technique within Syspect is currently restricted to the special case of a *single* cut. In order to allow the tool to support the decomposition of arbitrary

specifications, an extension of the according plug-in is required. Here, the main aspect to deal with is to allow the definition of two separate cut sets, with each of them complying to the implemented theory for one cut set.

- *Implementation of Decomposition Improvement:* In Section 4.3.7, we discussed an improvement of the decomposition approach in terms of reducing the set of initial data dependences. This optimisation is not yet implemented within Syspect and thus, several valid cuts are currently rejected. An implementation of this improvement, dependent on the respective specification, would lead to larger set of valid cuts.
- *Modelling of Verification Properties:* Currently, a manual specification of verification properties in terms of CSP_M is required. As a facilitation, a modelling of the system requirements as a transition system is imaginable. Such an editor could be similar to the existing Syspect state machine editor.
- *Extension of Counter Trace Plug-In:* Section 7.2.4 introduced the countertrace plug-in, a Syspect extension for visualising counterexamples. Currently, the analysis only considers the CSP part of a specification. Thus, the set of detected error traces is possibly too large. An additional analysis of the Object-Z part would yield an exact counterexample analysis.
- *Elimination of System Classes:* As a more technical aspect, the LaTeX-export of a Syspect specification, comprising more than one class, requires the definition of an additional class for describing the overall system composition. These classes may be replaced by a simple CSP process, as they comprise an empty Object-Z part. In order to speed-up model checking, the definition of the overall system within Syspect should be given by a CSP process instead of a CSP-OZ class.

Glossary of Symbols

CSP-OZ (Section 2.2)

$S.I$	the interface definition of a specification S
$S.Events$	the global set of events of a specification S
$S.OZ$	the Object-Z part of a CSP-OZ specification S
$S.\texttt{main}$	the CSP part of a CSP-OZ specification S

Object-Z part (Section 2.2)

`State`	the state schema of OZ
`Init`	the initial state schema of OZ
Op	the set of operation schemas of OZ
`enable_`op	the precondition of the operation schema op
`effect_`op	the effect of the the operation schema op
$op.delta$	the delta list of the operation schema op
$op.dec$	the parameter declaration part of the operation schema op
$op.pred$	the predicate part of the operation schema op
$In(op)$	the set of possible values for the input parameters of op
$Simple(op)$	the set of possible values for the simple parameters of op
$Out(op)$	the set of possible values for the output parameters of op
$ref(op)$	the set of referenced variables of the operation schema op
$mod(op)$	the set of modified variables of the operation schema op
$all(op)$	the union of the sets of referenced and modified variables of op
$s \upharpoonright V$	the state s, projected onto the set of state variables V
M^{OZ}	the labelled transition system of OZ
$Traces(OZ)$	the set of traces of the Object-Z part
$\pi[i]$	the i-th state of $\pi \in Traces(OZ)$
$\pi.i$	the i-th event of $\pi \in Traces(OZ)$
$traces(OZ)$	the set of traces of OZ, projected onto events
$traces(OZ) \rhd Op$	the set of traces of OZ, projected onto operation names

CSP part (Section 2.2)

`Skip`	termination
`Stop`	deadlock
$a \rightarrow P$	a then P
$P_1 \Box P_2$	P_1 external choice P_2
$P_1 \sqcap P_2$	P_1 internal choice P_2
$P_1 \parallel_A P_2$	P_1 parallel on A P_2
$P_1 \ {}_{A_1}\!\parallel_{A_2} P_2$	P_1 parallel on A_1, A_2 P_2
$P_1 \mid\mid\mid P_2$	P_1 interleave P_2
$P_1 \setminus A$	P hide A
$P[\![\mathtt{R}]\!]$	P renamed by `R` (relational renaming)
`main`	the initial CSP process of a CSP-OZ specification
L^{CSP}	the set of all CSP terms
M^{CSP}	the labelled transition system of `main`
$tr \upharpoonright E$	the restriction of the trace tr on events in E
$traces(\mathtt{main}) \rhd Op$	the set of traces of `main`, projected onto operation names
$tr.i$	the i-th event of the trace tr
$initials(P)$	the initial events of the process P
$foot(tr)$	the last event of the trace tr
$P\vert_E$	the projection of the process P on events in E
$P \sqsubseteq_T Q$	Q is a trace refinement of P
$P =_T Q$	P and Q are trace equivalent
αP	the alphabet of the process P
$\{\mid C \mid\}$	the extension set for the set of channels C
c_p	a partial event for the channel c

Dependence Graphs (Section 2.3)

$\mathtt{DG}_S = (N, \longrightarrow_{\mathtt{DG}})$	the dependence graph of a specification S
$\mathtt{CFG}_S = (N, \longrightarrow)$	the control flow graph of a specification S
$\mathtt{DDG}_S = (\mathtt{op}(N), \overset{}{\dashrightarrow})$	the data dependence graph of a specification S
$\mathtt{op}(N)$	the set of operation nodes of a dependence graph
$\mathtt{cf}(N)$	the set of CSP operator nodes of a dependence graph
$\longrightarrow$	a control dependence
$\overset{\mathtt{dd}}{\dashrightarrow}$	a direct data dependence
$\overset{\mathtt{dd}}{\dashrightarrow}_{(v)}$	a direct data dependence by reason of v
$\overset{\mathtt{idd}}{\dashrightarrow}$	an initial data dependence
$\overset{\mathtt{sd}}{\leftarrow\!\dashrightarrow}$	a synchronisation dependence
$\overset{\mathtt{sdd}}{\dashrightarrow}$	a synchronisation data dependence
$\overset{\mathtt{ifdd}}{\dashrightarrow}$	an interference data dependence
$\overset{\mathtt{ifdd}}{\dashrightarrow}_{(v)}$	an interference data dependence by reason of v
$\mathtt{path}_{\mathtt{DG}\,/\,\mathtt{CFG}}$	the paths of the DG / CFG
$\mathtt{path}_{\mathtt{DG}\,/\,\mathtt{CFG}}(\mathsf{n}, \mathsf{n'})$	the paths of the DG / CFG from n to n'
succ(n)	the sole successor of the node n
succ_one(n)	the first successor of the node n
succ_two(n)	the second successor of the node n

Compositional Reasoning (Sections 3.2 and 3.3)

(B-AGR)	the basic assume-guarantee proof rule
(P-AGR)	the parallel assume-guarantee proof rule
(C-AGR)	the circular assume-guarantee proof rule
$\mathcal{L}(A)$	the language of the assumption A, given as a DFA
$\mathcal{L}(A)^C$	the complement of the language $\mathcal{L}(A)$
Σ^*	the set of *finite* words over Σ
Σ^ω	the set of *infinite* words over Σ

Cut of a Dependence Graph (Section 4.2)

$\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$	a (general) cut of a dependence graph
$\mathbf{C} = (\mathbf{C}_1, \varnothing)$	a single cut of a dependence graph
$\mathbf{Ph}_i$	the i-th phase of a fragmentation of a dependence graph
$N_1 \xrightarrow{\texttt{to}} N_2$	the interval of nodes from N_1 to N_2 of a dependence graph
disjointness	the first correctness constraint on a valid cut
no crossing	the second correctness constraint on a valid cut
no reaching back	the third correctness constraint on a valid cut
all-or-none	the fourth correctness constraint on a valid cut

Decomposition of a Specification (Section 4.3)

S_1	the first component of a decomposition of S
S_2	the second component of a decomposition of S
CV	the set of cut variables
$op.orig$	the set of original parameters of the operation op
$op.add$	the set of address parameters of the operation op
$op.tr\_in$	the set of transmission parameters of the operation op, decorated with "?"
$op.tr\_out$	the set of transmission parameters of the operation op, decorated with "!"
Op_i	the set of local operation schemas of the component S_i
$Op_{\mathbf{C}}$	the set of cut operation schemas
Op'	the union $Op_1 \cup Op_2 \cup Op_{\mathbf{C}}$
E_{S_i}	the set of events of the component S_i
$E_{S'}$	the union $E_{S_1} \cup E_{S_2}$
$\mathtt{R}^{\mathtt{C}}$	the (relational) event renaming for a decomposition of S
$\mathtt{R}'$	the inverse renaming relation
$InitClos(x)$	the initial closure of the state variable x

Correctness Proof (Chapter 5)

`noev`	a special event to denote stuttering
$\overline{CV}$	the shared state variables, excluding cut variables
$s \oplus t$	the state t overrides the state s

Decomposition Heuristics (Section 6.1)

h_{CS}	the heuristic for minimising the cut size
h_{ED}	the heuristic for minimising the size difference
h_{FT}	the heuristic for minimising the transmission
h_{FA}	the heuristic for minimising the addressing

Predicate Logic

$Free(p)$	the set of free variables occurring in the predicate p
$p[x/a]$	the predicate p with all free occurrences of x replaced with a
$Atoms(p)$	the set of atomic sub-predicates of the predicate p
$vars(p)$	the set of variables occurring in the predicate p

Miscellaneous

Id_X	the identity on X, that is, the set $\{(x,x) \mid x \in X\}$

Bibliography

[Abr96] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr06] J.-R. Abrial. Formal methods in industry: achievements, problems, future. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 761–768, New York, NY, USA, 2006. ACM.

[ACHH92] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer, 1992.

[AČMN05] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices*, 40(1):98–109, 2005.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AH06] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, XXI, 2006.

[All70] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 221–234, London, UK, 2001. Springer-Verlag.

[BBK+04] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. Interactive verification of UML state machines. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2004.

[BCC98] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. *Lecture Notes in Computer Science*, 1536:81–102, 1998.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems. Part of European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[BGH+05] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The FUJABA real-time tool suite: Model-driven development of safety-critical, real-time systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, pages 670–671. ACM Press, 2005.

[BGL+00] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.

[BGP03] H. Barringer, D. Giannakopoulou, and C. S. Pasareanu. Proof rules for automated compositional verification through learning. In *International Workshop on Specification and Verification of Component Based Systems, Finland*, 2003.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJR99] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bow09] J. Bowen. Formal Methods Wiki. `http://formalmethods.wikia.com`, 2009.

[Brü08] I. Brückner. *Slicing Integrated Formal Specifications for Verification*. PhD thesis, Universität Paderborn, 2008.

[Bry86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, 2003.

[But09] M. Butler. Decomposition structures for event-B. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, volume 5423 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2009.

[CAC06] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 97–108, New York, NY, USA, 2006. ACM.

[CBRZ01] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles*, pages 238–252. ACM Press, New York, 1977.

[CCST05] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 534–547. Springer, 2005.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGK97] S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of liveness properties using compositional reachability analysis. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 227–243. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, 1997.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[CGP03] J. Cobleigh, D. Giannakopoulou, and C. Păsăreanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[CMP94] E. Chang, Z. Manna, and A. Pnueli. Compositional verification of real-time systems. In *Logic in Computer Science (LICS '94)*, pages 458–467, Los Alamitos, Ca., USA, 1994. IEEE Computer Society Press.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Conference record of third annual ACM symposium on theory of Computing*, pages 151–158, Shaker Heights, Oh., 1971. ACM.

[Cor] Correct System Design Group. Syspect subversion repository. `https://homer.informatik.uni-oldenburg.de/svn/syspect`.

[CS07] S. Chaki and O. Strichman. Optimized L*-based assume-guarantee reasoning. In *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2007.

[CS08] S. Chaki and O. Strichman. Three optimizations for assume-guarantee reasoning with L^*. *Formal Methods in System Design*, 32(3):267–284, 2008.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Colloque sur la Programmation*. Springer-Verlag, Berlin, DE, 1974.

[Die05] H. Dierks. Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. Habilitation thesis, 2005.

[Dij72] E. W. Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, London, 1972.

[DNS08] J. Derrick, S. North, and A. J. H. Simons. Z2SAL - building a model checker for Z. In *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2008.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[dRHH+01] W. de Roever, U. Hanneman, J. Hooiman, Y. Lakhneche, M. Poel, J. Zwiers, and F. de Boer. *Concurrency Verification*. Cambridge University Press, Cambridge, UK, 2001.

[DW04] O. L. De Weck. Multiobjective optimization : history and promise. 2004.

[DW07] J. Derrick and H. Wehrheim. On using data abstractions for model checking refinements. *Acta Inf*, 44(1):41–71, 2007.

[DWQQ01] W. Dong, J. Wang, X. Qi, and Z. Qi. Model checking UML statecharts. In *APSEC*, pages 363–370. IEEE Computer Society, 2001.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming, ICALP'80*, volume 85 of *LNCS*, pages 169–181. Springer-Verlag, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong, 1980.

[EDK89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional Model Checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 353–361, Washington D.C., 1989. IEEE Computer Society Press.

[EF82] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, 1982.

[Ehr00] M. Ehrgott. *Multicriteria optimization*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 2000.

[ESB+09] S. Edelkamp, V. Schuppan, D. Bošnački, A. Wijs, A. Fehnker, and H. Aljazzar. Survey on directed model checking. pages 65–89, 2009.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis99] C. Fischer. Printing CSP-OZ documents with latex; documentation for csp-oz.sty. Technical report, University of Oldenburg, 1999.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[FMS01] A. Farias, A. Mota, and A. Sampaio. Java translator from CSP-Z to CSPM notation. `http://www.di.ufpe.br/~acf/translator/CSPZtoCSPM.html`, 2001.

[For05] Formal Systems (Europe) Ltd. Failure divergence refinement: FDR2 user manual. `http://www.fsel.com/documentation/fdr2/fdr2manual.pdf`, 2005.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, 1987.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9(2), 1978.

[FW99] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In *IFM*, pages 315–334, 1999.

[Gal04] D. Galin. *Software quality assurance*. Pearson Education Limited, Harlow, England, 2004.

[GGP07] M. Gheorghiu, D. Giannakopoulou, and C. S. Păsăreanu. Refining interface alphabets for compositional verification. In *TACAS*, pages 292–307, 2007.

[GL91] O. Grumberg and D. E. Long. Model checking and modular verification. In *CONCUR '91: Proceedings of the 2nd International Conference on Concurrency Theory*, pages 250–265, London, UK, 1991. Springer-Verlag.

[GMF07] A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. In *CAV*, pages 420–432, 2007.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, Berlin, 1996.

[GP08] D. Giannakopoulou and C. S. Păsăreanu. Special issue on learning techniques for compositional reasoning. *Form. Methods Syst. Des.*, 32(3):173–174, 2008.

[GP09] D. Giannakopoulou and C. S. Pasareanu. Interface generation and compositional verification in Java PathFinder. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5503 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2009.

[GPB02] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, pages 3–12, Washington, DC, USA, 2002. IEEE Computer Society.

[GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370, London, UK, 2002. Springer-Verlag.

[Gri97] A. Griffiths. Modular reasoning in Object-Z. In *Proceedings: 4th Asia-Pacific Software Engineering and International Computer Science Conference*, pages 140–149. IEEE Computer Society Press, 1997.

[Gri98] A. Griffiths. *A formal semantics to support modular reasoning in Object-Z*. PhD thesis, University of Queensland, 1998.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Her09] K. Herbold. Konzeption & Implementierung eines Dekompositions-Werkzeugs für kompositionelle Verifikation. Diploma's thesis, Universität Paderborn, 2009.

[HJ98] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, London, 1998.

[Hoa78] C. Hoare. Communicating sequential processes. *CACM*, 21:666–677, 1978.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoe06] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, 2006.

[Hol03] G. J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.

[HR92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM.

[ISO89] ISO - International Standards Organization. Information processing systems – open systems interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour. Technical report, 1989. ISO/IEC 8807.

[ISO00] ISO - International Standards Organization. Information technology – Z formal specification notation – syntax, type system and semantics. Technical report, 2000. ISO/IEC 13568.

[ISO01] ISO - International Standards Organization. Information technology – enhancements to LOTOS (E-LOTOS). Technical report, 2001. ISO/IEC 15437.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Jon83] C. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[KK99] G. Karypis and V. Kumar. Multilevel k -way hypergraph partitioning. In *DAC*, pages 343–348, 1999.

[KP88] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in LNCS, pages 489–507, Noordwijkerhout, The Netherlands, 1988. Springer-Verlag.

[KS01] G. Kassel and G. Smith. Model checking Object-Z classes: Some experiments with FDR. In *APSEC*, pages 445–452. IEEE Computer Society, 2001.

[Kur94] R. P. Kurshan. *Computer-aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton Univ. Press, 1994.

[Leu] M. Leuschel. ProB homepage. `http://www.stups.uni-duesseldorf.de/ProB`.

[LF08] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new fdr-compliant validation tool. In *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*, volume 5256 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 2008.

[LMC01] M. Leuschel, T. Massart, and A. Currie. How to make FDR spin - LTL model checking of CSP by refinement. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 99–118. Springer, 2001.

[LZ74] Liskov and Zilles. Programming with abstract data types. *Sigplan Notices*, 9, 1974.

[Mai03] P. Maier. Compositional circular assume-guarantee rules cannot be sound and complete. In *FoSSaCS*, pages 343–357, 2003.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Softw. Eng.*, 7(4):417–426, 1981.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[MDA] OMG model driven architecture. `http://www.omg.org/mda`.

[MG07] N. Moffat and M. Goldsmith. Assumption-commitment support for CSP model checking. *Electron. Notes Theor. Comput. Sci.*, 185:121–137, 2007.

[Mic10] S. Micus. Rückführung und Visualisierung von Gegenbeispielen aus einem Model Checker. Bachelor's thesis, Universität Paderborn, 2010.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, 1999.

[Moo90] A. P. Moore. The specification and verified decomposition of system requirements using CSP. *IEEE Transactions on Software Engineering*, 16:932–948, 1990.

[MORW08] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a formal method into a software engineering process with UML and Java. *Formal Apsects of Computing*, 20:161–204, 2008.

[MS98] A. Mota and A. Sampaio. Model-checking CSP-Z. In *FASE*, pages 205–220, 1998.

[MS01] A. Mota and A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Sci. Comput. Program*, 40(1):59–96, 2001.

[MWW08] B. Metzler, H. Wehrheim, and D. Wonisch. Decomposition for compositional verification. In *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*, volume 5256 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2008.

[NA06] W. Nam and R. Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *ATVA*, pages 170–185, 2006.

[Nam07] W. Nam. *Synthesis and Compositional Verification Using Language Learning*. PhD thesis, University of Pennsylvania, 2007.

[NAS] NASA Ames Research Center. Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`.

[Obj05] Object Management Group. OMG unified modeling language 2.0. `http://www.omg.com/uml`, 2005.

[OD08] E.-R. Olderog and H. Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Scientific and Engineering Computation Series. Cambridge University Press, New York, NY, 2008.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[OW05] E.-R. Olderog and H. Wehrheim. Specification and (property) inheritance in CSP-OZ. *Science of Computer Programming*, 55(1-3):227–257, 2005.

[Par71] V. Pareto. *Manual of Political Economy*. Kelley, New York, 1971. Originally published 1927. Translated from the Italian by A. S. Schwier, edited by A. S. Schwier and A. N. Page.

[PBG05] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[PGB+08] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[Pie10] M. Piepmeyer. Effiziente Validierung und Bewertung von Modellzerlegungen. Diploma's thesis, Universität Paderborn, 2010.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, Berlin, New York, 1984.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. *Lecture Notes Comp. Sci.*, 194:15–32, 1985.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[Res] Research Group Specification and Modelling of Software Systems. Syspect extensions subversion repository. `https://svn-serv.cs.upb.de/syspect-plugins`.

[Ros98] W. A. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[RR95] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT FSE*, pages 41–52, 1995.

[RW94] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In *Proceedings of the Fifth International Conference on Concurrency Theory CONCUR'94, Uppsala (Sweden)*, pages 226–241, Berlin-Heidelberg-New York, 1994. Springer.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Sch02] P. Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic*, pages 393–436. King's College Publications, 2002.

[Sch05] S. Schneider. Non-blocking data refinement and traces-divergences semantics. Technical report, University of Surrey, 2005.

[Sch09] S. Schneider. Personal communication, 2009.

[SGT+03] W. Schäfer, H. Giese, M. Tichy, S. Burmester, and S. Flake. Towards the compositional verification of real-time UML designs. In *ESEC/SIGSOFT FSE*, pages 38–47. ACM, 2003.

[SLU89] K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An algorithmic procedure for checking safety properties of protocols. *IEEE Trans. Communications*, 37(9):940–948, 1989.

[Smi95] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[SNT85] Y. Sawaragi, H. Nakayama, and T. Tanino. *Theory of multi-objective optimization*. Academic Press, Inc., Orlando, FL, 1985.

[Spi92] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[ST02] S. Schneider and H. Treharne. Communicating B machines. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 416–435, London, UK, 2002. Springer-Verlag.

[ST04] S. Schneider and H. Treharne. Verifying controlled components. In *IFM*, pages 87–107, 2004.

[ST05] S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.

[Sta06] A. Stamer. Integration von CSP-OZ in die OO-Softwareentwicklung für die automatische Verifikation. Diploma's thesis, Universität Oldenburg, 2006.

[STE05] S. Schneider, H. Treharne, and N. Evans. Chunks: Component verification in CSP||B. In *IFM'2005*, pages 89–108, 2005.

[SW05] G. Smith and L. Wildman. Model checking Z specifications using SAL. In *ZB*, pages 85–103, 2005.

[SWC02] A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in Circus. In *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.

[Sys06] Syspect. Endbericht der Projektgruppe Syspect. Technical report, Carl von Ossietzky University of Oldenburg, 2006.

[TA97] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In *ICFEM*, pages 283–292, 1997.

[TJ02] J. J. P. Tsai and E. Y. T. Juan. Model and heuristic technique for efficient verification of component-based software systems. In *IEEE ICCI*, pages 59–68. IEEE Computer Society, 2002.

[TS99] H. Treharne and S. Schneider. Using a process algebra to control B operations. In *IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456, London, UK, 1999. Springer-Verlag.

[WC02] J. Woodcock and A. Cavalcanti. The semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[Weh00] H. Wehrheim. Data abstraction techniques in the validation of CSP-OZ specifications. *Formal Aspects of Computing*, 12, 2000.

[Wei81] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[Wik06] Wikipedia. Retrieved from `http://en.wikipedia.org/wiki/Wikipedia`, 2006.

[Won] D. Wonisch. CSPLChecker homepage. `http://www.cs.uni-paderborn.de/index.php?id=8967&L=1`.

[Won08] D. Wonisch. Automatisiertes kompositionelles Model Checking von CSP Spezifikationen. Bachelor's thesis, Universität Paderborn, 2008.

[WS03] K. Winter and G. Smith. Compositional verification for Object-Z. In *ZB*, pages 280–299, 2003.

[WW09] H. Wehrheim and D. Wonisch. Compositional CSP traces refinement checking. In *Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS 2008)*, volume 250, Issue 2, pages 135–151. Elsevier B.V., 2009.

[Xie96] M. Xie. *Handbook of Software Reliability Engineering*. McCraw Hill, New York, 1996.

[Zel74] M. Zeleny. *Linear Multiobjective Programming*, volume 95 of *Lecture Notes in Economics and Mathematical Systems*. Springer, Berlin/New York, 1974.

[ZH04] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.

List of Figures

1.1 Decomposition of a specification S into S_1 and S_2 4
1.2 Illustration of the overall approach of this thesis 4

2.1 Structure of a CSP-OZ specification 12
2.2 Structure of the Object-Z part of a CSP-OZ specification 12
2.3 Candy machine specification 14
2.4 Illustration of the CSP part of the candy machine specification 16
2.5 Simplified grammar of CSP 20
2.6 Correctness requirement for the candy machine specification 24
2.7 Translation of a CSP-OZ specification into a CSP process 25
2.8 Simple CSP-OZ class specification for swapping two numbers 28
2.9 Control flow graph (CFG) for the candy machine specification 31
2.10 Simple CSP-OZ class specification for a ticket machine 35
2.11 Data dependence graph (DDG) for the ticket machine specification 36
2.12 Extract of DDG for the candy machine specification 37
2.13 Dependence graph (DG) for the candy machine specification 39

3.1 Basic assume-guarantee proof rule **(B-AGR)** 44
3.2 Parallel assume-guarantee proof rule **(P-AGR)** 44
3.3 Circular assume-guarantee rule **(C-AGR)** 45
3.4 Illustration of the L^* algorithm 46
3.5 Illustration of the L^* based learning framework 47
3.6 Rule **(B-AGR)** rephrased in terms of CSP trace refinement 49
3.7 Rule **(P-AGR)** rephrased in terms of CSP trace refinement 49
3.8 CSP specification of a simple elevator system 50

4.1 Overview of the cut identification and the decomposition 57
4.2 Illustration of Definition 4.2.1 58
4.3 Fragmentation of the DG 60
4.4 Disallowed control flow edges based on disjointness 62
4.5 Motivation for the correctness criterion **no crossing** 62
4.6 Disallowed data dependences based on **no crossing** 63
4.7 Motivation for the correctness criterion **no reaching back** 64
4.8 Disallowed edges based on **no reaching back** 65
4.9 Fragmentation of the set of operation nodes in general case 67
4.10 Assignment of DG edges to the subgraphs 68
4.11 Fragmentation of the set of operation nodes in the special case 69
4.12 Cut of the dependence graph for the candy machine 71
4.13 Constituents of a CSP-OZ class specification 72

4.14 Correspondence between graph nodes and specification operations 73
4.15 Simple CSP-OZ specification for increasing two natural numbers 80
4.16 Intermediate decomposition of *Increaser* 80
4.17 Possible data dependences targeting the cut and originating from the cut . 81
4.18 Illustration of the transmission parameters 84
4.19 Decomposition of *Increaser*, modified according to Definition 4.3.10 85
4.20 Synchronisation of events for external choice 87
4.21 Addressing extension for CFG branching 89
4.22 Addressing extension for nested branching 90
4.23 Illustration of Theorem 4.3.16 96
4.24 Decomposition of the candy machine, first component 104
4.25 Decomposition of the candy machine, second component 105
4.26 CSP-OZ specification for swapping two numbers, extended 107
4.27 Decomposition of the number swapper, first component 108
4.28 Decomposition of the number swapper, second component 108
4.29 Correctness requirement for *Swapper* 109

5.1 Illustration of the steps of the correctness proof 112
5.2 Algorithm for the address extension: procedure ADDRESSMAIN 114
5.3 Algorithm for the address extension: procedure ADDRESSCUT 114
5.4 Algorithm for the address extension: procedure MODIFYCUT 115
5.5 Algorithm for the address extension: procedure ADD 116
5.6 Illustration of a violation of Lemma 5.2.1 120
5.7 Illustration of a violation of Lemma 5.2.2 120
5.8 Illustration of the CSP correctness proof of binary operators 121
5.9 Case differentiation for Lemma 5.2.3, parallel composition 131
5.10 Illustration of Definition 5.3.2 and Lemmas 5.3.3, 5.3.4 141
5.11 Illustration of Lemma 5.3.12 151
5.12 Illustration of Lemma 5.3.15 157

6.1 Illustration of the Two Phase Commit Protocol 176
6.2 Phase one of the Two Phase Commit Protocol 176
6.3 Phase two of the Two Phase Commit Protocol 177
6.4 Two Phase Commit Protocol: *Coord* specification 177
6.5 Two Phase Commit Protocol: *Page* specification 178

7.1 Syspect class diagram for the TPCP 185
7.2 Syspect property view for the operation *Page.inform* 185
7.3 Syspect state machine for the class *Page* of TPCP 186
7.4 Syspect state machine for the class *Coord* of TPCP 187
7.5 Syspect component diagram for the TPCP 188
7.6 Toolchain for the verification framework 189
7.7 Screenshot of a selected invalid cut 190
7.8 Screenshot of the decomposition options after selection of a valid cut . . . 191

7.9 Screenshot of the mass validation framework 193
7.10 Correctness requirement for the TPCP in terms of CSP_M 194
7.11 Compilation from LaTeX to CSP_M 195
7.12 Screenshot of the CSPLChecker 196
7.13 Screenshot of the counterexample visualisation 197
7.14 Verification framework 199
7.15 Correctness requirement for the candy machine in terms of CSP_M 202
7.16 Correctness requirement for the number swapper in terms of CSP_M 207
7.17 Decomposition of the TPCP: *Coord* specification 213
7.18 Decomposition of the TPCP: *Page* specification 214
7.19 Justification for predominance of cut $\{vote, decide\}$ 215

List of Tables

1.1 Contributions of this thesis 6

2.1 Comparison between the different semantics for CSP-OZ 27
2.2 Table of nodes of the control flow graph 30
2.3 Table of edges of the data dependence graph 33

4.1 Comparison between the general cut and the single cut 70
4.2 Comparison of two traces for *Increaser* and its components 82
4.3 Comparison of two traces of *Increaser* and its components after modification 85

6.1 Heuristic $h_{\mathbf{CS}}$: **cut size** 170
6.2 Heuristic $h_{\mathbf{ED}}$: **even distribution** 170
6.3 Heuristic $h_{\mathbf{FT}}$: **few transmission** 172
6.4 Heuristic $h_{\mathbf{FA}}$: **few addressing** 172
6.5 Set of valid cuts for the candy machine 175
6.6 Set of valid cuts for the TPCP 179

7.1 Experimental results for the candy machine 203
7.2 Experimental Results for the TPCP, first part 205
7.3 Experimental Results for the TPCP, second part 206
7.4 Experimental results for the (extended) number swapper, first part 208
7.5 Experimental results for the (extended) number swapper, second part . . . 209
7.6 Summary of the experimental results 209

Index

Symbols

Skip 20
Stop 20
$A_1 \| A_2$ 20
$\Box$ 20
$\sqcap$ 20
$|||$ 20
$\|_A$ 20
Run_A 21
$\fatsemi$ 20
$\mathrm{CSP_M}$ 193
L^* 45, 182
SPIN 182

A

abstract interpretation 42
address algorithm 114
    correctness 118
    termination 117
allowed synchronisation 92
assume-guarantee reasoning 3, 43
    basic proof rule 44
    circular proof rule 44
    parallel proof rule 44
    soundness of basic proof rule 50
    soundness of parallel proof rule . 51

B

black box checking 54
bounded model checking 43

C

CCS 10
CCS-Z 10
CFG . . . . . *see* control flow graph
Circus 11
Communicating Sequential Processes *see* CSP
Communication Closed Layers law . 212
compositional verification 3, 43
    learning 45
cone-of-influence reduction 42
control flow analysis 29
control flow graph 29
    fragmentation 59
        completeness 60
    labelling of nodes 32
    paths 30
        recursion-free 141
    phase 59
CSP 11, 20
    compositional verification 53
    failures-divergences model 22
    semantics 22
    set of CSP terms 21
    stable failures model 22
CSP||B 10
    compositional verification 53
CSP process
    alphabetised parallel 20
    channel 20
        extension set 21
    channel type 20
    choice 20
    guarding of events 25
    hiding 20
    indexed choice 21
    indexed parallel composition 21
    interface parallel 20
    interleaving 20
    partial event 22
    prefix 20
    prefix choice 21
    process call 20
    projection 76
    redistribution laws 164, 165
    renaming 20
    traces 22
        initials 23

projection 23
CSP_Z 24, 194
CSP-OZ
Δ list 15, 17
`effect` schema 15, 17
`enable` schema 15, 17
class structure 12
constant parameters 212
dependence analysis 27
failures-divergences semantics . . . 25
initial state schema 13, 17
input parameter 13
interface 12
operation schema 12, 16
declaration part 18
delta list 18
modified variables 17
predicate part 18
referenced variables 17
output parameter 13
parameter 13
semantics 24
set of events 16
simple parameter 13
state 17
projection 17
state invariants 17
state schema 13, 16
state variable 16
initial closure 106
CSP-OZ-DC 11
CSP-Z 10, 193
CSPLChecker 192, 195
cut 66
comparison of single and general 70
correctness criteria
all-or-none 66
disjointness 61
no crossing 63
no reaching back 65
general 66
interval between cut sets 58
single 68
properties 69

D

data abstraction 3, 42
data dependence 33
direct 34
direct by reason 34
initial 33
interference 34
interference by reason 34
synchronisation 34
data dependence graph 33
decomposition 100
correctness 101
no distribution of initial events 120
no split of synchronisation . . . 121
redistribution of CSP processes 122
correctness proof 166
correctness proof of CSP part . . 132
correctness proof of Object-Z part 152, 157
Pareto-optimal 173
unreasonable 173
connection to heuristics 173
weakly dominated 173
decomposition components
`Init` schemas 77
correctness 143, 144
`State` schemas 77
conditions for correct synchronisation 90
correctness proof 93
CSP parts 100
cut variables 83
event sets 97
interfaces 99
operation schemas 83
renaming of channels 98
renaming of events 99
distributivity law 162
properties 161
sets of operations 75
decomposition heuristics
cut size 169
even distribution 170
few addressing 172

few transmission 170
dependence graph 37
DG . . . . . *see* dependence graph
directed model checking 182
Duration Calculus 11

E

E-LOTOS 11
earlier stage 140
equivalence relation 106
Event-B 10
compositional verification 53

F

`FDR2` 25, 53, 192
formal methods 2, 9
formal verification 2

H

hypergraph partitioning 181

I

identity relation 106, 123
integrated formal methods 2, 9

J

Java PathFinder 182

L

labelled transition system 18
CSP part 23
language 43
Object-Z part 19
parallel composition 26
path 19
liveness property 48
LOTOS 10
LTS . . . . . *see* labelled transition system

M

model checking 3, 41, 42
model driven development (MDD) 1
multicriteria optimisation 173, 182

N

NP completeness 192

O

Object-Z 11, 16
compositional verification 53
history semantics 18
semantics 18
structure 12
operational semantics
CSP part 22
CSP-OZ 24
Object-Z part 18

P

partial order reduction 3, 42
PDG . . . . . *see* dependence graph
π-calculus 10
program dependence graph . . . . . *see* dependence graph

R

real-time systems 182
refinement 23

S

safety properties 22
safety property 48
SAL 53
SAT solving 192
software quality assurance (SQA) 1
software testing 1
software verification 2
state explosion 3, 41, 42
symbolic model checking 3, 43
symbolic transition systems . . . 110, 181
symmetry reduction 42
synchronisation dependence 34
realisation of 91
Syspect 184
class diagram 184
component diagram 187
countertrace plug-in 196
decomposition framework 188
decomposition plug-in 189
export to LaTeX 187
mass validation 191
property view 185

state machine 186

T

timed automata 10
trace equivalence 23
trace refinement 23
Two Phase Commit Protocol 175

U

Unified Modelling Language 1, 9
activity diagram 198
compositional verification 53

Z

Z notation 11
axiomatic definition 12
basic type 11
free type 11