Property Preserving Development and Testing for CSP-CASL

Temesghen Kahsai

Submitted to the University of Wales in fulfillment of the requirements for the degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

May 2010

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

Summary

This thesis describes a theoretical study and an industrial application in the area of formal systems development, verification and formal testing using the specification language CSP-CASL. The latter is a comprehensive specification language which allows to describe systems in a combined algebraic / process algebraic notation. To this end it integrates the process algebra CSP and the algebraic specification language CASL.

In this thesis we propose various formal development notions for CSP-CASL capable of capturing informal vertical and horizontal software development which we typically find in industrial applications. We provide proof techniques for such development notions and verification methodologies to prove interesting properties of reactive systems.

We also propose a theoretical framework for formal testing from CSP-CASL specifications. Here, we present a conformance relation between a physical system and a CSP-CASL specification. In particular we study the relationship between CSP-CASL development notions and the implemented system.

The proposed theoretical notions of formal system development, property verification and formal testing for CSP-CASL, have been successfully applied to two industrial application: an electronic payment system called EP2 and the starting system of the BR725 ROLLS-ROYCE jet engine control software.

Acknowledgements

This thesis would not have been possible without the support of a number of people who directly or indirectly influenced my work during the last four years in Swansea. I am very grateful to all these people. Especially I would like to thank my supervisor Dr. Markus Roggenbach for his constant encouragement, guidance, patience and support. I appreciate all his contributions of time, ideas and funding to make my Ph.D experience productive and stimulating. I thank Prof. Holger Schlingloff, for valuable lessons on how important is to apply formal methods in real world systems and for hosting me many times in Berlin.

I am very grateful to my external examiner Prof. Hans-Jorg Kreowski and my internal examiner Dr. Monika Seisenberger. Their valuable comments made this thesis better.

I am grateful to all members of the Department of Computer Science as whole. Special thanks goes to the theory research group; in particular Prof. Faron Moller for his encourgement, and my second supervisor Dr. Ulrich Berger.

I also would like to thank past and present members of '*Processes and Data*' research group: Andy Gimblett, Liam O'Reilly, Gift Samuel and Phillip James, who have been a pleasure working with them. Within this context I am also grateful to Erwin R. Catesbeiana (Jr) for *preserving* my sanity in the last four years.

My time in Swansea was made enjoyable in large part due to the many friends; special thanks goes to Ben Spencer (for being an awesome cubical mate) and Fredrik (for proof reading part of the thesis). A special thanks goes also to all the resident tutors at HSV. I am also grateful to all my Eritrean friends and relatives in London; special thanks to Sara and Tiblez.

My final words are reserved to my family: my mother Abrehet, my sisters Yordanos, Selamawit, Lidia, my brother Efrem and my two nieces Alex and Yoram. Although I am so far away from them, their continuous support and unconditional love has been vital, thank you very much.

TEMESGHEN KAHSAI

Contents

List of Figures

To
Yeshareg, Abrehet & Azieb Ghebremariam.

Introduction

(CHAPTER ...1)

Introduction

Contents

THIS thesis describes a theoretical and industrial application in the area of formal systems development, verification and formal testing using the specification language CSP-CASL.

CSP-CASL [Rog06] integrates specification of data and processes in order to describe in an expressive way reactive systems. Typically, process algebra have paid little attention to modelling data, whereas algebraic specification languages have not directly supported the modeling of concurrent process behavior. CSP-CASL integrates the process algebra CSP [Hoa85, Ros98, AJS05, Hoa06] with the algebraic specification CASL [Mos04, ABK+02]. The general idea is to describe reactive systems in the form of processes based on CSP operators, where the communications of these processes are the values of data types, which are loosely specified in CASL.

This chapter provides an introduction to the main body of this thesis. We first give a general introduction to the notions of formal vertical and horizontal development as well as the notion of formal testing. We then present the main contribution of this thesis. Finally, we conclude this chapter by giving the outline of the thesis and the list of articles in which parts of this work have been published.

1.1 Formal software development and testing

The theoretical aspect of this thesis is concerned with the notions of vertical and horizontal development for CSP-CASL. Vertical development means to change the level of

abstraction, i.e., from an abstract specification to a concrete specification. Horizontal development means to enlarge the system with new feature – staying in the same level of abstraction.

We study new development notions for CSP-CASL capable of capturing informal vertical and horizontal software development which we typically find in industrial applications. We provide proof techniques for such development notions and verification methodologies to prove interesting properties of reactive systems. We present also a theoretical framework for formal testing from CSP-CASL specifications. Here, we provide a conformance relation between a physical system and a CSP-CASL specification. In particular we study the relation between CSP-CASL development notions and the implemented system.

In the next two subsections we give a brief introduction to the two main theoretical aspects that this thesis is concerned with: *property preserving system development* and *formal software testing*.

1.1.1 Property preserving system development

In a vertical development, a system is developed vertically in a step-by-step fashion. Such an approach has been central to software engineering at least since Wirth's seminal paper on program development [Wir71] in 1971. Such a development starts with an abstract specification, which defines the general setting, e.g., it might define the components and interfaces involved in the system. In several design steps this abstract specification is then further developed towards a design specification which can be implemented directly. In each of these steps some design decisions are taken and implementation issues are resolved. A design step can for instance refine the type system, or it might set up a basic dialogue structure. It is essential, however, that these design steps preserve properties. This idea is captured by the notion of *refinement*. Refinement is typically performed in several steps, and at each step we verify that any behavior of the refined model is allowed by the previous model, thus ensuring that the final detailed model is correct with respect to the original system-level model. For a notion of refinement to be useful, it should reflect the ways in which we might want to make concrete our abstract specification.

In the other development direction, horizontal system development, new functionality or features are added to an existing system. For the corresponding software development process, this means that the specification of an advanced product is developed by enhancement and combination of basic specifications. Such a concept allows one to capture the notion of software product lines.

Today, very few software systems are developed from scratch; most systems are derived by extending or enhancing previous versions. Thus, traditional engineering approaches, in which a complete system is derived from a given set of informal or formal specifications, are only partially adequate. This holds in particular for *software product lines*, where a set of similar products is targeted. The CMU SEI defines a software product line to be a "set of software-intensive systems that share a common, managed set of features satisfying the

specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [CMU]. Thus, the individual products in a product line have a similar "look-and-feel", however, they differ in that one product may offer more functionality than the other one. In a product line, there are low-end products with a basic set of features, specialized products for particular markets, and high-end products which combine many features.

1.1.2 Formal software testing

Dijkstra observed that a major limitation of software testing is that it can only show the presence of faults, and not their absence [DDH72]. Despite this obvious limitation, software testing is recognized as a necessary means of system verification. Even when other program verification techniques such as static analyses and formal proofs are employed, testing is still considered necessary to complement such techniques, and to build greater confidence in the system being developed. In "Testing: A Roadmap" [Har00], M.J. Harrold points out that software quality will become the main criteria for success in the software industry. She refers to software testing as the critical element in software quality.

It is well accepted that formal specifications can be useful for software testing. *M-C Gaudel*, in the article "Testing can be formal too" [Gau95], gives a first formal treatment of testing. For other pioneering papers survey we refer to [BGM91, BCFG86, Bri88, Cho78, GMH81, LY94]. On the other side *Brinksma* in [Bri99] illustrates an overview that formal testing can be successfully applied to industrial applications.

Systematic testing is the most important quality assurance method in software and systems design. Testing can be done at all stages during the design, e.g., on unit-, integration- and system level. System tests often are conceived as black-box-tests, where the inner structure of the system is hidden from the observer's view. In contrast to formal verification, black-box-testing is concerned with all parts of a computational system – software, middleware and hardware. The 'black box' view abstracts from the actual implementation details and considers the observable behaviour of the system only. The main purpose of testing is: on one side to determine that the system under test (SUT) conforms to the specified intended behaviour; and on the other side to determine whether or not the SUT contains errors, where an error is a deviation of the actual behaviour from the intended behaviour of the SUT. If in a systematic test no errors are found, this can increase the confidence that the system is apt for its intended use.

The computational behaviour of a system can be denoted in a formal way, e.g., as a set of sequences of input/output events which occur at a certain point of control and observation (PCO). Such sequences are called *traces* of the SUT. Testing then becomes the task of comparing specified and traceable behaviour of a computational system, i.e., checking whether all intended behaviour is realised by traces, no unintended behaviour can be observed, and other behaviour is neither forced on the system nor inhibited.

1.2 Project aims and contributions of this thesis

The aims set at the beginning of this Ph.D project can be summarized as follows:

- To define and investigate formal development notions that are capable of mirroring typical informal development relations as are present in industrial application.
- To provide tool support for analysing CSP-CASL specifications, e.g., to prove deadlock freedom, as well as for verifying the formal development relations between them.
- To set up a testing approach that relates the formal specifications in CSP-CASL with real world systems.

The main contributions of this thesis are summarised in the following three subsections.

1.2.1 Property preserving development notions for CSP-CASL

We have developed two directions of development notions for CSP-CASL capable of mirroring typical *vertical* and *horizontal* developments.

For the vertical development, we propose refinement notions based on model class inclusion with arbitrary change of signature. Our notions of refinement for CSP-CASL are based on refinements developed in the context of the single languages CSP and CASL. In the context of algebraic specification, e.g., [EK99] provides an excellent survey on different approaches. For CSP, each of its semantical models comes with a refinement notion of its own. There are for instance traces refinement, failure/divergences refinement, and stable failures refinement [Ros98]. For *system development* one is often interested in liberal notions of refinements, which allow substantial changes in the design. For *system verification*, however, it is important that refinement steps preserve properties. The latter concept allows one to verify properties already on abstract specifications – which in general are less complex than the more concrete ones. The properties, however, are preserved over the design steps. These two purposes motivate our study of various refinement notions.

We also develop proof support for the proposed development notions of CSP-CASL. For the refinement notion, we decompose a CSP-CASL refinement into a refinement over CSP and a refinement over CASL alone. We show how to use existing tools to discharge the arising proof obligations. Reactive systems often exhibit the undesirable behaviour of deadlock or divergence (livelock), which both result in lack of progress in the system. Here, we develop proof techniques based on refinement for proving deadlock freeness and divergence freeness.

For the horizontal development, we have developed a notion of enhancement for CSP-CASL. In a horizontal development new functionality or features are added to an existing system. For the corresponding software development process, this means that the specification of an advanced product is developed by enhancement, which allows for the

combination of basic specifications. Such concept allows to capture the notion of software product lines. Here, we have identified some enhancement patterns that are capable of capturing some basic horizontal development step. Using such patterns we prove an enhancement relation between CSP-CASL specifications.

1.2.2 Testing theory for CSP-CASL

We propose a theory for the evaluation of test cases with respect to the specification language CSP-CASL. In a formal systems development process, an abstract specification can be refined to a concrete implementation, where all design decisions have been fixed and which has a deterministic behaviour. In our approach, we can build test suites for any level of abstraction in this process. It is possible that test cases are constructed either from the specification or independently from it. Therefore, it is possible to structure a test suite according to the features of the system under test (SUT). Each test case checks the correct implementation of a certain feature according to a particular specification. The specification determines the alphabet of the test suite, and the expected result of each test case. The expected result is coded in a colouring scheme of test cases. If a test case is constructed which checks for the presence of a required feature (according to the specification), we define its colour to be *green*. If a test case checks for the absence of some unwanted behaviour, we say that it has the colour *red*. If the specification does neither require nor disallow the behaviour tested by the test case, i.e., if an SUT may or may not implement this behaviour, the colour of the test case is defined to be *yellow*. During the execution of a test on a particular SUT, the *verdict* is determined by comparing the colour of the test case with the actual behaviour. A test *fails*, if the colour of the test case is green but the SUT does not exhibit this behaviour, or if the colour is red but the behaviour can be observed in the SUT. The execution of a yellow test case yields an inconclusive verdict. Otherwise, the test passes.

Moreover, we develop a theoretical framework to relate the evaluation of test cases with the notion of vertical and horizontal development in CSP-CASL. In particular for the vertical development we show that test cases developed for abstract specification preserve their colour under a well-defined refinement notion. This approach ensures that test cases which are designed at an early stage can be used without modification for the test of a later development step.

As for horizontal development, we show that test cases developed for a basic specification preserve their colour after the enhancement step. In a horizontal development, the advanced product incorporates features from more basic versions. Even if all features of the basic products have been thoroughly tested, it is necessary to validate that these features still work correctly in the enhanced version. Usually the design and testing of the basic version is completed before the advanced version is begun; in this case, for all basic features elaborate test cases are available. Our approach allows to re-use the test cases of the basic specification in a test suite for the enhanced specification.

1.2.3 Industrial applications

The presented theoretical framework have been successfully applied to two 'real' world systems. Those are:

Electronic payment system EP2. The EP2 system is an electronic payment system and it stands for 'EFT/POS 2000', short for 'Electronic Fund Transfer/Point Of Service 2000'. This is a joint project established by a number of (mainly Swiss) financial institutes and companies in order to define EFT/POS infrastructure for credit, debit, and electronic purse terminals in Switzerland (`www.eftpos2000.ch`). The system consists of seven autonomous entities: CardHolder, Point of Service, Attendant, POS Management, Acquirer, Service Center and Card. These components are centered around an EP2 Terminal. The EP2 specification consists of twelve documents, each of which describe the different components or some aspect common to the components. The way that the specifications are written is typical of a number of similar industrial application. CSP-CASL is able to match such a document structure by a library of specifications, where the informal design steps of the EP2 specification are mirrored in terms of a formal refinement relation defined in the previous sections. A first modeling approach of the different levels of EP2 in CSP-CASL has been described in [GRS05]. Here, we have extended the modeling in more detail by carrying out the specification of the various EP2 components at different levels of abstraction. We have systematically proven the refinement steps of the various level of specification using CSP-CASL-PROVER [OIR09]. Moreover, we have proven that the interaction of the EP2 components is deadlock free. Again this is done systematically using CSP-CASL-PROVER.

For the testing part, we evaluate test cases using CSP-CASL-PROVER. Moreover, we present a testing framework for a EP2 payment terminal. Such testing framework, tests the EP2 payment terminal in a hardware-in-the-loop testing fashion.

ROLLS-ROYCE jet engine Here, we apply the theory of testing from CSP-CASL to the starting system of the jet engine ROLLS-ROYCE BR725 control software. The BR725 is a newly designed jet engine for ultra-long-range and high-speed business jets. It is part of the BR700 family. We model the starting system in CSP and validate our model using the CSP simulator PROBE [Ltd03]. We then evaluate the test suites against the formal model. Such evaluation is done using the model checker FDR2 [Ltd06]. We execute our test suite in an in-the-loop setting on the so-called "rig".

1.3 Synopsis

The rest of the thesis is organized as follows. Chapters 2 to 5 describe the background material of this thesis.

- Chapter 2 and 3 introduces the specification languages CSP and CASL respectively. Here, we outline their key features and present different notions of refinement for both languages. Both languages are the key ingredient of the specification language we will concentrate on in this thesis.
- Chapter 4 describes the combination of CSP and CASL to form the specification language named CSP-CASL.
- Chapter 5 contains references to approaches which define the context of this thesis and provides short characterisations of the cited work. Here, we give an overview of different approaches and formalisms which combines data and process specifications, as well as notions of system developments. We also give an overview of related approaches in the area of specification based testing.

Chapters 6 to 8 describe the theoretical framework of property preserving development notions for CSP-CASL.

- Chapter 6 presents a theory of CSP-CASL refinement and CSP-CASL enhancement.
- Chapter 7 illustrates proof support for CSP-CASL refinement and CSP-CASL enhancement.
- Chapter 8 describes how we can verify interesting properties using the newly introduced refinement notion for CSP-CASL, namely analysis of deadlock and livelock freedom.

Chapter 9 introduces the theoretical framework of testing from CSP-CASL specifications.

- Section 9.1 illustrates the challenges for CSP-CASL based testing and reasonable expectations for specification based testing.
- Section 9.2 describes the notion of a CSP-CASL test process and the expected result of a CSP-CASL test process with respect to a CSP-CASL specification.
- Section 9.4 describes how we execute test cases on the SUT as well as the derivation of test verdicts.
- Section 10.1 illustrates the relation between CSP-CASL refinement and test evaluation.
- Section 10.2 illustrates the relation between the CSP-CASL enhancement and test evaluation. In particular we illustrate the notion of test case reuse in the setting of software product lines.

Chapters 11 and 12 present two industrial applications of the theoretical framework presented in the previous chapters.

- Chapter 11 introduces an electronic payment system called EP2. Here, we illustrate the modeling of EP2 in CSP-CASL; we prove the refinement of the different level of abstraction. We analyze EP2 for deadlock and livelock freedom. We also present the tool TEV– a testing framework for EP2.

- Chapter 12 describes the work done by the author during a two months internship at ROLLS-ROYCE part of the BR725 system verification team. The work concentrates on the specification-based testing in the context of control software for the jet engine ROLLS-ROYCE BR725.

Finally, in Chapter 13 we summarize the overall contribution of this work and conclude the thesis by considering possible future work.

1.4 Publications

Some parts of this thesis have been published in the following articles:

1. Temesghen Kahsai, Greg Holland, Markus Roggenbach, and Bernd-Holger Schlingloff.
TOWARDS FORMAL TESTING OF JET ENGINE ROLLS-ROYCE BR725. In Proceedings of the 18th International Conference on *Concurrency, Specification and Programming*, pp. 217-229, 2009.

2. Temesghen Kahsai and Markus Roggenbach.
PROPERTY PRESERVING REFINEMENT NOTIONS FOR CSP-CASL. In *Recent Trends in Algebraic Development Techniques*, LNCS 5486, pp. 206-220, 2009.

3. Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff.
SPECIFICATION-BASED TESTING FOR SOFTWARE PRODUCT LINES. In *Software Engineering and Formal Methods 2008*, IEEE Computer Society, pp. 149-159, 2008.

4. Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff.
SPECIFICATION-BASED TESTING FOR REFINEMENT. In *Software Engineering and Formal Methods 2007*, IEEE Computer Society, pp. 237-247, 2007.

My main contribution to these papers are respectively:

1. The work described in this paper is the result of my two months internship at ROLLS-ROYCE. I have done most of the specifications (M. Roggenbach helped in the specification of one activity diagram - Section 3.2). I have done all the proofs of test case evaluation and the execution of test cases in the rig (Section 4). Overall, I have written most of the paper.

2. I wrote the whole introduction section and most of the other sections. Specifically I have done most of the technical proofs: proof of the decomposition theorem (Theorem 1), proof of deadlock analysis (Lemma 1 and Theorem 2, 3), proof of livelock analysis (Lemma 2 and Theorem 4, 5). I wrote the whole section on EP2 (Section 4); here, I have done all the proofs of refinement, deadlock and livelock freedom.

3. I contribute to the CSP-CASL specification of the remote control unit product line (Section 3). I wrote the detailed proofs of Lemma 3.1, 3.3 and Theorem 3.5, 4.1. I have done the detailed proofs of refinement and enhancement of the remote control

unit. I contributed in proving the colour of the test cases and implemented the SUT as well as the implementation of the testing environment for the remote control unit (Section 5).

4. I contributed to the technical proof of well-behaved development notion of data and process refinement (Section 5.2). I wrote the whole section on specification, implementation and testing of the binary calculator example, as well as the colouring proof of the various test cases (Section 6).

This thesis contains more results than the one published in the aforementioned papers. For each paper we indicate the extra results included in this thesis.

1. Nothing more added. (See Chapter 12).
2. The following are new results which are reported in Chapters 6, 7, 8, 11:
   - New definition of refinement with arbitrary change of signature in the traces, stable failure and failures/divergences CSP models (c.f. Lemma 6.1.3, 6.1.4, 6.1.5, 6.1.6, Theorem 6.1.8 and Definition 6.1.9).
   - Proof of that process and data refinement with change of signature are special cases of CSP-CASL refinement (c.f. Definition 6.1.10, 6.1.11 and Lemma 6.1.12).
   - Decomposition theorem with change of signature (c.f. Theorem 7.1.1).
   - Deadlock analysis with change of signature (c.f. Theorem 8.1.2).
   - Livelock analysis with change of signature (c.f. Theorem 8.2.2).
3. Nothing more added. (See Chapters 6, 7 and Sections 10.2, 10.3.).
4. The following are new results which are reported in Chapter 9 and Section 10.1:
   - Detailed proof of the syntactic characterization for the traces condition (c.f. Theorem 9.3.1).
   - New syntactic characterization for the failures condition (c.f. Definition 9.3.2, Theorem 9.3.3, Corollary 9.3.4, Corollary 9.3.5).
   - Complete syntactic characterization to prove green, red and yellow test cases.
   - Testing with new refinement notion based on arbitrary change of signature (c.f. Theorem 10.1.1, Theorem 10.1.2).

Moreover the thesis contains some results which are not published yet. Those are related to the specification, verification and testing of the industrial case study EP2.

PART I

Background

(CHAPTER ...2)

Communicating Sequential Processes (CSP)

Contents

THE process algebra CSP [Hoa85, Ros98, AJS05, Hoa06] – *Communicating Sequential Processes* – is a well established and widely used formalism. In CSP-CASL it is deployed for the description of the reactive/process part. In this chapter we introduce the language and give an overview of its key features, syntax and semantics. We present the different denotational models of CSP, namely the *traces model*, *stable-failure model* and *failures/divergences model*. Each of these semantical model comes with a refinement notion. We also discuss how each semantical model are deployed for the verification of some interesting properties of concurrent systems.

2.1 CSP – fundamental concepts

Process algebras have proved to be a valuable mathematical tool to reason about the behaviour of concurrent and communicating systems. The process algebra CSP is one of a number of formalisms for modelling and verifying *concurrent reactive systems*, i.e., systems are described in terms of *interactions* with other systems in a concurrent way.

CSP is one of the three process algebras which have historically dominated the field; the others are CCS (*Calculus of Communicating Systems*) [Mil89] and ACP (*Algebra of Communicating Processes*) [BW90]. Some recent developments of particular interest includes: π-*calculus* [Mil99] which introduces the notion of process mobility; *spi-calculus* [AG97] a variant of the π-*calculus* that includes cryptographic features to reason about security protocols; *ambient-calculus* [CG98], which introduces the notion of explicit locations.

CSP has been deployed successfully in the industrial context, often applied in areas as varied as distributed databases [Ros98], parallel algorithms [IRG05], train control systems [BS99], fault-tolerant systems [BKS97], and analysis of security protocols [RSG+01].

The basic units of abstraction in CSP are *processes*, the set of *communication events* or *alphabet of communication* and a logic on the alphabet. Here, processes are named entities which live in some environment. Communications are instantaneous, atomic synchronisations between processes, and usually carry some semantic content, for instance a value which could be regarded as a 'message'. Another assumption about communication events is that an event occurs only when all its participants are ready to execute it. Communication names may be prefixed, this is represented by the use of *channels*. For example, the event *button*.1 conceptually represents communication of the value '1' over the channel *button*. Here, we are just communicating the value *button*.1 which intuitively could mean the pressing of a button that transmits the value '1'. Semantically however, channels are just syntactic sugar – very useful one. A set of communications is called an *alphabet*; $\mathcal{A}$ refers to the 'total' alphabet of communications over which a system of processes is defined. We will devote much discussion to the alphabet of communication in Chapter 4, where we introduce CSP-CASL.

A *process equation* binds a process to a name, which may be referenced by other processes. Such names may be *parameterised*; a parameterised process represents a family of processes, one for each possible combination of parameter values. In the next sub-section we describe the syntax of CSP process operators.

2.1.1 CSP – Syntax

Figure 2.1 illustrates the basic CSP processes P. It involves elements $a \in \mathcal{A}$ as communications, subsets $X, Y \subseteq \mathcal{A}$ as synchronization sets in parallel operators or for hiding certain communications, uses binary relations $R \subseteq \mathcal{A} \times \mathcal{A}$ in order to describe renaming, and allows non-further specified formulae φ in its conditional.

Let us discuss some of the operators reported in Figure 2.1.

Primitive processes The process *SKIP* represents *successful termination*: it never communicates anything in the alphabet, however signals successful termination. Conversely, *STOP* represents *deadlock*: it represents a process which has entered a state in which is not able to communicate. We also have *DIV*, representing divergence, which also is in a state of perpetual non-communication; however the reason for doing so is different from *STOP*. The process *DIV* represents *livelock*: it is engaging in an infinite sequence of internal, non-observable actions. We will discuss more about such process in the coming sections.

Action prefix: The process $P = a \rightarrow SKIP$, offers the communication a and then behaves like *SKIP*, hence it terminates successfully.

$$
\begin{array}{lll}
P & ::= & STOP & \%\% \text{ deadlock process} \\
& | & SKIP & \%\% \text{ terminating process} \\
& | & DIV & \%\% \text{ divergence process} \\
& | & a \rightarrow P & \%\% \text{ action prefix} \\
& | & ?x : X \rightarrow P & \%\% \text{ prefix choice} \\
& | & P \fatsemi P & \%\% \text{ sequential composition} \\
& | & P \,\Box\, P & \%\% \text{ external choice} \\
& | & P \sqcap P) & \%\% \text{ internal choice} \\
& | & P \,|[\, X \,]|\, P & \%\% \text{ generalized parallel} \\
& | & P \;\; |[\, X \mid Y \,]|| \;\; P & \%\% \text{ alphabetized parallel} \\
& | & P \,||\, P & \%\% \text{ synchronous parallel} \\
& | & P \,|||\, P & \%\% \text{ interleaving} \\
& | & P \setminus X & \%\% \text{ hiding} \\
& | & P[[R]] & \%\% \text{ relational renaming} \\
& | & \textbf{if } \varphi \textbf{ then } P \textbf{ else } P & \%\% \text{ conditional}
\end{array}
$$

$a \in \mathcal{A}$ and $X, Y \subseteq \mathcal{A}$ and $R \subseteq \mathcal{A} \times \mathcal{A}$
φ : formulae.

Figure 2.1: Syntax of basic CSP processes.

Let X be a set of communications, then $?x : X \rightarrow P(x)$ is a process which will communicate any value $x \in X$ and then behaves like $P(x)$. Such operator allows a choice of values to be communicated. We can use a 'channeled' version in the following way: $c?x \rightarrow P(x)$, here we communicate $c.x$. In case we would like to send a value over a channel, we write $c!x \rightarrow P(x)$ which is a syntactic sugar for $c.x \rightarrow P(x)$. Here, the sending process is just the process of *choosing* which value is to be synchronized on.

Sequential composition The process $P \fatsemi Q$ is a process which behaves like P. Should P terminates, it behaves like Q. It is very useful for composing named processes, hence useful for modularity. For example:

$$
\begin{aligned}
P1 &= a \rightarrow SKIP \\
P2 &= c \rightarrow SKIP \\
Comp &= P1 \fatsemi P2
\end{aligned}
$$

The process *Comp* is equivalent to $a \rightarrow c \rightarrow SKIP$.

Choice operators *External choice operator*: The process $P \,\Box\, Q$ offers the environment the choice of the first communication of P and Q, and then behaves accordingly. For example:

$$ExtC = a \rightarrow SKIP \,\Box\, b \rightarrow STOP$$

if the environment offers 'a', the process *ExtC* will communicate 'a' and then successfully terminate. Otherwise, if the environment offers 'b', *ExtC* will communicate 'b'

and then deadlock. If both sides offer the same communication, the choice of which side is taken *internally*.

Internal choice operator: The process $P \sqcap Q$ behaves either like P or like Q. Here, the choice is made internally to the process; this means for the environment point of view the choice is made in a nondeterministic way. For example:

$$IntC = a \rightarrow SKIP \sqcap b \rightarrow STOP$$

If the environment offers 'a', the process *IntC* may choose to communicate 'b' and then there is a deadlock between the environment and *IntC*. Only if the environment offers both 'a' and 'b' the process *IntC* is obliged to communicate, because it must choose one of the alternatives.

Parallel operators *Generalized parallel*: In the process $P \,|[\, X \,]|\, Q$, both P and Q synchronize on all events in X, and for events outside X both processes proceed independently. For example:

$$GenP1 = a \rightarrow SKIP \sqcap c \rightarrow SKIP$$
$$GenP = GenP1 \,|[\, \{b, c\} \,]|\, GenP1$$

The process *GenP*1 can choose internally whether to communicate 'a' or 'c'; putting two *GenP*1 in a generalized parallel, both processes must synchronize on everything in $\{b, c\}$. If one of *GenP*1 chooses to communicate 'c', they must both do so. Clearly, then *GenP* can deadlock, if one side offers 'a' and the other side offers 'c'.

Alphabetized parallel: Let X and Y be sets of communications, then in the process $P \;\; |[\, X \,|\, Y \,]|\;\; Q$, P is allowed to communicate in the set X, while Q communicates in the set Y. However, they must agree on events in the intersection $X \cap Y$. Thus, $P \;\; |[\, X \,|\, Y \,]|\;\; Q$ is equivalent to $P \,|[\, X \cap Y \,]|\, Q$.

Synchronous parallel: The process $P \;||\; Q$ behaves like P and Q in which every event of P and Q are totally synchronized; that is, it only communicates events on which they both agree. Thus, $P \;||\; Q$ is equivalent to $P \,|[\, \mathcal{A} \,]|\, Q$ (or $P \;\; |[\, \mathcal{A} \,|\, \mathcal{A} \,]|\;\; Q$), where $\mathcal{A}$ is the alphabet of communications.

Interleaving: The process $P \;|||\; Q$ is a process where P and Q run in parallel, independent of one another; if the environment offers a communication which both P and Q could engage in, exactly one does so, the choice being nondeterministic. Thus, we have that $P \;|||\; Q$ is equivalent to $P \,|[\, \{\} \,]|\, Q$. For example:

$$IntP = a \rightarrow b \rightarrow SKIP \;|||\; c \rightarrow a \rightarrow SKIP$$

The process *IntP* can initially engage in 'a' or 'c'; supposing it engages in 'a', it *may still* engage in 'c' subsequently: it presents a choice of 'b' and 'a'. Otherwise, if it initially engages in 'c', then it can subsequently only engage in 'a', but after that it might offer 'a' and 'b', or 'a' and 'c', depending on the (internally) chosen 'a'.

Hiding Let X be a set of communications then $P \setminus X$ is a process which behaves like P except that any event in X is not observable from outside $P \setminus X$.

Renaming Let $R \subseteq \mathcal{A} \times \mathcal{A}$ be a relation over the alphabet of communication, then the process $P[[R]]$ behaves like the process P where all the events x of P are renamed in y for $(x, y) \in R$. For example, let $R = \{(a, d), (b, j)\}$ be a relation:

$$P1 = a \rightarrow STOP \,\Box\, b \rightarrow c \rightarrow SKIP$$
$$Ren = P1[[R]]$$

The process *Ren* is equivalent to $d \rightarrow STOP \,\Box\, j \rightarrow c \rightarrow SKIP$.

CSP introduces recursion in the form of systems of process equations. Parameterised processes are defined in terms of basic process expressions including also process names (see Figure 2.1):

$$\begin{array}{rll} P ::= & \ldots & \\ & | & PName \\ & | & PName(x_1, \ldots, x_n) \end{array}$$

Here $x_1, \ldots, x_n$ are variables over $\mathcal{A}$. A CSP process equation is of the form:

$$\begin{array}{rll} \text{CSP\_}equation ::= & PName = P & \\ & | & PName(x_1, \ldots, x_n) = P \end{array}$$

Moreover in the CSP literature, two further processes are presented: *RUN* and *CHAOS*.

$$RUN_X = ?x : X \rightarrow RUN_X$$
$$CHAOS_X = STOP \sqcap (?x : X \rightarrow CHAOS_X)$$

For a set of events $X \in \mathcal{A}$, the process RUN_X can always communicate any member of X desired by the environment. The process $CHAOS_X$ can always choose to communicate or reject any member of X.

Before illustrating the various semantical notion of CSP, in Figure 2.2 we report some standard CSP notation that will be used in the upcoming sections. For a full glossary of mathematical notation used in the context of CSP, the reader can refer to [Ros98].

$x \setminus y$	Difference $(= \{a \in x \mid a \notin y\})$.
$\mathbb{P}(x)$	Power set $(= \{y \mid y \subseteq x\})$.
$\langle\rangle$	Empty sequence
$\langle a_1, \ldots, a_n \rangle$	The sequence containing $a_1, \ldots, a_n$ in that order.
$s \frown t$	Concatenation of two sequences .
$\sharp s$	Length of s.
$s \setminus X$	Hiding all members of X deleted from s.
$s \upharpoonright X$	Restriction $s \setminus (\Sigma^{\checkmark} \setminus X)$.
$s \leq t$	prefix order $(\equiv \exists u.s \frown u = t)$.

Figure 2.2: Standard CSP notation.

2.2 CSP – denotational semantic model

CSP offers a number of approaches to semantics. A process written in CSP may be understood in terms of *operational* semantics (where the process is transformed to a *labelled transition system*, with transitions representing communications); or in terms of *algebraic* semantics (where properties of a process — such as equivalence to some other process — may be deduced by syntactic transformations on the process text following a set of algebraic laws); or in terms of *denotational* semantics (where the process corresponds to a value in some mathematical model, typically a *complete partial order* or a *complete metric space*). The latter is the dominant one, and of particular interest for our work. In the next subsections we describe three denotational models.

2.2.1 Traces model – $\mathcal{T}$

The **traces model** $\mathcal{T}$, denotes a CSP process according to its *traces*, which are the set of sequences of communications in which the process is willing to engage.

Let $\mathcal{A}^{*\checkmark} = \mathcal{A}^* \cup \{s \frown \langle\checkmark\rangle \mid s \in \mathcal{A}^*\}$ be the alphabet of communications, where $\checkmark \notin \mathcal{A}$ represents the event of successful termination. Formally in the traces model each process is identified by a set $T \subseteq \mathcal{A}^{*\checkmark}$ that satisfies the following healthiness conditions:

T1. T is nonempty; i.e., it always contains the empty trace $\langle\rangle$.

T2. T is prefix-closed; i.e., if $s \frown t \in T$ then $s \in T$.

Given a CSP process P, the traces of P are denoted as $traces(P)$. In Figure 2.3 we report the semantic clauses of the basic processes in the traces model $\mathcal{T}$.

STOP never communicates anything: its set of traces consists only of the empty trace $\langle\rangle$; the traces of an action prefix process are the traces of the prefixed process P, each prefixed with the event a first communicated and the empty trace added. In the clause of the prefix choice $?x : X \to P$, which is the only way to introduce a variable x, every free occurrence of x in the process P is syntactically substituted by a communication.

As an example let us consider the following processes:

$$
\begin{aligned}
P &= a \to b \to STOP \\
R &= a \to b \to SKIP \,|[\,\{a\}\,]|\, a \to c \to SKIP \\
Q &= (a \to SKIP) \,\Box\, (b \to c \to STOP)
\end{aligned}
$$

Then, the trace set of P, R and Q are given by:

$$
\begin{aligned}
traces(P) &= \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\} \\
traces(R) &= \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle, \langle a, c\rangle, \langle a, c, b\rangle, \langle a, b, c, \checkmark\rangle, \langle a, c, b, \checkmark\rangle\} \\
traces(Q) &= \{\langle\rangle, \langle a\rangle, \langle a, \checkmark\rangle, \langle b\rangle, \langle b, c\rangle\}
\end{aligned}
$$

$$
\begin{array}{lcl}
traces(STOP) & = & \{\langle\rangle\} \\
traces(SKIP) & = & \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(DIV) & = & \{\langle\rangle\} \\
traces(a \rightarrow P) & = & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces(P)\} \\
traces(?x : X \rightarrow P) & = & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces(P[a/x]), a \in X\} \\
traces(P \fatsemi Q) & = & (traces(P) \cap A^*) \\
 & & \cup\ \{s \frown t \mid s \frown \langle\checkmark\rangle \in traces(P), t \in traces(Q)\} \\
traces(P \,\Box\, Q) & = & traces(P) \cup traces(Q). \\
traces(P \sqcap Q) & = & traces(P) \cup traces(Q). \\
traces(P \,|[\, X \,]|\, Q) & = & \bigcup \{s \,|[\, X \,]|\, t \mid s \in traces(P) \wedge t \in traces(Q)\} \\
traces(P \ |[\, X \,|\, Y \,]|\ Q) & = & \{s \in (X \cup Y)^{*\checkmark} \mid s \upharpoonright X \cup \{\checkmark\} \in traces(P) \wedge \\
 & & \qquad\qquad s \upharpoonright Y \cup \{\checkmark\} \in traces(Q) \quad\} \\
traces(P \,||\, Q) & = & traces(P) \cap traces(Q) \\
traces(P \,|||\, Q) & = & \bigcup \{s \,|||\, t \mid s \in traces(P) \wedge t \in traces(Q)\} \\
traces(P \setminus X) & = & \{s \setminus X \mid s \in traces(P)\} \\
traces(P[[R]]) & = & \{t \mid \exists s \in traces(P) \,.\, sR^*t\} \\
traces(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) & = & \begin{cases} traces(P); & \varphi \text{ if evaluates to } true \\ traces(Q); & \varphi \text{ if evaluates to } false \end{cases}
\end{array}
$$

Figure 2.3: Semantic clauses for the traces model $\mathcal{T}$.

DEFINITION 2.2.1 (TRACES REFINEMENT) *Let P, Q be arbitrary* CSP *processes. P is a trace refinement of Q written as $Q \sqsubseteq_{\mathcal{T}} P$ if and only if: $traces(P) \subseteq traces(Q)$.*

Two processes P and Q are *traces-equivalent*, $P =_{\mathcal{T}} Q$, if $P \sqsubseteq_{\mathcal{T}} Q$ and $P \sqsupseteq_{\mathcal{T}} Q$, i.e., $traces(P) = traces(Q)$. The process *STOP* is the most refined process in the traces model, i.e., $P \sqsubseteq_{\mathcal{T}} STOP$ for all processes P.

The refinement notion in CSP (independent of the semantic model) has many properties that can be exploited, for example it is *transitive*:

$$P \sqsubseteq Q \ \wedge Q \sqsubseteq T \ \Rightarrow\ P \sqsubseteq T$$

and *monotone*: if $C[.]$ is any process context, namely a process definition with a slot to put a process in, then

$$P \sqsubseteq Q \ \Rightarrow C[P] \sqsubseteq C[Q]$$

The refinement $P \sqsubseteq Q$ is also expressible as the equality $P \sqcap Q = P$. The following lemma proves such equality.

LEMMA 2.2.2 *Let P and Q be arbitrary* CSP *processes. Then,*

$$P \sqsubseteq_{\mathcal{T}} Q \iff P =_{\mathcal{T}} P \sqcap Q$$

PROOF. The trace refinement $P \sqsubseteq_{\mathcal{T}} Q$ holds if and only if $traces(Q) \subseteq traces(P)$. The trace inclusion could be rewritten as $traces(P) \cup traces(Q) =_T traces(P)$. Thus, by the definition of the trace set of $\sqcap$ we have that $P =_{\mathcal{T}} P \sqcap Q$. ∎

The model $\mathcal{T}$ is the weakest of the three denotational models of CSP that we consider. In fact, the traces of internal and external choice are indistinguishable. This indicates that $traces(P)$ does not give a complete description of P, since we would like to be able to distinguish between $P \sqcap Q$ and $P \,\Box\, Q$. For example, the process $a \rightarrow SKIP$ guarantees that if the environment is prepared to engage in the event a and then terminate, then it can engage in the event a and terminate successfully. However, $a \rightarrow SKIP \sqcap a \rightarrow STOP$ does not guarantee that it can engage in the event a and terminate successfully if the environment is ready to engage in the event a and terminates. The traces model identifies both processes as they have the same traces. However, one of them guarantees that it will terminate successfully, but the other does not guarantee.

In terms of verification, the traces model can be deployed for the verification of safety conditions. That is, a process Q which is a trace refinement of a process P, will perform traces already defined in P and nothing more, i.e., $traces(Q) \subseteq traces(P)$. Safety conditions are concerned with the exclusion of traces only.

2.2.2 Stable failure model – $\mathcal{F}$

The stable failure model gives a finer information about processes. For instance it allows us to distinguish between internal and external choice (and much else besides). In particular, it allows us to detect *deadlocked* processes. A failure of a process is a pair (s, X), that describes sets of communications X which a process can fail to accept after executing the trace s. The set X is called the *refusal* set; the process can not perform any event in the set X no matter how long it is offered.

Formally, in the stable failures model, each process is modelled by a pair (T, F) where $T \subseteq \mathcal{A}^{*\checkmark}$ and $F \subseteq \mathcal{A}^{*\checkmark} \times \mathbb{P}(\mathcal{A}^{\checkmark})$, satisfying the following healthiness conditions:

T1. T is non-empty and prefix closed.

T2. $\forall\, s, X : (s, X) \in F \implies s \in T$. This asserts that all traces performed by the failures should be recorded in the traces component T. In other words it establishes consistency between the traces component and the failures component.

T3. $\forall\, s, X : s \frown \langle\checkmark\rangle \in T \implies (s \frown \langle\checkmark\rangle, X) \in F$. If a trace terminates successfully by producing $\checkmark$, then it should refuse all events in $\mathcal{A}^{\checkmark}$ at the stable state after $s \frown \langle\checkmark\rangle$.

F2. $\forall\, s, X : (s, X) \in F \wedge Y \subseteq X \implies (s, Y) \in F$. This asserts that in a stable state if a set X is refused, then any subset Y of X should also be refused.

F3. $\forall\, s, X, Y : (s, X) \in F \wedge \forall\, a \in Y \,.\, s \frown \langle a\rangle \notin T \implies (s, X \cup Y) \in F$.

This asserts that if a process P can refuse the set X of events in some stable state, then the same state must also refuse any set of events Y that the process can never perform after s.

$$
\begin{array}{lcl}
\mathit{failures}(\mathit{STOP}) & = & \{(\langle\rangle, X) \mid X \subseteq \mathcal{A}^{\checkmark}\} \\
\mathit{failures}(\mathit{SKIP}) & = & \{(\langle\rangle, X) \mid X \subseteq \mathcal{A}\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \mathcal{A}^{\checkmark}\} \\
\mathit{failures}(\mathit{DIV}) & = & \emptyset \\
\mathit{failures}(a \rightarrow P) & = & \{(\langle\rangle, X) \mid a \notin X\} \\
 & & \cup \{(\langle a\rangle \frown s, X) \mid (s, X) \in \mathit{failures}(P)\} \\
\mathit{failures}(?x : X \rightarrow P) & = & \{(\langle\rangle, X) \mid \mathcal{A} \cap X = \emptyset\} \\
 & & \cup \{(\langle x\rangle \frown s, X) \mid (s, X) \in \mathit{failures}(P([a/x])), x \in \mathcal{A}\} \\
\mathit{failures}(P \fatsemi Q) & = & \{(s, X) \mid s \in \mathcal{A}^*, (s, X \cup \{\checkmark\}) \in \mathit{failures}(P)\} \\
 & & \cup \{(s \frown t, X) \mid s \frown \langle\checkmark\rangle \in \mathit{traces}(P), (t, X) \in \mathit{failures}(Q)\} \\
\mathit{failures}(P \,\Box\, Q) & = & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}(P) \cap \mathit{failures}(Q)\} \\
 & & \cup \{(t, X) \mid (t, X) \in \mathit{failures}(P) \cup \mathit{failures}(Q), t \neq \langle\rangle\} \\
 & & \cup \{(\langle\rangle, X) \mid X \subseteq \mathcal{A} \wedge \langle\checkmark\rangle \in \mathit{traces}(P) \cup \mathit{traces}(Q)\} \\
\mathit{failures}(P \sqcap Q) & = & \mathit{failures}(P) \cup \mathit{failures}(Q) \\
\mathit{failures}(P \,|[\, X \,]|\, Q) & = & \{(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}), \\
 & & \exists s, t.\ (s, Y) \in \mathit{failures}(P), (t, Z) \in \mathit{failures}(Q), \\
 & & u \in s \,|[\, X \,]|\, t\} \\
\mathit{failures}(P \setminus X) & = & \{(t \setminus X, Y) \mid (t, Y \cup X) \in \mathit{failures}(P)\} \\
\mathit{failures}(P[[R]]) & = & \{(t, X) \mid \exists\, t'.\, (t', t) \in R^*, (t', R^{-1}(X)) \in \mathit{failures}(P)\} \\
\mathit{failures}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) & = & \begin{cases} \mathit{failures}(P); & \varphi \text{ evaluates to } \mathit{true} \\ \mathit{failures}(Q); & \varphi \text{ evaluates to } \mathit{false} \end{cases}
\end{array}
$$

Figure 2.4: Semantic clauses for the stable failure model $\mathcal{F}$.

F4. $\forall\, s :\ s \frown \langle\checkmark\rangle \in T \Longrightarrow (s, \mathcal{A}) \in F$.

> This asserts that if we have any terminating trace $s \frown \langle\checkmark\rangle$, these should refuse $\mathcal{A}$ at the stable state after s.

Similar to the trace semantics, Figure 2.4 illustrates the clauses to determine the failures of the various processes.

In the failures of the renaming operator, $R^{-1}(X) = \{a \mid \exists\, a' \in X\,.\,(a, a') \in X\}$ is the set of all events that map to X under R.

As an example let us consider the following processes over the alphabet $\{a, b\}$:

$$
\begin{aligned}
R &= a \rightarrow \mathit{STOP} \sqcap b \rightarrow \mathit{STOP} \\
Q &= a \rightarrow \mathit{STOP} \,\Box\, b \rightarrow \mathit{STOP}
\end{aligned}
$$

The stable failure set of R and Q are given by:

$$
\begin{aligned}
\mathit{failures}(R) &= \{(\langle\rangle, Y), (\langle\rangle, Z), (\langle a\rangle, X), (\langle b\rangle, X) \mid X \subseteq \{a, b, \checkmark\}, Y \subseteq \{a, \checkmark\}, Z \subseteq \{b, \checkmark\}\} \\
\mathit{failures}(Q) &= \{(\langle\rangle, \{\checkmark\}), (\langle a\rangle, X), (\langle b\rangle, X) \mid X \subseteq \{a, b, \checkmark\}\}
\end{aligned}
$$

Here, R and Q have different failures, i.e., the stable failure model $\mathcal{F}$ can distinguish between internal and external choice.

DEFINITION 2.2.3 (STABLE FAILURE REFINEMENT) *Let P, Q be arbitrary* CSP *processes. P is a stable failure refinement of Q written as $Q \sqsubseteq_{\mathcal{F}} P$ if and only if: $\mathit{traces}(P) \subseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \subseteq \mathit{failures}(Q)$.*

In other words, if every trace s of Q is possible for P and every refusal after this trace is possible for P. Q can neither accept an event nor refuse unless P does. Two processes P and Q are *stable failure-equivalent*, $P =_{\mathcal{F}} Q$, if $P \sqsubseteq_{\mathcal{F}} Q$ and $P \sqsupseteq_{\mathcal{F}} Q$, i.e., $traces(P) = traces(Q)$ and $failures(P) = failures(Q)$. The bottom element in $\sqsubseteq_{\mathcal{F}}$ is $(\mathcal{A}^{*\checkmark}, \mathcal{A}^{*\checkmark} \times \mathbb{P}(\mathcal{A}^{\checkmark}))$, while its top element is $(\langle\rangle, \emptyset)$.

2.2.2.1 Deadlock analysis in CSP

Deadlock is a phenomenon pertaining to networks of communicating processes which occur when two processes can not agree to communicate with each other, thus the whole system becomes permanently frozen. This is potentially catastrophic in safety-critical computing applications. A network which can never exhibit deadlock is said to be deadlock-free.

The deadlock problem was first identified by Dijkstra [Dij02] in the early days of multi-user operating systems. Early work focused on the scenario of *user-resource* networks, where a collection of user processes compete for allocation of a set of shared resources, without any direct communication between the user processes.

In CSP deadlock is represented by the process *STOP*, and it has

$$(\{\langle\rangle\}, \{(\langle\rangle, X) \mid X \subseteq A^{\checkmark}\}) \in \mathbb{P}(A^{*\checkmark}) \times \mathbb{P}(A^{*\checkmark} \times \mathbb{P}(A^{\checkmark}))$$

as its denotation in $\mathcal{F}$, i.e., the process *STOP* can perform only the empty trace, and after the empty trace the process *STOP* can refuse to engage in all events. In CSP, a process P is considered to be deadlock free, if the process P after performing a trace s never becomes equivalent to the process *STOP*.

DEFINITION 2.2.4 *A process P is* **deadlock-free** *in* CSP *iff*

$$\forall s \in \mathcal{A}^*.(s, \mathcal{A}^{\checkmark}) \notin failures(P).$$

This definition is justified, as in the model $\mathcal{F}$ the set of stable failures is required to be closed under the subset-relation: $(s, X) \in failures(P) \wedge Y \subseteq X \Rightarrow (s, Y) \in failures(P)$. In other words: Before termination, the process P can never refuse all events; there is always some event that P can perform. Moreover, the stable failure refinement notion preserves the deadlock-freeness of a process. That is, if P is deadlock free and $P \sqsubseteq_{\mathcal{F}} Q$, then Q is deadlock free.

THEOREM 2.2.5 *Let P and Q be processes such that P is deadlock free and $P \sqsubseteq_{\mathcal{F}} Q$. Then Q is deadlock free.*

PROOF. Let $P \sqsubseteq_{\mathcal{F}} Q$ and let Q be a deadlocked process. We show that also P is a deadlocked process. Let Q have a deadlock, i.e., there exists $s \in \mathcal{A}^*$ with $(s, \mathcal{A}^{*\checkmark}) \in failures(Q)$. From the stable failure refinement ($\sqsubseteq_{\mathcal{F}}$) arguments, we know that $failures(Q) \subseteq failures(P)$. Hence, $(s, \mathcal{A}^{*\checkmark}) \in failures(P)$ and P is a deadlocked process. ∎

Deadlock analysis in CSP has been studied in [Ros98], and an industrial application has been described in [BKS97]. Tools for deadlock analysis are developed in [IRG05, IR].

2.2.3 Failures divergences model – $\mathcal{N}$

This model has long been taken as the '*standard*' model for CSP. Here, the processes are represented by two sets of behaviors: the *failures* and the *divergences*. The divergences of a process are the finite traces on which the process can perform an infinite sequence of internal actions. In this model each process P is modeled by the pair[1]

$$(\mathit{failures}^{\perp}(P), \mathit{divergences}(P))$$

where:

- $\mathit{failures}^{\perp}(P)$ is the set of all stable failures (s, X) (where s is a trace and X is a set of actions that the process can refuse in some stable state after s (unable to perform τ or $\checkmark$), or results from state after s which can perform $\checkmark$ and $X \subseteq \mathcal{A}$), together with all the pairs of the form (s, X) for $s \in \mathit{divergences}(P)$.
- $\mathit{divergences}(P)$ is the (extension-closed) set of traces s on which a process can diverge.

In such model, if s is a trace that process P can perform then either P diverges after s or reaches a stable state or one that can perform $\checkmark$.

Formally the failures/divergences model $\mathcal{N}$ is defined to be the pairs $(F^{\perp}, D)$ satisfying the following healthiness condition (where s, t range over $\mathcal{A}^{*\checkmark}$ and X, Y over $\mathbb{P}(\mathcal{A}^{\checkmark})$):

F.1 $\mathit{traces}^{\perp}(P) = \{t \mid (t, X) \in F\}$ is non-empty and prefix closed.

F.2 $\forall s, X : (s, X) \in F$ and $Y \subseteq X$ then $(s, Y) \in F$.

F.3 $\forall s, X : (s, X) \in F$ and $\forall a \in Y : s \frown \langle a \rangle \notin \mathit{traces}^{\perp}(P)$ implies $(s, X \cup Y) \in F$.

F.4 $\forall s : s \frown \langle \checkmark \rangle \in \mathit{traces}^{\perp}(P)$ then $(s, \mathcal{A}) \in F$.

D.1 $\forall s : s \in D \cap \mathcal{A}^*$ and $t \in \mathcal{A}^{*\checkmark}$ then $s \frown t \in D$.

D.2 $\forall s : s \in D$ then $(s, X) \in F$.
This adds all divergences-related failures of F.

D.3 $\forall s : s \frown \langle \checkmark \rangle \in D$ then $s \in D$.
This ensures that we don't distinguish between how processes behaves after successful termination.

Figure 2.5 illustrates the clauses to determine the divergences of some processes.

[1] In the standard CSP literature the failure set in the failures/divergences model is denoted as $\mathit{failures}_{\perp}(P)$. However, in this thesis will be denoted as $\mathit{failures}^{\perp}(P)$. This is to avoid, later in the thesis, confusion with notation out of the context in algebraic specification.

$$
\begin{array}{lcl}
\mathit{divergences}(STOP) & = & \emptyset \\
\mathit{divergences}(SKIP) & = & \emptyset \\
\mathit{divergences}(DIV) & = & A^{*\checkmark} \\
\mathit{divergences}(a \rightarrow P) & = & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{divergences}(P)\} \\
\mathit{divergences}(?x : X \rightarrow P) & = & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{divergences}(P[a/x]), a \in X\} \\
\mathit{divergences}(P \fatsemi Q) & = & \mathit{divergences}(P) \\
 & & \bigcup \{s \frown t \mid s \frown \langle\checkmark\rangle \in \mathit{traces}^{\perp}(P), t \in \mathit{divergences}(Q)\} \\
\mathit{divergences}(P \,\Box\, Q) & = & \mathit{divergences}(P) \cup \mathit{divergences}(Q). \\
\mathit{divergences}(P \sqcap Q) & = & \mathit{divergences}(P) \cup \mathit{divergences}(Q). \\
\mathit{divergences}(P \,|[\,X\,]|\, Q) & = & \{u \frown v \mid \exists\, s \in \mathit{traces}^{\perp}(P), t \in \mathit{traces}^{\perp}(Q). \\
 & & \quad u \in (s \,|[\,X\,]|\, t) \cap \mathcal{A}^* \\
 & & \quad (s \in \mathit{divergences}(P) \vee t \in \mathit{divergences}(Q))\} \\
\mathit{divergences}(P \setminus X) & = & \{(s \setminus X) \frown t \mid s \in \mathit{divergences}(P)\} \\
 & & \bigcup \{(u \setminus X) \frown t \mid u \in \mathcal{A}^w \wedge (u \setminus X) \text{ finite} \\
 & & \quad \wedge\, \forall s < u \,.\, s \in \mathit{traces}^{\perp}(P)\} \\
\mathit{divergences}(P[[R]]) & = & \{s' \frown t \mid \exists\, s \in \mathit{divergences}(P) \cap \mathcal{A}^* \,.\, sR^*t\} \\
\mathit{divergences}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) & = & \begin{cases} \mathit{divergences}(P); & \varphi \text{ evaluates to } \mathit{true} \\ \mathit{divergences}(Q); & \varphi \text{ evaluates to } \mathit{false} \end{cases}
\end{array}
$$

Figure 2.5: Semantic clauses for the failures/divergences model $\mathcal{N}$.

DEFINITION 2.2.6 (FAILURES/DIVERGENCES REFINEMENT) *Let P, Q be arbitrary* CSP *processes. P is a failures-divergences refinement of Q written as $Q \sqsubseteq_{\mathcal{N}} P$ if and only if: $\mathit{failures}^{\perp}(P) \subseteq \mathit{failures}^{\perp}(Q) \;\wedge\; \mathit{divergences}(P) \subseteq \mathit{divergences}(Q)$.*

2.2.3.1 Livelock analysis in CSP

A process is said to *diverge* or *livelock* if it reaches a state from which it may forever compute internally through an infinite sequence of invisible actions. This is clearly a highly undesirable feature of the process, described by some as "even worse than deadlock" [Hoa85]. Livelock may invalidate certain analysis methodologies, and is often caused by a bug in the modeling. However the possibility of writing down a divergent process arises from the presence of two crucial constructs: *hiding* and *ill-formed recursive* processes. *Hiding* turns a visible action into an invisible one. For example, let us consider the process $P = a \rightarrow P$, which performs an infinite stream of a's. If one now conceals the event 'a' in this process

$$P = (a \rightarrow P) \setminus \{a\}$$

it no longer becomes possible to observe any behaviour of this process.

The CSP process *DIV* represents this phenomenon: immediately, it can refuse every event, and it diverges after any trace. *DIV* is the least refined process in the $\sqsubseteq_{\mathcal{N}}$ model. The process *DIV* has

$$(\mathcal{A}^{*\checkmark} \times \mathbb{P}(\mathcal{A}^{\checkmark}), \mathcal{A}^{*\checkmark}) \in \mathbb{P}(\mathcal{A}^{*\checkmark} \times \mathbb{P}(\mathcal{A}^{\checkmark})) \times \mathbb{P}(\mathcal{A}^{*\checkmark})$$

as its semantics over $\mathcal{N}$. Then, a process is said to be free of divergence (or livelock free) if after carrying out a sequence of events, its denotation is different from *DIV*.

DEFINITION 2.2.7 *A process P is said to be livelock free iff one of the following conditions holds:*

C1. $\forall s \in \mathcal{A}^*.\{(t, X) \mid (s \frown t, X) \in \mathit{failures}(P)\} \neq \mathcal{A}^{*\checkmark} \times \mathcal{P}(\mathcal{A}^{\checkmark})$

C2. $\forall s \in \mathcal{A}^*.\{t \mid (s \frown t) \in \mathit{divergences}(P)\} \neq \mathcal{A}^{*\checkmark}$.

The failures/divergences refinement notion preserves the livelock freeness of a process. That is, if P is livelock free and $P \sqsubseteq_{\mathcal{N}} Q$, then Q is livelock free.

THEOREM 2.2.8 *Let P and Q be processes such that P is livelock free and $P \sqsubseteq_N Q$. Then Q is livelock free.*

PROOF. Let $P \sqsubseteq_{\mathcal{N}} Q$ and Q be a livelocked process. We show that also P is a livelocked process. Let Q have a livelock due to **C.2**, i.e., for all $s \in \mathcal{A}^*$ with $\{t \mid (s \frown t) \in \mathit{divergences}(P)\} = \mathcal{A}^{*\checkmark}$. From the failures/divergences refinement ($\sqsubseteq_{\mathcal{N}}$) definition, we know that $\mathit{divergences}(Q) \subseteq \mathit{divergences}(P)$. Hence, $\{t \mid (s \frown t) \in \mathit{divergences}(P)\} = \mathcal{A}^{*\checkmark}$ and P is a livelocked process. We can draw the same conclusion following the failures argument, w.r.t., **C.1** of Definition 2.2.7. ∎

Livelock analysis in CSP has been applied to an industrial application in [SPK99].

2.2.4 Analysing CSP recursion

In this section we briefly present how the semantics of recursive processes is defined in CSP. The semantics of recursive processes in CSP is determined in terms of fixed points. CSP offers two standard approaches to deal with fixed points: *complete partial orders* (cpo) or *complete metric spaces* (cms). These two approaches follow a similar pattern: the first step consists of proving that the domain of a given CSP model is a *cms* or a *cpo*, respectively. As a particularity of CSP, metric spaces are introduced in terms of so-called restriction spaces. The second step consists of proving that the various CSP operators satisfy the pre-requisite properties, namely *contractiveness* for *cms* and *continuity* for *cpo*.

In the case of *cms*, Banach's fixed point theorem is employed, while for the *cpo* approach Tarski's fixed point theorem is used. Banach's theorem leads to a unique fixed point, while Tarski's theorem does not guarantee uniqueness. Here, the least fixed point is chosen in the CSP models $\mathcal{T}$ and $\mathcal{N}$, while the largest fixed point is chosen for the model $\mathcal{F}$.

In order to prove properties of a recursive process in CSP, for example that Q *refines* P, both the *cms* and the *cpo* approach offer as a technique the so-called *fixed point induction.*

In the CSP literature recursive processes are written in an equational style, for example:

$$P_1 = a \rightarrow b \rightarrow P_2$$
$$P_2 = c \rightarrow d \rightarrow P_2$$

The Tarski fixed point theorem guarantees that a solution exists to the equations $X = F(X)$, for X an element in some domain $\mathcal{D}$.

Let $P = exp(P)$ be a process equation in CSP. CSP defines for each equation exp an underlying function f_{exp} on the domain. Given f_{exp}, in the traces domain the semantics of the above process equation is defined as

$$traces(P = exp(P)) = \bigcup_{n \in \mathbb{N}} \{f_{exp}^n(\bot) \mid n \geqslant 0\}$$

As all CSP operators are continuous in the traces domain, such a solution always exists. This means that the trace set of P is determined by iteratively applying the function f_{exp} starting with the minimal element $\bot$ of $\mathcal{T}$. As illustrated previously, in the case of the traces model the minimal element is identified by the process *STOP*.

For example, let us consider the trace set of the process $Beep = beep \rightarrow Beep$. The induced semantic function $F(Beep)$ is $\{\langle\rangle \cup \{\langle beep\rangle \frown t \mid t \in Beep\}$. The successive iterations $F(Beep)$ yields:

$$\begin{array}{lcl} F^0(\bot) & = & \bot = \{\langle\rangle\} \\ F^1(\bot) & = & F(F^0(\bot)) = \{\langle beep\rangle, \langle\rangle\} \\ F^2(\bot) & = & F(F^1(\bot)) = \{\langle beep, beep\rangle, \langle beep\rangle, \langle\rangle\} \\ \vdots & & \end{array}$$

Taking the union of all the values yields the trace semantic of *Beep*; thus, $traces(Beep) = \{tr \mid tr \leq beep^*\}$, i.e., the prefixed closed set of traces. Here, $beep^*$ is a regular expression where * denotes Kleene's star.

2.2.5 Tools for CSP

CSP's practical success is founded on well developed tool support. Here we describe some successful tools developed for CSP. In order to support some of these tools, a machine readable version of CSP (CSP-M) is introduced, this is described in detailed in [Ros98, Sca98].

Data in CSP-M is defined using a purely functional programming language with a strong static type system, requiring explicit type declarations for channels and data types. For a concrete example, let us consider the following specification taken from the FDR2 distribution (`www.fsel.com/software.html`).

```
-- First, the set of values to be communicated
 datatype FRUIT = apples | oranges | pears

-- Channel declarations
 channel left,right,mid : FRUIT
 channel ack
```

```
-- The specification is simply a single place buffer
 COPY = left ? x -> right ! x -> COPY

-- The implementation consists of two processes
 SEND = left ? x -> mid ! x -> ack -> SEND
 REC = mid ? x -> right ! x -> ack -> REC

-- These components are composed in parallel
-- and the internal communication is hidden
 SYSTEM = (SEND [| {| mid, ack |} |] REC) \ {| mid, ack |}

-- Checking that "SYSTEM" is correct implementation of "COPY"
 assert COPY [FD= SYSTEM

-- In fact, the processes are equal, as shown by
 assert SYSTEM [FD= COPY
```

This example specifies a single-place buffer implemented over two channels, `left` and `right`. It includes two specifications of such a buffer, and asserts that they are equivalent. First an enumerated type is defined: `FRUIT`. Channels `left`, `right` and `mid` communicate values of type `FRUIT`. The channel `ack` is singleton-typed. The abstract specification `COPY`, states that a buffer's behaviour is to repeatedly read some value on channel `left`, bind it to the local variable x, and then write it out on the channel `right`. Then `SYSTEM` is a concrete specification; here the receiving and sending parts are separate processes, i.e., `SEND` and `REC`. Those processes they synchronise on the channel `mid` and uses the channel `ack` to proceed in lockstep. The channels `mid` and `ack` are then hidden from everything outside the process `SYSTEM`. Finally, we assert that over the failures/divergences model ($[FD =$), the processes `COPY` and `SYSTEM` are equivalent.

We now list some successful tools developed for CSP:

FDR2 [Ltd06] is a model checking tool developed by Formal Systems (Europe). Specifically, it allows us to check whether or not a process refines another, in each of the three semantic models. It also performs checks for determinism, deadlock-freedom and divergence-freedom. The tool takes as input a text file containing process descriptions written in CSP-M.

PROBE [Ltd03] is an animator developed by Formal Systems (Europe). PROBE allows you to load a CSP *script file*, then simulate some arbitrary CSP process description and interact with it. In some sense, this tool allows the user to act as the environment and to choose how much control he wants over the process.

CSP-PROVER [IR05, IRG05, IR06, IR07, IR08] is an interactive theorem prover built upon Isabelle/HOL [NPW02] . CSP-PROVER is dedicated to refinement proofs within the process algebra CSP. It is generic in the models of CSP that can be used. Currently the trace model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$ are available. CSP-PROVER pro-

vides a deep-encoding of CSP within Isabelle/HOL; this means that the syntax and semantics of CSP processes have been encoded within the logical framework of Isabelle/HOL. CSP-PROVER supports three methods for allowing the user to prove process refinements, namely syntactical, semantical and semi-automatic proofs.

(CHAPTER ...3)

Common Algebraic Specification Language (CASL)

Contents

CASL stands for *Common Algebraic Specification Language*. It's a specification language designed by the *Common Framework Initiative for algebraic specification and development* (CoFI). In this chapter we describe the main features of CASL following [BM04, Mos04]. Section 3.1 describes the fundamental concepts of CASL, while in Section 3.2 we describe the institutional framework of CASL. Here, we illustrate some institutions of CASL. Finally, in Section 3.3 we present some notions of refinement for CASL.

3.1 CASL – fundamental concepts

The aim of the CoFI project was to design a *Common Framework for Algebraic Specification and Development* — an attempt to create a *de facto* standard framework for algebraic specification, providing a family of languages which are coherent, and are extensions or restrictions of some main algebraic specification language. This was motivated by the existence of a number of competing algebraic specification languages developed over the years, with varying levels of tool support and industrial uptake; major examples include *ACT-ONE/TWO* [EM85, CEW93], *OBJ* [GWM$^+$93] and functional *CafeOBJ* [DF98], *Extended ML* [KST97], *Larch* [GH93], *ASF* [BHK89], *ASF-SDF* [Kli93, DHK96] and *Maude* [MFS$^+$07].

3.1.1 Overview of CASL

CASL provides basic specifications consisting of many-sorted signatures with subsorting, partial functions, sort generation constraints and axioms written in first order logic with equality.

CASL consists of several layers, including basic (unstructured) specifications, structured specifications and architectural specifications. We give a brief overview of the constructs for writing basic specifications in CASL. A detailed description of the CASL language and its semantics can be found in the "CASL *Reference Manual*" [Mos04].

A CASL *basic specification* consists of a set of declarations of symbols (i.e., names) for *sorts* (the data types of the specification), symbols and profiles for *operations* (total and partial functions on the sorts), symbols and profiles for *predicates* (relations on the sorts), and a set of *axioms* and *constraints* which restrict the interpretations of the declared symbols. As noted below, such specifications may be *named* for reference in structured specifications and named specifications may be *generic*, i.e., include parameterised elements. The axioms are formulas in two–valued first order logic with equality ($FOL^{=}$), with the usual connectives and universal quantifiers; furthermore, they may make assertions regarding definedness (e.g. of the results of partial functions) and subsorting.

CASL includes *sort generation constraints* to further control the contents of the Σ-algebras' carrier sets. The models of a *loose* specification (the default interpretation) include all those with the properties defined by the specification's axioms, without further restraint on the carrier sets (so for example trivial carriers such as singletons will tend to be included); CASL also allows for *generated* and *free* datatypes. If a sort is declared as a *generated* datatype, values of the sort are built only using the sort's provided constructors ('no junk' – this provides an induction proof principle on such types); if a sort is declared as a *free* datatype, it is generated and furthermore, values denoted by different constructor terms are necessarily distinct ('no confusion').

We now illustrate some of the concepts mentioned above through the following example: *monoid* specification in CASL.

spec MONOID =
  sort *Element*
  ops n : *Element*; @: *Element* $\times$ *Element* $\rightarrow$ *Element*;
  axioms
    vars x, y, z : *Element*
    . $n@x = x$ %(left unit 1)%
    . $x@n = x$ %(right unit 1)%
    . $(x@y)@z = x@(y@z)$ %(associativity)%
end

This example illustrates: a sort name declaration, two operations, one of which is 0-ary and hence defines a *constant*, and three axioms on the items declared. The semantics

of such a specification consists of a *a many-sorted signature* $\Sigma(\text{MONOID})$, and a class of $\Sigma(\text{MONOID})$-*algebras* corresponding to interpretations of the signature in which the axioms and constraints are satisfied. The Σ-algebra contains *carrier sets* corresponding to Σ's sorts, and *functions* and *relations* corresponding to Σ's operations and predicates. In the monoid example, the class of Σ-algebras forming this specification model includes the natural numbers under multiplication with an identity of 1, the natural numbers under addition with an identity of 0, and lists under concatenation with an identity of the empty list.

A basic CASL specification may include declarations of *subsorts*. Subsorts are interpreted by arbitrary *embeddings* (1-1 functions) between the sorts' corresponding carrier sets. In the following example, we illustrate a subsort declaration in CASL.

spec VEHICLE =
  free type $Nat ::= 0 \mid suc(Nat)$
  sorts $Car, Bicycle < Vehicle$
  ops $speed : Vehicle \to Nat$;
    $engine : Car \to Nat$
end

The above example introduces three sorts, *Car*, *Bicycle* and *Vehicle*. It declares both *Car* and *Bicycle* to be subsorts of *Vehicle*. A subsort declaration entails that any term of a subsort is also a term of the supersort, so here, any term of sort *Car* is also a term of sort *Vehicle*. Here we can apply the operation *speed* to it. But we can also define operations on the subsorts, for instance the operation *engine* can only be applied to *Car*.

Structured CASL specifications These are specifications which are built on basic specifications by allowing them to be combined into larger and more complex specifications with the same underlying semantics. Specifically, a structured specification can combine basic specifications, references to *named* specifications, and instances of *generic* (i.e., parameterised) specifications, using constructs of *extension*, *union*, *hiding* and *renaming*. Like a basic specification, the semantics of a structured specification consists of a signature Σ and a class of Σ-algebras. Structured specifications allow us to build complex specifications out of simpler ones; *architectural specifications* explicitly describe the intended structure/composition of a system in terms of its components; they target the *design* phase of development, in which the decomposition of a system into components/modules for further development and implementation in target languages is considered. We do not consider architectural specifications any further in this work.

CASL Libraries These are "named collections of named specifications", and provide the highest-level organizational mechanism in CASL, collecting specifications into libraries identified by name and version number, with the aim of promoting reuse of specifications.

3.2 CASL – the institutional framework

CASL integrates subsorts, partiality, first-order logic and induction (known also as sort generation constraints). In this section, we present the institutional framework of many-sorted partial first-order logic with sort generation constraints and equality *PCFOL*$^{=}$, and *SubPCFOL*$^{=}$ which adds subsorting. We first give a general description of *institutions*.

Institutions capture the nature of logic systems and are the language used to define CASL and CSP-CASL. Institution were introduced by Joseph Goguen and Rod Burstall in the late 1970's in order to deal with the large volume of logical systems being used to developed in various subjects in computer science. The use of institutions allows to create specification languages, proof calculus and tools which are independent of the underlying logical system. The institutional framework allows one to relate and translate institutions with other institutions.

Informally, an institution consists of a collection of signatures with signature morphisms and for each signature a collection of sentences, models and a satisfaction relation between the sentences and models such that the satisfaction condition holds. The satisfaction condition ensures that if one translates a sentence under a signature with a signature morphism, then the satisfaction of a translated sentence and a model is preserved.

The formal definition of institutions rely on category theory. An institution I as defined by Mossakowski in [Mos02], is a quadruple $(\mathbf{SIGN}, \mathbf{sen}, \mathbf{mod}, \models)$ where:

- $\mathbf{SIGN}$ is a category.
- $\mathbf{sen} : \mathbf{SIGN} \to \mathbf{SET}$ is a functor.
- $\mathbf{mod} : (\mathbf{SIGN})^{op} \to \mathbf{CAT}$ is a functor.
- $\models_\Sigma \subseteq |\mathbf{mod}(\Sigma)| \times \mathbf{sen}(\Sigma)$, for each $\Sigma : \mathbf{SIGN}$,

such that the satisfaction condition holds: for every $\sigma : \Sigma \to \Sigma'$ in $\mathbf{SIGN}$,

$$\mathbf{mod}(\sigma)(M') \models_\Sigma \varphi \Leftrightarrow M' \models_{\Sigma'} \mathbf{sen}(\sigma)(\varphi)$$

holds for every $\varphi \in \mathbf{sen}(\Sigma)$ and for every $M' \in |\mathbf{mod}(\Sigma')|$.

The category $\mathbf{SIGN}$ denotes the collection of signatures and signature morphism which map symbols in a compatible way.

The functor $\mathbf{sen} : \mathbf{SIGN} \to \mathbf{SET}$ gives for each signature $\Sigma : \mathbf{SIGN}$, the set of sentences $\mathbf{sen}(\Sigma)$ over the signature Σ, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the map $\mathbf{sen}(\sigma) : \mathbf{sen}(\Sigma) \to \mathbf{sen}(\Sigma')$ which translates sentences built over Σ to sentences built over Σ'.

The functor $\mathbf{mod} : (\mathbf{SIGN})^{op} \to \mathbf{CAT}$ gives for each signature $\Sigma : \mathbf{SIGN}$, the category of models for that signature $\mathbf{mod}(\Sigma)$, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the reduct functor $\mathbf{mod}(\sigma) : \mathbf{mod}(\Sigma') \to \mathbf{mod}(\Sigma)$ which reduces models over the signature

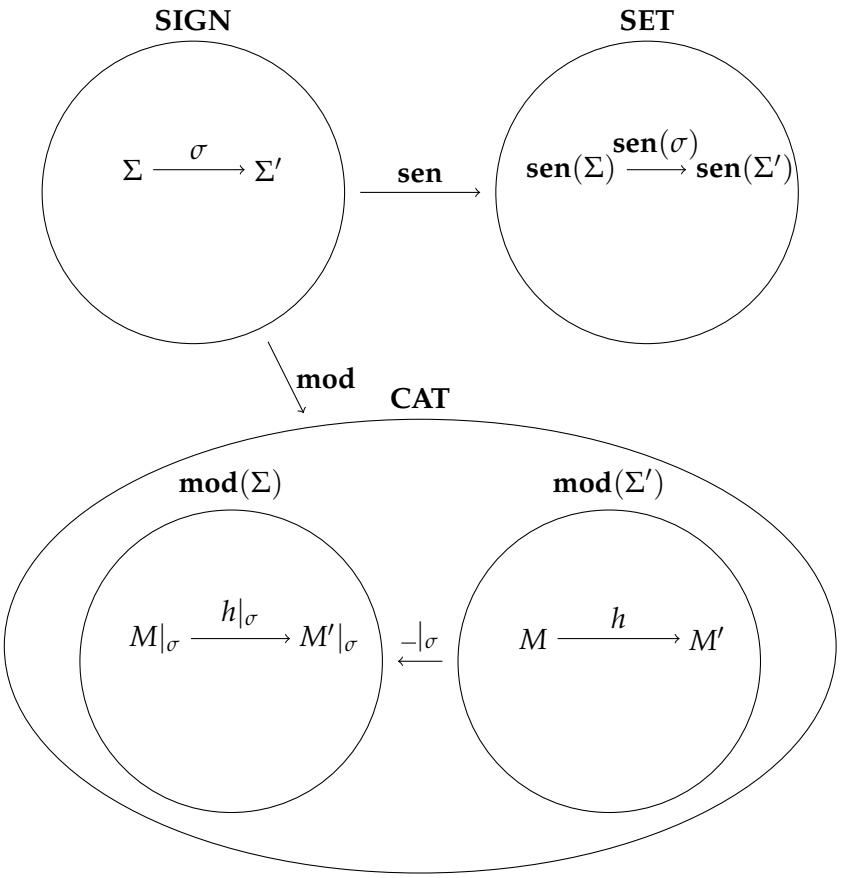

Figure 3.1: The institutional framework.

Σ' to models over the signature Σ. Figure 3.1 illustrates the overall framework of an institution.

Some basic shorthand notations that are often used when dealing with institutions: we write $\sigma(\varphi)$ for $\mathbf{sen}(\sigma)(\varphi)$, and $M' \mid_\sigma$ for $\mathbf{mod}(\sigma)(M')$. Hence, for each $\sigma : \Sigma \to \Sigma'$ in **SIGN**, the satisfaction condition becomes

$$M'|_\sigma \models_\Sigma \varphi \Leftrightarrow M' \models_{\Sigma'} \sigma(\varphi)$$

for each $M' \in |\mathbf{mod}(\Sigma')|$ and $\varphi \in \mathbf{sen}(\Sigma)$.

Given an arbitrary fixed institution, we can define the usual notion of logical consequence or semantical entailment. Given a set of Σ-sentences $\Gamma \subseteq \mathbf{sen}(\Sigma)$ and a Σ-sentences $\varphi \in \mathbf{sen}(\Sigma)$, we say

$$\Gamma \models_\Sigma \varphi \text{ iff for all } \Sigma \text{ models } M \in |\mathbf{mod}(\Sigma)|, M \models_\Sigma \Gamma \text{ implies } M \models_\Sigma \varphi$$

where $M \models_\Sigma \Gamma$ means $M \models_\Sigma \psi$ for every $\psi \in \Gamma$.

3.2.1 The institution $PCFOL^=$

Here, we present the institution of the many-sorted partial first-order logic with sort generation constraint and equality $PCFOL^=$.

Signatures A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of

- a set S of sorts,
- two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of *total function symbols* and *partial function symbols*, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$ for each $(w, s) \in S^* \times S$, and
- a family $P = (P_w)_{w \in S^*}$ of *predicate symbols*.

Given a function $f : A \to B$, let $f^* : A^* \to B^*$ be its component-wise extension to finite strings. Given two signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, a *many-sorted signature morphism* $\sigma : \Sigma \to \Sigma'$ consists of

- a map $\sigma^S : S \to S'$,
- a map $\sigma^F_{w,s} : TF_{w,s} \cup PF_{w,s} \to TF'_{\sigma^{S^*}(w), \sigma^S(s)} \cup PF'_{\sigma^{S^*}(w), \sigma^S(s)}$ preserving totality, for each $w \in S^*, s \in S$, and
- a map $\sigma^P : P_w \to P_{\sigma^{S^*}(w)}$.

Models Given a many-sorted signature $\Sigma = (S, TF, PF, P)$, a *many-sorted* Σ*-model* M consists of

- a non-empty carrier set M_s for each $s \in S$,
- a partial function $(f_{w,s})_M : M_w \to M_s$ for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, the function being total for $f \in TF_{w,s}$, and
- a relation $(p_w)_M \subseteq M_w$ for each predicate symbol $p \in P_w$.

A *many-sorted* Σ*-homomorphism* $h : M \to N$ is a family of functions $h = (h_s : M_s \to N_s)_{s \in S}$ with the property that for all $f \in TF_{w,s} \cup PF_{w,s}$ and $(a_1, \ldots, a_n) \in M_w$ with $(f_{w,s})_M(a_1, \ldots, a_n)$ defined, we have

$$h_s((f_{w,s})_M(a_1, \ldots, a_n)) = (f_{w,s})_N(h_{s_1}(a_1), \ldots, h_{s_1}(a_n)),$$

and for all $p \in P_w$ and $(a_1, \ldots, a_n) \in M_w$,

$$(a_1, \ldots, a_n) \in (p_w)_M \text{ implies } (h_{s_1}(a_1), \ldots, h_{s_1}(a_n)) \in (p_w)_N \,.$$

Let $\sigma : \Sigma \to \Sigma'$ be a a many-sorted signature morphism, M' be a Σ'-model. Then the *reduct* $M' \mid_\sigma =: M$ of M' is the Σ-model with

- $M_s := M'_{\sigma^S(s)}$ for all $s \in S$,

- $(f_{w,s})_M := (\sigma^F_{w,s}(f))_{M'}$ for all $f \in TF_{w,s} \cup PF_{w,s}$, and
- $(p_w)_M := (\sigma^P_w(p))_{M'}$ for all $p \in P_w$.

Given a many-sorted Σ'-homomorphism $h' : M' \to N'$, its *reduct* $h' \mid_\sigma : M' \mid_\sigma \to N' \mid_\sigma$ is defined by $(h' \mid_\sigma)_s := h'_{\sigma^S(s)}$ for all $s \in S$.

Sentences Given a many-sorted signature $\Sigma = (S, TF, PF, P)$, a *variable system* over Σ is an S-sorted, pairwise disjoint family of variables $X = (X_s)_{s \in S}$. The sets $T_\Sigma(X)_s$ of *many-sorted* Σ*-terms* of sort s, $s \in S$, with variables in X are the least sets satisfying

- $x \in T_\Sigma(X)_s$, if $x \in X_s$, and
- $f_{w,s}(t_1, \ldots, t_n) \in T_\Sigma(X)_s$,
  if $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1 \ldots n)$, $f \in TF_{w,s} \cup PF_{w,s}$, $w = s_1 \ldots s_n$.

Given a *total variable valuation* $\nu : X \to M$, the *term evaluation* $\nu^\sharp : T_\Sigma(X) \to ?M$ is inductively defined by

- $\nu^\sharp_s(x) := \nu(x)$ for all $x \in X_s$ and all $s \in S$.
- $\nu^\sharp_s(f_{w,s}(t_1, \ldots, t_n)) :=$
  $$\begin{cases} (f_{w,s})_M(\nu^\sharp_{s_1}(t_1), \ldots, \nu^\sharp_{s_n}(t_n)) & \text{if } \nu^\sharp_s(t_i) \text{ defined } (i = 1 \ldots n) \text{ and} \\ & \quad (f_{w,s})_M(\nu^\sharp_{s_1}(t_1), \ldots, \nu^\sharp_{s_n}(t_n)) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$
  for all $f \in TF_{w,s} \cup PF_{w,s}$ and $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1 \ldots n)$, where $w = s_1 \ldots s_n$. Note that $\nu^\sharp$ can be undefined. That's the reason why we add '?' in its profile.

The set $AF_\Sigma(X)$ of *many-sorted atomic* Σ*-formulae with variables in* X is the least set satisfying the following rules:

1. $p_w(t_1, \ldots t_n) \in AF_\Sigma(X)$, if $t_i \in T_\Sigma(X)_{s_i}$, $p_w \in P_w$, $w = s_1 \ldots s_n \in S^*$,
2. $t_1 \stackrel{e}{=} t_2 \in AF_\Sigma(X)$, if $t_1, t_2 \in T_\Sigma(X)_s, s \in S$ (existential equations),
3. $t_1 = t_2 \in AF_\Sigma(X)$ if $t_1, t_2 \in T_\Sigma(X)_s, s \in S$ (strong equations),
4. $def\, t \in AF_\Sigma(X)$, if $t \in T_\Sigma(X)$ (definedness assertions),

The set $FO_\Sigma(X)$ of *many-sorted first-order* Σ*-formulae with variables in* X is the least set satisfying the following rules:

1. $AF_\Sigma(X) \subseteq FO_\Sigma(X)$,
2. $F \in FO_\Sigma(X)$ (read: false),
3. $\varphi \wedge \psi \in FO_\Sigma(X)$, if $\varphi, \psi \in FO_\Sigma(X)$,
4. $\varphi \Rightarrow \psi \in FO_\Sigma(X)$, if $\varphi, \psi \in FO_\Sigma(X)$,
5. $\forall x : s \bullet \varphi \in FO_\Sigma(X)$, if $\varphi \in FO_\Sigma(X \cup \{x : s\}), s \in S$,

A *many-sorted* Σ*-sentence* is a closed many-sorted first order formula over Σ. Concerning the definition of the translation of many-sorted Σ sentences along a many-sorted Σ-morphism we refer to [Mos02].

Satisfaction The satisfaction of a many sorted first-order formula $\varphi \in FO_\Sigma(X)$ relative to a valuation $\nu : X \to M$ is defined inductively over the structure of φ :

- $\nu \Vdash p_w(t_1, \ldots t_n)$ if and only if $\nu^\sharp(t_i)$ is defined $(i = 1 \ldots n)$ and $(\nu^\sharp(t_1), \ldots, \nu^\sharp(t_n)) \in (p_w)_M$.
- $\nu \Vdash t_1 \stackrel{e}{=} t_2$ if and only if $\nu^\sharp(t_1)$ and $\nu^\sharp(t_2)$ are both defined and equal.
- $\nu \Vdash t_1{=}t_2$ iff $\nu^\sharp(t_1)$ and $\nu^\sharp(t_2)$ are either both undefined, or both are defined and equal.
- $\nu \Vdash def\, t$ if and only if $\nu^\sharp(t)$ is defined.
- not $\nu \Vdash F$.
- $\nu \Vdash \varphi \wedge \psi$ if and only if $\nu \Vdash \varphi$ and $\nu \Vdash \psi$.
- $\nu \Vdash \varphi \Rightarrow \psi$ if and only if $\nu \Vdash \varphi$ implies $\nu \Vdash \psi$.
- $\nu \Vdash \forall\, x : s \bullet \varphi$ if and only if for all valuations $\zeta : X \cup \{x : s\} \to M$ with $\zeta(y) = \nu(y)$ for $y \neq x : s$, $y \in X$, we have $\zeta \Vdash \varphi$.

$M \models \varphi$ holds for a many-sorted Σ-model and a many-sorted formula φ, iff $\nu \Vdash \varphi$ for all variable valuations ν into M. [Mos02] proves the satisfaction condition of $PCFOL^=$.

3.2.2 The institution *SubPCFOL*$^=$

Signatures A *subsorted signature* $\Sigma = (S, TF, PF, P, \leq)$ consists of a many-sorted signature (S, TF, PF, P) together with a reflexive and transitive *subsort relation* $\leq_S \subseteq S \times S$. The relation $\leq_S$ extends point-wise to sequences of sorts. We drop the subscript S when it is obvious from the context.

For a *subsorted signature* $\Sigma = (S, TF, PF, P, \leq)$ we define *overloading relations* $\sim_F$ and $\sim_P$ for function and predicate symbols, respectively. Let $f : w_1 \to s_1, f : w_2 \to s_2 \in TF \cup PF$. Then $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ if and only if there exist $w \in S^*, s \in S$ such that $w \leq w_1, w \leq w_2, s_1 \leq s$, and $s_2 \leq s$. Let $p : w_1, p : w_2 \in P$. Then $p : w_1 \sim_P p : w_2$ if and only if there exists $w \in S^*$ such that $w \leq w_1$ and $w \leq w_2$.

A *subsorted signature morphism* $\sigma : \Sigma \to \Sigma'$ is a many-sorted signature morphism that preserves the subsort relation and the overloading relations[1], i.e., for σ holds:

p1 $s_1 \leq s_2$ implies $\sigma^S(s_1) \leq \sigma^S(s_2)$ for all $s_1, s_2 \in S$

[1]Note that, thanks to preservation of subsorting, the preservation of the overloading relations can be simplified.

p2 $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ implies $\sigma^F_{w_1,s_1}(f) = \sigma^F_{w_2,s_2}(f)$
for all $f \in TF \cup PF$

p3 $p : w_1 \sim_P p : w_2$ implies $\sigma^P_{w_1}(p) = \sigma^P_{w_2}(p)$ for all $p \in P$

With each subsorted signature $\Sigma = (S, TF, PF, P, \leq)$ we associate a many-sorted signature $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$, which extends the underlying many-sorted signature (S, TF, PF, P) with

- a total *injection* function symbol $\mathtt{inj} : s \to s'$ for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\mathtt{pr} : s' \to ?s$ for each pair of sorts $s \leq_S s'$, and
- an unary *membership* predicate symbol $\epsilon^s_{s'} : s'$ for each pair of sorts $s \leq_S s'$.

Given a subsorted signature morphism $\sigma : \Sigma \to \Sigma'$, we can extend it to a many-sorted signature morphism $\hat{\sigma} : \hat{\Sigma} \to \hat{\Sigma}'$ by just mapping the injections, projections and memberships in $\hat{\Sigma}$ to the corresponding injections, projections and memberships in $\hat{\Sigma}'$.

Models *Subsorted* Σ*-models* are many-sorted $\hat{\Sigma}$-models satisfying in *PCFOL*$^=$ the following set of axioms $\hat{J}(\Sigma)$ (all variables are universally quantified):

1. $\mathtt{inj}_{s,s}(x) \stackrel{e}{=} x$ for $s \in S$.
2. $\mathtt{inj}_{s,s'}(x) \stackrel{e}{=} \mathtt{inj}_{s,s'}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq s'$.
3. $\mathtt{inj}_{s',s''}(\mathtt{inj}_{s,s'}(x)) \stackrel{e}{=} \mathtt{inj}_{s,s''}(x)$ for $s \leq s' \leq s''$.
4. $\mathtt{pr}_{s',s}((\mathtt{inj}_{s,s'}(x)) \stackrel{e}{=} x$ for $s \leq s'$.
5. $\mathtt{pr}_{s',s}(x) \stackrel{e}{=} \mathtt{pr}_{s',s}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq s'$.
6. $\epsilon^s_{s'}(x) \Leftrightarrow \mathit{def}\, \mathtt{pr}_{s',s}(x)$ for $s \leq s'$.
7. $\mathtt{inj}_{s',s}(f_{w',s'}(\mathtt{inj}_{s_1,s'_1}(x_1), \ldots, \mathtt{inj}_{s_n,s'_n}(x_n))) =$
$\mathtt{inj}_{s'',s}(f_{w'',s''}(\mathtt{inj}_{s_1,s''_1}(x_1), \ldots, \mathtt{inj}_{s_n,s''_n}(x_n)))$ for $f_{w',s'} \sim_F f_{w'',s''}$,
where $w \leq w', w''$, $w = s_1 \ldots s_n, w' = s'_1 \ldots s'_n, w'' = s''_1 \ldots s''_n$, $s', s'' \leq s$.
8. $p_{w'}(\mathtt{inj}_{s_1,s'_1}(x_1), \ldots, \mathtt{inj}_{s_n,s'_n}(x_n)) \Leftrightarrow$
$p_{w''}(\mathtt{inj}_{s_1,s''_1}(x_1), \ldots, \mathtt{inj}_{s_n,s''_n}(x_n))$ for $p_{w'} \sim_P p_{w''}$,
where $w \leq w', w''$, $w = s_1 \ldots s_n, w' = s'_1 \ldots s'_n, w'' = s''_1 \ldots s''_n$.

Sentences The *Subsorted formulae over* Σ are the many-sorted first order formulae over $\hat{\Sigma}$. A *sub-sorted* Σ*-sentence* is a many-sorted first order sentences over $\hat{\Sigma}$.

Satisfaction The satisfaction relations $\nu \Vdash \varphi$ and $M \models \varphi$ are defined as in *PCFOL*$^=$. Again, for the proof of the satisfaction condition we refer to [Mos02].

3.3 Refinement based on model class inclusion

The standard development paradigm of algebraic specification [EK99] postulates that development begins with a formal *requirement specification* D_0 – extracted from a software project's informal specification. Such specification fixes only expected properties but ideally says nothing about implementation issues. This is to be followed by a number of *refinement* steps that fix more and more details of the design until specification D_n is obtained. D_n is detailed enough that its conversion into a program P is relatively straightforward:

$$D_0 \rightsquigarrow D_1 \rightsquigarrow \ldots \rightsquigarrow D_n$$

In the context of algebraic specification, [EK99] provides an excellent survey on different refinement approaches. In the following paragraph we summarize the main concept highlighted in the survey.

Refinement and Implementation described in [EK99]. Let D_1 and D_2 be two specifications. A *refinement* of D_1 by D_2 is a 5-tuple $(D, \epsilon, \sigma, \kappa, \alpha)$, where:

- D is the intermediate specification,
- $\epsilon : D_2 \rightarrow D$ is an enrichment,
- $\sigma : \Sigma(D_1) \rightarrow \Sigma(D)$ is a signature morphism,
- $\kappa : \mathbf{Mod}(D_2) \rightarrow \mathbf{Mod}(D)$ is a constructor,
- $\alpha : \mathbf{Mod}(D_1) \rightarrow \mathbf{Mod}(\Sigma(D_1))$ is an abstractor.

A refinement of a specification D_1 into a specification D_2 consists of a *constructive* part and an *abstraction* part. The constructive part enriches D_2 to D such that we find the signature of D_1 in D immediately or after some renaming. The abstraction part states which $\Sigma(D_1)$-models are acceptable as realizations of any D_1-model.

A constructor κ is a mapping from D_2-models to D-model classes. An abstractor α is a mapping from D_1 models to $\Sigma(D_1)$ models, that has to satisfy the reflexivity and transitivity conditions:

- $A \in \alpha(A)$ for all $A \in \mathbf{Mod}(D_1)$
- $A_1 \in \alpha(A_2)$ and $A_2 \in \alpha(A_3) \Rightarrow A_1 \in \alpha(A_3)$, for all $A_1 \in \mathbf{Mod}(\Sigma(D_1))$ and $A_2, A_3 \in \mathbf{Mod}(D_1)$

Intuitively the enrichment ϵ brings all the syntactic features that are needed in the refined specification D_2. However, if in D_2 the needed signature is already specified, we can assume D_2 is the intermediate specification D. Constructor and abstractor relate the models of D, D_1 and D_2.

A refinement is said to be *correct* if it is *consistent* and *complete* :

$$\text{(consistent) } \kappa(\mathbf{Mod}(D_2)) \mid_\sigma \subseteq \alpha(\mathbf{Mod}(D_1))$$
$$\text{(complete) } \kappa(\mathbf{Mod}(D_2)) \mid_\sigma \cap \alpha(A) \neq \emptyset \text{ for every } A \in \mathbf{Mod}(D_1)$$

Intuitively a correct refinement step relates possible and acceptable representations in a suitable way. A consistent refinement means that every possible representation is acceptable, and a complete refinement requires that every algebra of the refined specification (implemented specification) is realized by the algebra of the specification to be refined (implementing specification), up to abstraction.

In the literature one finds a vast number of implementation approaches: [Hoa76], [GTW78], [EKP80], [Ehr82] and [ST97]. Mossakowski et.al. in [MST04] present a simple refinement language for CASL.

The primary use of specifications is to describe programs, nevertheless CASL abstracts away from all details of programming languages and programming paradigms. This is common with most work on algebraic specification. Aspinall et.al. in [AS02] studies the relationship between CASL and programming languages. The connection with programs is indirect, via the use of partial first-order structures or similar mathematical models of program behavior. Let $\Sigma(P)$ be the CASL signature of a program P, and let $[\![P]\!] \in Alg(\Sigma(P))$ be the partial first-order structure of the program P, i.e., the semantical denotation of P. Then P is regarded as satisfying a specification Sp if $\Sigma(Sp) = \Sigma(P)$ and $[\![P]\!] \in [\![Sp]\!]$ where $\Sigma(Sp)$ and $[\![Sp]\!] \subseteq Alg(\Sigma(Sp))$ are given by the semantics of CASL.

The simplest form of refinement is just *model class inclusion*. In the following we give some definitions and examples for refinement based on model class inclusion.

DEFINITION 3.3.1 (MODEL CLASS INCLUSION) *Given a signature Σ, and two model classes C_1 and C_2. We say that C_1 refines to C_2, written as $C_1 \rightsquigarrow C_2$ if and only if $C_2 \subseteq C_1$, for $C_1, C_2 \subseteq \boldsymbol{Mod}(\Sigma)$.*

EXAMPLE 3.3.2 As a first example we consider the refinement of the MONOID specification to the specification of a *commutative monoid*.

spec COMMMONOID =
  sort *Element*
  ops n : *Element*; @: *Element* $\times$ *Element* $\rightarrow$ *Element*;
  axioms
    vars x, y, z : *Element*
    . $n@x = x$ %(left unit)%
    . $x@n = x$ %(right unit)%
    . $(x@y)@z = x@(y@z)$ %(assoc)%
    . $(x@y) = (y@x)$ %(comm)%
end

Let C_M and of C_{CM} be model classes of MONOID and COMMMONOID respectively, we show that $C_M \rightsquigarrow C_{CM}$. This trivially holds as every model of C_{CM} is a model of C_M.

■

The notion of model class inclusion over the same signature is not enough to capture realistic refinement steps. For instance, let us consider the specification of a *Ring*.

spec RING =
  sort R
  ops $1 : R$; $0 : R$; $\_ + \_ : R \times R \to R$; $\_ * \_ : R \times R \to R$
  axioms
    vars $x, y, z : R$
    . $1 * x = x$ %(left unit 1)%
    . $x * 1 = x$ %(right unit 1)%
    . $(x * y) * z = x * (y * z)$ %(assoc)%
    . $0 + x = x$ %(left unit 0)%
    . $x + 0 = x$ %(right unit 0)%
    . $(x + y) + z = x + (y + z)$ %(assoc2)%
    . $x + y = y + x$ %(comm)%
    . $x * (y + z) = (x * y) + (x * z)$ %(distrib1)%
    . $(x + y) * z = (x * z) + (y * z)$ %(distrib2)%
end

Very rarely in the process of program development does the user work with just a single signature: operations and sorts of data are renamed, added and hidden as the need arises. This is captured by the *signature morphism*. A signature morphism $\sigma : \Sigma \to \Sigma'$ maps the sorts and operations of Σ to those in Σ'. This results in a translation of any Σ-equation φ to a Σ'-equation $\sigma(\varphi)$, and on semantic level, a translation of any Σ'-algebra $A' \in \mathbf{Mod}(\Sigma')$ to its reducts $A'|_\sigma \in \mathbf{Mod}(\Sigma)$.

DEFINITION 3.3.3 *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Let C_1 and C_2, be model classes of Σ and Σ' respectively. A refinement based on model class inclusion with change of signature is defined as follows:*

$$C_1 \rightsquigarrow^\sigma C_2$$

if and only if $C_2|_\sigma \subseteq C_1$, for $C_1 \subseteq \mathbf{Mod}(\Sigma)$ and $C_2 \subseteq \mathbf{Mod}(\Sigma')$.

EXAMPLE 3.3.4 Let C_M, C_R be the model classes of MONOID and RING respectively. We show that $C_M \rightsquigarrow^\sigma C_R$, where $\sigma : \Sigma(\text{MONOID}) \to \Sigma(\text{RING})$ is a signature morphism such that

$$\sigma(Elem) := R \quad \sigma(@) := * \quad \sigma(n) := 1.$$

Let $A \in C_R$, then:

$$A \models \forall x, y, z : R.\ ((x * y) * z) = (x * (y * z))$$
$$A \models \forall x : R.\ x * 1 = x$$
$$A \models \forall x : R.\ 1 * x = x.$$

We first consider the *assoc* axiom. This holds by definition if and only if $\forall \nu : X \to A$: $\nu \Vdash ((x * y) * z) = (x * (y * z))$. Here, ν is a total variable evaluation and $X = \{x : R, y :$

$R, z : R\}$ is a variable system. This is the case if and only if $\nu^{\#}((x * y) * z) = \nu^{\#}(x * (y * z))$. For the left hand side we compute:

$$\begin{aligned} \nu^{\#}((x * y) * z) &= A(*)(\nu^{\#}(x * y), \nu^{\#}(z)) \\ &= A(*)(A(*)(\nu^{\#}(x), \nu^{\#}(y)), \nu^{\#}(z)) \end{aligned}$$

Same computation is done also for the left hand side of the *left unit* axiom:

$$\begin{aligned} \nu^{\#}(1 * x) &= A(*)(\nu^{\#}(1), \nu^{\#}(x)) \\ &= A(*)(A(1), \nu^{\#}(x)) \end{aligned}$$

and *right unit* axiom:

$$\begin{aligned} \nu^{\#}(x * 1) &= A(*)(\nu^{\#}(x), \nu^{\#}(1)) \\ &= A(*)(\nu^{\#}(x), A(1)) \end{aligned}$$

Similar result holds also for the right hand side of each axioms. Now let $B \in C_M$ then:

$$\begin{aligned} &B \models \forall\, x, y, z : \mathit{Element}.\ ((x@y)@z) = (x@(y@z)) \\ &B \models \forall\, x : \mathit{Element}.\ x@n = x \\ &B \models \forall\, x : \mathit{Element}.\ n@x = x. \end{aligned}$$

is required to hold for all $\mu : \{x : \mathit{Element}, y : \mathit{Element}, z : \mathit{Element}\} \to B$.

We have to show that $A\,|_\sigma \in C_M$. Let $\alpha : \{x, y, z\} \to A\,|_\sigma$ be a variable evaluation. Corresponding to α we define a variable evaluation $\beta : \{x : R, y : R, z : R\} \to A$ as follows:

$$\beta(x) := \alpha(x) \quad \beta(y) := \alpha(y) \quad \beta(z) := \alpha(z)$$

Consider $(A\,|_\sigma\ (@))((A\ |_\sigma\ (@))(\alpha(x : \mathit{Element}), \alpha(y : \mathit{Element}), \alpha(z : \mathit{Element})))$, it follows:

$$\begin{aligned} & (A(\sigma(@))((A(\sigma)(@))(\alpha(x : \sigma(\mathit{Element})), \alpha(y : \sigma(\mathit{Element}))), \alpha(z : \sigma(\mathit{Element}))) \\ = \ & A(*)(A(*)(\alpha(x : \sigma(\mathit{Element})), \alpha(y : \sigma(\mathit{Element}))), \alpha(z : \sigma(\mathit{Element}))) \\ = \ & A(*)(A(*)(\beta(x : R), \beta(y : R)), \beta(z : R)). \end{aligned}$$

Same evaluation holds also for the *left unit* and *right unit* axioms:

$$\begin{aligned} & (A\ |_\sigma\ (@))((A\ |_\sigma\ (n)), \alpha(x : \mathit{Element})) \\ = \ & A(\sigma(@))(A(\sigma(n)), \alpha(x : \sigma(\mathit{Element}))) \\ = \ & A(*)(A(1), \beta(x : R)) \end{aligned}$$

Again similar result holds also for the right hand side of each axioms. Hence, we have that $C_M \rightsquigarrow^\sigma C_R$, i.e., MONOID $\rightsquigarrow^\sigma$ RING. ∎

3.3.1 HETS – tool for CASL

The Heterogeneous Tool Set (HETS)[MML07] is a parsing, static analysis and proof management tool for various specification languages centered around CASL. HETS is an interactive system with a graphical user interface and also facility to be called on a command line.

HETS is a system which keeps track of open proof goals (which are caused by theorems which have not yet been proven) and closed proof goals (which are caused by theorems which have been proven or disproven). HETS reads a specification text possibly including open proof goals, parses it, and then performs static analysis on it. After this, a graph of the structure of the specification is displayed in its user interface. Within this graph, the user can see which goals are open and which are closed. The user can also perform various operations on each node in the graph, for instance requesting the theory of such a node, HETS will then display the relevant information.

HETS can interface with different theorem provers, including Isabelle [NPW02] and SPASS [spa]. HETS can call theorem provers with proof obligations and axioms given by specifications in order to discharge open proof goals. This allows the user to pass control over to a theorem prover to discharge open proof obligations.

(CHAPTER ...4)

Processes and Data: CSP-CASL

Contents

THE specification language CSP-CASL integrates data specification in CASL and process specification in CSP. Following [Rog06], we present the language's main features and its fundamental concepts. In Section 4.2 we describe the semantical concepts of CSP-CASL and a simple refinement notion. As a running example we use a binary calculator (taken from [KRS07]), to illustrate the various concepts.

4.1 CSP-CASL – fundamental concepts

CSP-CASL [Rog06] is a specification language which combines *processes* written in CSP with the specification of *data types* in CASL [Mos04]. The general idea is to describe reactive systems in the form of processes based on CSP operators, where the communications of these processes are the values of data types, which are loosely specified in CASL. All standard CSP operators are available, such as multiple prefix, the various parallel operators and operators for non-deterministic choice, communication over channels (see Chapter 2). Concerning CASL features, the full language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and sub-sorting. Furthermore, the various CASL structuring constructs are included, where the structured **free** construct adds the possibility to specify data types with initial semantics (see Chapter 3).

Syntactically, a CSP-CASL specification with name *Sp* consists of a data part *D*, which is a structured CASL specification, an (optional) channel part *Ch* to declare channels, which are typed according to the data part, and a process part *P* written in CSP.

ccspec *Sp* = **data** *D* **channel** *Ch* **process** *P* **end**

In the process part P, the CASL terms are used as communications – CASL sorts denote sets of communications, relational renaming is described by a binary CASL predicate, and the CSP conditional construct uses CASL formulae as conditions.

In the process part, recursive process definitions may be written using systems of process equations, binding processes to process names. Processes can also be parameterised with variables typed by CASL sorts.

The channel part Ch is just a syntactic encoding over the data part. A CSP-CASL specification with channel declaration can be translated into one without a channel declaration as presented in Figure 4.1.

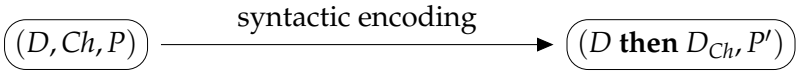

Figure 4.1: Channel declaration – syntactic encoding.

Here, D_{Ch} is a data specification fragment, which monomorphically extends the data part D to a CASL specification D **then** D_{Ch}. This construction neither adds new diversity nor removes interpretation of the data part. The process P is rewritten to a form P' without channels.

The definition of CSP-CASL is generic in the choice of a specific CSP semantics. For example, all denotational CSP models mentioned in Chapter 2, are possible parameters. In CSP-CASL the data type (specified in CASL) not only provides the values of the alphabet of communication, but they also provide certain test functions. For instance, as reported in [Rog06], in CSP-CASL the following tests are possible:

- Test on equality for arbitrary CASL terms – Can two communications synchronize?
- Test on membership for a CASL term concerning a CASL sort – Does a communication belong to a certain subset of the alphabet of communications?
- Test whether a binary predicate holds between two CASL terms – Are the terms in a renaming relation?
- Satisfaction of a CASL first order formula – Is the formula of the conditional construct true?

Tool support for CSP-CASL consists of: A parser and static analyzer described in [Gim08]; A prover for process and data refinement, described in [OIR09].

We now give a concrete instance of CSP-CASL specification syntax through an example of a *binary calculator* specification. We first give a note on syntax in CSP-CASL. The syntax of CSP used in CSP-CASL, is slightly different than the one presented in Chapter 2. Figure 4.2 illustrates the different syntax of the CSP operators. This change of concrete syntax is required to resolve overlaps in the syntax of CSP and CASL.

Process Type	Implemented in CSP-CASL	CSP syntax
External prefix choice	$?x :: s \rightarrow P$	$!x : s \rightarrow P$
Internal prefix choice	$!x :: s \rightarrow P$	$?x : s \rightarrow P$
Channel send	$c!t \rightarrow P$	$c!x \rightarrow P$
Channel nondeterministic send	$c!x :: s \rightarrow P$	$c!x : s \rightarrow P$
Channel nondeterministic receive	$c?x :: s \rightarrow P$	$c?x : s \rightarrow P$

Figure 4.2: CSP notation in CSP-CASL– c.f. [Gim08].

EXAMPLE 4.1.1 Our binary calculator (Figure 4.3) has two input buttons and can compute the addition function only. In the end, the implemented binary calculator has the following characteristics: whenever one of the buttons is pressed, the integrated control circuit displays the corresponding digit on the display. After pressing a second button, the corresponding addition result is displayed and the calculator returns to its initial state.

Figure 4.3: Binary Calculator.

In a first high-level specification we abstract from the control flow and just specify the interface of the system.

ccspec BCALC0 =
  data sort *Number*
    ops 0, 1 : *Number*;
      $\_+\_ : Number \times Number \rightarrow ? Number$
  channels *Button* : *Number*;
    Display : *Number*
  process *P0* : *Button*, *Display* ;
    $P0 = Button\ ?\ x :: Number \rightarrow P0 \sqcap Display\ !\ y :: Number \rightarrow P0$
end

In this specification, 0 and 1 are constants of sort *Number*, and $+$ is a partial function from pairs of type *Number* to *Number*. In the channel part, the statement *Button*:*Number* and *Display*:*Number* declares two channels *Button* and *Display* of sort *Number*. In the process part, we first declare the process name *P*0, which is typed over the events that it communicates.

The calculator receives values of type *Number* on the channel *Button*, while it sends values of type *Number* over the channel *Display*. The process $Button?x :: Number \rightarrow P_0$ is

willing to receive any value of type *Number* over the channel *Button*, stores this value of type *Number* in x, and behaves like $P0$. This corresponds to a user input. The process $Display!y \rightarrow P0$ chooses an arbitrary value y of type *Number*, sends this value over the channel *Display*, and behaves like $P0$. This corresponds to the computed output of the calculator. The process $P0$ as a whole repeatedly chooses one of the two above processes in a non-deterministic way, which corresponds to an arbitrary interleaving of inputs and outputs.

In the next section, we will present a more refined specification of BCALC0. ■

4.2 CSP-CASL – semantical construction

In this section we describe the semantical construction of CSP-CASL. We illustrate how the alphabet of communication is constructed and a simple refinement notion of CSP-CASL specification presented in [Rog06]. As a consequence of CASL's loose semantics, *semantically*, a CSP-CASL specification $Sp = (D, P)$ is a family of process denotations for a CSP process P, where each model of the data part D gives rise to one process denotation.

In Chapter 3 we have presented different institutions for CASL. CSP-CASL as described in [Rog06], requires an additional institution: the institution $FinCommSubPFOL^{=}$.

The definition of the institution $FinCommSubPFOL^{=}$ provides the data-logic of the process part of a CSP-CASL specification. It is a specialisation of the institution $SubPCFOL^{=}$: Only sub-sorted-signatures with finitely many sorts are allowed. Also, the notion of a model is changed: A *data-logic* Σ*-model* M is the strict extension $M := ext(C)$ of an ordinary many-sorted model C over $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$ which satisfies in $PCFOL^{=}$ the set of axioms $\hat{J}(\Sigma)$. For the carrier sets, this extension is defined as: $M_s = ext(C_s) = C_s \cup \{\bot\}$ for all $s \in \hat{S}$, where $\bot \notin C_s$ for all $s \in \hat{S}$. Given a model C, its extension $ext(C) = M$ is uniquely determined. Forgetting the strict extension results again in C.

Alphabet construction The purpose of the alphabet construction is to transform a CASL model into a set for use as an alphabet of communication in the process algebra CSP.

Let $Sp = (D, P)$ be a CSP-CASL specification, and let model M over the data D signature $\Sigma = (S, TF, PF, P, \leqslant)$, with local top elements, i.e., for all $u, u', s \in S$ the following holds: if $u, u' \geqslant s$ then there exists $t \in S$ with $t \geqslant u, u'$. The alphabet of communications is constructed as follows: Relatively to a model M, the alphabet

$$\mathcal{A}lph(M) := (\biguplus_{s \in S} M_s \cup \{\bot_s\})_{/\sim_M} \qquad (*)$$

is constructed by disjointly uniting all carrier sets extended by a bottom element $\bot$, but identifying carriers along subsort injections. The latter is captured by the equivalence relation $\sim_M$. The relation $\sim_M$ is an equivalence relation for any model M. $\sim_M$ is defined as follows:

$$(s, x) \sim_M (s', x')$$

if and only if either

- $x = x' = \bot$ and
- there exists $u \in S$ such that $s \leq u$ and $s' \leq u$,

or

- $x \neq \bot, x' \neq \bot$,
- there exists $u \in S$ such that $s \leq u$ and $s' \leq u$, and
- for all $u \in S$ with $s \leq u$ and $s' \leq u$ the following holds:

$$(\mathtt{inj}_{(s,u)})_M(x) = (\mathtt{inj}_{(s',u)})_M(x')$$

for $s, s' \in S$, $x \in M_s, x' \in M_{s'}$.

The semantics of CSP-CASL is defined in a two-step approach, see Figure 4.4.

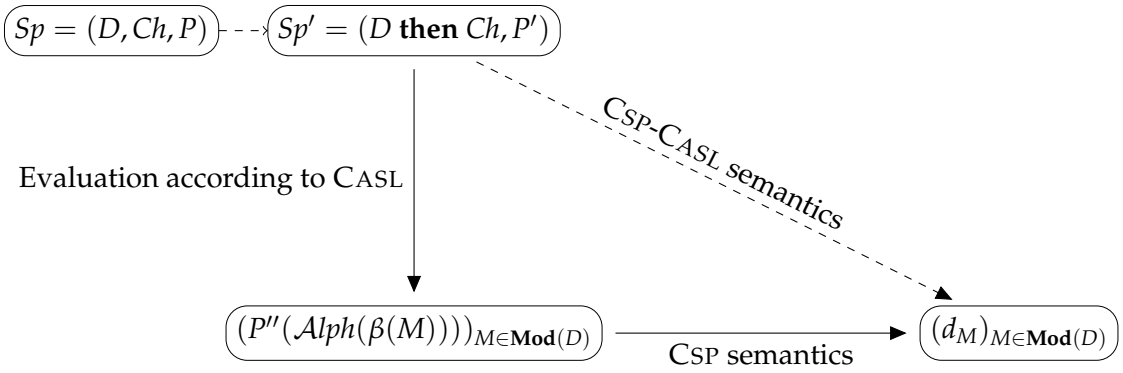

Figure 4.4: CSP-CASL 2-step semantics.

Let $Sp = (D, Ch, P)$ be a CSP-CASL specification, i.e., D is a CASL specification, Ch is the (optional) channel declaration and P is a CSP process.

We first obtain a CSP-CASL specification $Sp' = (D', P')$, without the channel declaration; here $D' = (D \textbf{ then } Ch)$. In the first step, the evaluation according to CASL, we translate D' into a $\textbf{Mod}(D')$-indexed family of CSP processes:

$$(P''(\mathcal{A}lph(\beta(M))))_{M \in \textbf{Mod}(D')}.$$

Here, we define for each model M of D' a CSP process $P''(\mathcal{A}lph(\beta(M)))$ over the alphabet of communication $\mathcal{A}lph(\beta(M))$ induced by M. This alphabet is constructed by first obtaining a model $\beta(M)$, in which partial functions of M are totalized. Then, we use the alphabet construction $\mathcal{A}lph$ as described in $(*)$.

We now present the evaluation of CASL terms, sorts, formulae, and relations occurring in P''. In order to do this, an evaluation function $[\![\_]\!]\_$ is defined, which takes a CSP-CASL process specification and an evaluation $\nu : X \to \beta(M)$ and yields a CSP process

$$
\begin{array}{lcl}
[\![SKIP]\!]_\nu & := & SKIP \\
[\![STOP]\!]_\nu & := & STOP \\
[\![DIV]\!]_\nu & := & DIV \\
[\![t \rightarrow P]\!]_\nu & := & [\![t]\!]_\nu \rightarrow [\![P]\!]_\nu \\
[\![?x :: s \rightarrow P]\!]_\nu & := & ?x :: [\![s]\!]_\nu \rightarrow [\![P]\!]_{(\lambda\, z.\nu)} \\
[\![!x :: s \rightarrow P]\!]_\nu & := & !x :: [\![s]\!]_\nu \rightarrow [\![P]\!]_{(\lambda\, z.\nu)} \\
[\![P \fatsemi Q]\!]_\nu & := & [\![P]\!]_\nu \fatsemi [\![Q]\!]_\emptyset \\
[\![P \,\Box\, Q]\!]_\nu & := & [\![P]\!]_\nu \,\Box\, [\![Q]\!]_\nu \\
[\![P \sqcap Q]\!]_\nu & := & [\![P]\!]_\nu \sqcap [\![Q]\!]_\nu \\
[\![P \,|[\, s \,]|\, Q]\!]_\nu & := & [\![P]\!]_\nu \,|[\, [\![s]\!]_\nu \,]|\, [\![Q]\!]_\nu \\
[\![P \;\; |[\, s_1 \mid s_2 \,]|\;\; Q]\!] & := & [\![P]\!]_\nu \;\; |[\, [\![s_1]\!]_\nu \mid [\![s_2]\!]_\nu \,]|\;\; [\![Q]\!]_\nu \\
[\![P \,||\, Q]\!]_\nu & := & [\![P]\!]_\nu \,||\, [\![Q]\!]_\nu \\
[\![P \,|||\, Q]\!]_\nu & := & [\![P]\!]_\nu \,|||\, [\![Q]\!]_\nu \\
[\![P \setminus s]\!]_\nu & := & [\![P]\!]_\nu \setminus [\![s]\!]_\nu \\
[\![P[[p]]]\!]_\nu & := & [\![P]\!]_\nu[[[\![p]\!]_\nu]] \\
[\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu & := & \textbf{if } [\![\varphi]\!]_\nu \textbf{ then } [\![P]\!]_\nu \textbf{ else } [\![Q]\!]_\nu
\end{array}
$$

Figure 4.5: Evaluation according to CASL

over $\mathcal{A}lph(\beta(M))$. Here, the evaluations ν deal with CSP binding. Figure 4.5 illustrates definition of the variable evaluations ν necessary to model the CSP binding. In Figure 4.5, the clause for prefix choice turns the current environment ν into a function $(\lambda\, z.\nu)$ which takes a substitution as its argument:

$$[\![\_\!]\!]_{\lambda\, z.\nu}[a/x] := [\![\_\!]\!]_{\nu[a/x]}$$

Here, $\nu[a/x](y) := \nu(y)$ for $y \neq x$ and $\nu[a/x](x) := a$. Substitutions are the way how the various CSP semantics model the binding concept of the prefix choice operator.

In the evaluation according to CSP, we apply point-wise a denotational CSP semantics. This translates a process $P''(\mathcal{A}lph(\beta(M)))$ into its denotation d_M in the semantic domain of the chosen CSP semantics. We will indicate as $[\![P]\!]_{\emptyset:\emptyset\rightarrow\beta(M)}$ to denote the process notation with empty evaluation $\emptyset$, i.e., P has no free variables.

As an example of CSP-CASL semantical construction, we revisit the example of the binary calculator (Example 4.1.1).

EXAMPLE 4.2.1 (SEMANTICAL CONSTRUCTION) We consider the specification of the binary calculator presented in the last section. Here, we construct the semantical model of BCALC0. Let BCALC0C be the equivalent of BCALC0 without the channel declaration, i.e., $Sp =$ BCALC0 and $Sp' =$ BCALC0C in Figure 4.4.

ccspec BCALC0C =
  data sort *Number*
    ops 0, 1 : *Number*;
      $\_\_+\_\_ : Number \times Number \to ? Number$
    then **free type** *Button* ::= *butt*(*n* : *Number*)
      free type *Display* ::= *disp*(*m* : *Number*)
  process *P0c* : *Button*, *Display* ;
    $P0c = (?\ x :: Button \to P0c) \sqcap (!\ y :: Display \to P0c)$
end

The data signature of BCALC0C, which is a subsorted CASL signature, is composed of:

$$\Sigma(\text{BCALC0C}) = (\{Number, Button, Display\}, \{0, 1, butt, disp, n, m\}, \{\_\_ + \_\_\}, \emptyset, \emptyset)$$

Let M be a CASL model such that:

$$\begin{array}{lll} M(Number) = \{H, L\} & M(O) = L & M(1) = H \\ M(Button) = \{b.H, b.L\} & M(Display) = \{d.H, d.L\} & \end{array}$$

Here, $M(butt)(x) = b.x$ and $M(disp)(x) = d.x$ where $x \in \{H, L\}$. We totalize the model M by adding the bottom element for each sort:

$$\begin{array}{l} \beta(M)(Number) = M(Number) \cup \{\bot_{Number}\} \\ \beta(M)(Button) = M(Button) \cup \{\bot_{Button}\} \\ \beta(M)(Display) = M(Display) \cup \{\bot_{Display}\} \end{array}$$

The CSP-CASL semantics construct the alphabet of communication. Here, as there is no subsorting, the equivalence relation $\sim_M$ is the identity relation. Thus, the alphabet of communication for the process *P0c* is:

$$\begin{array}{rcl} \mathcal{A}lph(\beta(M)) & = & \beta(M)(Number) \cup \beta(M)(Button) \cup \beta(M)(Display) \\ & = & \{H, L, \bot_{Number}\} \cup \{b.H, b.L, \bot_{Button}\} \cup \{d.H, d.L, \bot_{Display}\} \end{array}$$

We now construct the process denotation in the traces model:

$$traces([\![\text{P0C}]\!]_M) \quad = \quad \{tr \mid tr \leq (b \cup d)^*,\ b \in M(Button) \text{ and } d \in M(Display)\}$$

Here, $(b \cup d)^*$ is the regular expression, using '$\cup$' for choice between two regular languages and * for Kleene's star operator. ■

4.2.1 CSP-CASL simple refinement notion

As described in the previous section, for a denotational CSP model with domain $\mathcal{D}$, the semantic domain of CSP-CASL specification $Sp = (D, P)$ consists of the $\mathbf{Mod}(D)$-indexed families of process denotations $d_M \in \mathcal{D}$, i.e.,

$$(d_M)_{M \in \mathbf{Mod}(D)}.$$

A refinement notion for CSP-CASL is defined over these elements. CSP-CASL refinement is based on refinements developed in the context of the single languages CSP and CASL. Intuitively, a refinement step, which we write here as '$\rightsquigarrow$', reduces the number of possible implementations. Concerning data, this means a reduced model class, concerning processes this mean less non-deterministic choice:

DEFINITION 4.2.2 *For families* $(d_M)_{M \in \mathbf{Mod}(D)}$ *and* $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$ *of process denotations we write*

$$(d_M)_{M \in \mathbf{Mod}(D)} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M' \in \mathbf{Mod}(D')}$$
$$\textit{iff}$$
$$\mathbf{Mod}(D') \subseteq \mathbf{Mod}(D) \wedge \forall M' \in \mathbf{Mod}(D') : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'}.$$

Here, $\mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathcal{D}}$ is the refinement notion in the chosen CSP model $\mathcal{D}$. In the traces model $\mathcal{T}$, as described in Chapter 2, we have $P \sqsubseteq_{\mathcal{T}} P' \Leftrightarrow \mathit{traces}(P') \subseteq \mathit{traces}(P)$, where $\mathit{traces}(P)$ and $\mathit{traces}(P')$ are prefixed closed sets of traces. Here we follow the CSP convention, where P *refines to* P' is written as $P \sqsubseteq_{\mathcal{D}} P'$, i.e., the more specific process is on the right-hand side of the symbol. The definitions of CSP refinements for $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$, c.f. Chapter 2, which are all based on set inclusion, yield that CSP-CASL refinement is a preorder.

Given CSP-CASL specifications $Sp = (D, P)$ and $Sp' = (D', P')$, by abuse of notation we also write

$$Sp \rightsquigarrow_{\mathcal{D}} Sp'$$

if the above refinement notion holds for the denotations of Sp and Sp', respectively.

On the syntactic level of specification text, we additionally define the notions of data refinement and process refinement in order to characterize situations, where one specification part remains constant. In a *data refinement*, only the data part changes:

$$\left.\begin{array}{l} \textbf{ccspec } Sp = \textbf{ data } D \textbf{ process } P \textbf{ end} \\ \qquad\qquad\qquad \overset{\texttt{data}}{\rightsquigarrow} \\ \textbf{ccspec } Sp = \textbf{ data } D' \textbf{ process } P \textbf{ end} \end{array}\right\} \quad \text{if} \quad \left\{\begin{array}{l} 1.\ \Sigma(D) = \Sigma(D'), \\ 2.\ \mathbf{Mod}(D') \subseteq \mathbf{Mod}(D). \end{array}\right.$$

Here, $\Sigma(D)$ denotes the CASL signature of D. As in a data refinement the process part remains the same, there is no need to annotate data refinement with a specific process model: all CSP refinements notions are reflexive. In a *process refinement*, the data part is constant:

$$\left.\begin{array}{l} \textbf{ccspec } Sp = \textbf{ data } D \textbf{ process } P \textbf{ end} \\ \qquad\qquad\qquad \overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}} \\ \textbf{ccspec } Sp = \textbf{ data } D \textbf{ process } P' \textbf{ end} \end{array}\right\} \quad \text{if} \quad \left\{\begin{array}{l} \text{for all } M \in \mathbf{Mod}(D) : \\ [\![[\![P]\!]_{\emptyset:\emptyset \to \beta(M)}]\!]_{\mathcal{D}} \sqsubseteq_{\mathcal{D}} [\![[\![P']\!]_{\emptyset:\emptyset \to \beta(M)}]\!]_{\mathcal{D}}. \end{array}\right.$$

Here, $[\![\_]\!]_{\mathcal{D}}$ is the evaluation according to the CSP denotational semantics $\mathcal{D} \subseteq \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$, and $\emptyset : \emptyset \to \beta(M)$ is the empty evaluation into the model $\beta(M)$.

Clearly, both these refinements are special forms of CSP-CASL refinement in general.

LEMMA 4.2.3 *Let $Sp = (D, P)$, $Sp\_d = (D', P)$, $Sp\_p = (D', P')$ be* CSP-CASL *specifications. Then, for all* CSP *models $\mathcal{D} \subseteq \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$ it holds:*

1. *$Sp \overset{data}{\rightsquigarrow} Sp\_d$ implies $Sp \rightsquigarrow_{\mathcal{D}} Sp\_d$, and*
2. *$Sp \overset{proc}{\rightsquigarrow_{\mathcal{D}}} Sp\_p$ implies $Sp \rightsquigarrow_{\mathcal{D}} Sp\_p$.*

PROOF. Let $(d_M)_{M \in \mathbf{Mod}(D)}$, $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$ and $(d''_M)_{M \in \mathbf{Mod}(D)}$ be the families of process denotations of *Sp*, *Sp_d* and *Sp_p* respectively. We prove the implication in (1) and (2).

1. We need to show that:

$$(d_M)_{M \in \mathbf{Mod}(D)} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M' \in \mathbf{Mod}(D')}$$

   This holds if $\mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)$ and $\forall M' \in \mathbf{Mod}(D').d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'}$. The data refinement: $(d_M)_{M \in \mathbf{Mod}(D)} \overset{data}{\rightsquigarrow} (d'_{M'})_{M' \in \mathbf{Mod}(D')}$ establishes the inclusion of the model classes, i.e., $\mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)$, where $\Sigma(D) = \Sigma(D')$. For the process refinement we have: $\forall M' \in Mod(D').\ d'_M =_{\mathcal{D}} d'_{M'}$. Hence, $(d_M)_{M \in \mathbf{Mod}(D)} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M' \in \mathbf{Mod}(D')}$, i.e., $Sp \rightsquigarrow_{\mathcal{D}} Sp'$.

2. Again, here we need to show that:

$$(d_M)_{M \in \mathbf{Mod}(D)} \rightsquigarrow_{\mathcal{D}} (d''_M)_{M \in \mathbf{Mod}(D)}$$

   Here, we work with the same model classes $\mathbf{Mod}(D)$. From the process refinement $(d_M)_{M \in \mathbf{Mod}(D)} \overset{proc}{\rightsquigarrow_{\mathcal{D}}} (d''_M)_{M \in \mathbf{Mod}(D)}$ we have: $\forall M \in \mathbf{Mod}(D).\ d_M \sqsubseteq_{\mathcal{D}} d''_M$. Hence, $(d_M)_{M \in \mathbf{Mod}(D)} \rightsquigarrow_{\mathcal{D}} (d''_{M'})_{M' \in \mathbf{Mod}(D')}$, i.e., $Sp \rightsquigarrow_{\mathcal{D}} Sp'$.

■

EXAMPLE 4.2.4 In this example, we show a refinement of the binary calculator specified in Example 4.1.1. A first refinement, could require that the pressing of buttons and the display of digits strictly alternate [1].

ccspec BCALC1 =
  data sort *Number*
    ops 0, 1 : *Number*;
      $\_+\_$: *Number* × *Number* →? *Number*
  channels *Button* : *Number*;
    Display : *Number*
  process *P1* : *Button*, *Display* ;
    P1 = *Button* ? *x* :: *Number* →*Display* ! *y* :: *Number* →*P1*
end

[1] In the process part of BCALC1, we write down explicitly the sort type (*Number*) in the channel nondeterministic receive (*Button*?*x* :: *Number*). This is a design decision taken in the development of the parser and static analyzer of CSP-CASL [Gim08]. However, as this case shows, typing can be unique and the current design of CSP-CASL forces the user to give superfluous type information.

We show by fixed point induction that BCALC0 $\overset{\text{proc}}{\rightsquigarrow_{\mathcal{F}}}$ BCALC1.

$$\begin{array}{lll} & & \mathit{Button}?x :: \mathit{Number} \to P0 \sqcap \mathit{Display}!y :: \mathit{Number} \to P0) \\ 1. & =_{\mathcal{F}} & (\mathit{Button}?x :: \mathit{Number} \to ((\mathit{Button}?x :: \mathit{Number} \to P0) \\ & & \sqcap (\mathit{Display}!y :: \mathit{Number} \to P0)) \sqcap (\mathit{Displat}!y :: \mathit{Number} \to P0) \\ 2. & \sqsubseteq_{\mathcal{F}} & \mathit{Button}?x :: \mathit{Number} \to ((\mathit{Button}?x :: \mathit{Number} \to P0) \\ & & \sqcap (\mathit{Display}!y :: \mathit{Number} \to P0)) \\ 3. & \sqsubseteq_{\mathcal{F}} & \mathit{Button}?x :: \mathit{Number} \to \mathit{Display}!y :: \mathit{Number} \to P0 \end{array}$$

The proofs use standard CSP algebraic step laws: In step 1, we unwind the recursion at the first occurrence of $P0$. In step 2, we leave out the second branch of the last internal choice, i.e., $P \sqcap Q \sqsubseteq_{\mathcal{F}} P$. Finally, in step 3, we select the second branch of the internal choice.

Again, we refine furthermore the specification BCALC1. Here, we require that the first displayed digit is echoing the input, and the second displays the result of the computation.

ccspec BCALC2 =
  data sort *Number*
    ops $0, 1 : \mathit{Number}$;
      $\_+\_ : \mathit{Number} \times \mathit{Number} \to ? \mathit{Number}$
  channels $\mathit{Button} : \mathit{Number}$;
    $\mathit{Display} : \mathit{Number}$
  process $P2 : \mathit{Button}, \mathit{Display}$;
    $P2 = \mathit{Button} ? x :: \mathit{Number} \to \mathit{Display} ! y :: \mathit{Number}$
      $\to \mathit{Button} ? y :: \mathit{Number} \to \mathit{Display} ! (x + y) \to P2$
end

We show that BCALC1 $\overset{\text{proc}}{\rightsquigarrow_{\mathcal{F}}}$ BCALC2.

$$\begin{array}{lll} & & \mathit{Button}?x :: \mathit{Number} \to \mathit{Display}!y :: \mathit{Number} \to P1 \\ 1. & =_{\mathcal{F}} & \mathit{Button}?x :: \mathit{Number} \to \mathit{Display}!y :: \mathit{Number} \to \mathit{Button}?x' :: \mathit{Number} \\ & & \to \mathit{Display}!y :: \mathit{Number} \to P1 \\ 2. & \sqsubseteq_{\mathcal{F}} & \mathit{Button}?x :: \mathit{Number} \to \mathit{Display}!x \to \mathit{Button}?x' :: \mathit{Number} \\ & & \to \mathit{Display}!x' \to P1 \\ 3. & =_{\mathcal{F}} & \mathit{Button}?x :: \mathit{Number} \to \mathit{Display}!x \to \mathit{Button}?y :: \mathit{Number} \\ & & \to \mathit{Display}!(x + y) \to P1 \end{array}$$

Again, in step 1, we unwind the recursion of $P1$. As $P1$ is independent of x we can rename x into x'. In step 2, choosing the specific values x and $(x + x')$ for the two occurrences of

$(Display!y :: Number)$ is a refinement. Finally, renaming x' into y preserves the semantics of the process.

So far we have refined the process part, what about the data part? For instance in $P2$ it is still open the value of $x + y$ shall be. We haven't specified the arithmetic properties of addition. In CSP-CASL, we express this functionality by adding some suitable axioms.

ccspec BCALC3 =
  data sort *Number*
    ops $0, 1 : Number$;
      $\_+\_ : Number \times Number \to ? Number$
    axioms
      • $0 + 0 = 0$
      • $0 + 1 = 1$
      • $1 + 0 = 1$
  channels *Button* : *Number*;
    Display : *Number*
  process *P3* : *Button*, *Display* ;
    $P3 = Button\ ?\ x :: Number \to Display\ !\ y :: Number$
      $\to Button\ ?\ y :: Number \to Display\ !\ (x + y) \to P3$
end

Here, we have that BCALC2 $\overset{\texttt{data}}{\rightsquigarrow}$ BCALC3. Adding axioms to a signature without changing the process part always results in a data refinement.
I.e., $\Sigma(\text{BCALC2DATA}) = \Sigma(\text{BCALC3DATA})$, where BCALC2DATA and BCALC3DATA is the data part of BCALC3 and BCALC2 respectively, then we have that
$\mathbf{Mod}(\text{BCALC3DATA}) \subseteq \mathbf{Mod}(\text{BCALC2DATA})$. BCALC3 has models which satisfy the axioms $0 + 0 = 0$, $1 + 0 = 1$, $1 + 0 = 0$ and is undefined for $1 + 1$. As the axioms stated in BCALC3 hold, e.g., for the natural numbers, the data part is consistent. ■

Often we study a more elaborated theory of CSP-CASL, which is called *Multi-process* CSP-CASL. In such theory we allow the definition of several processes in the CSP part. In the following subsection we illustrate how the CSP-CASL original theory as presented in this chapter differs from *Multi-process* CSP-CASL.

4.2.2 Multi-Process CSP-CASL

In CSP-CASL, as designed in [Rog06], one specification denotes one unnamed system, see e.g., specification CSPCASLSPEC in Figure 4.6.

The semantics of this specification is given as one family of process denotations. Over the CSP traces model $\mathcal{T}$, this family has the following structure: In CASL models M with $M \models a = b$, the terms a and b can synchronize, and we obtain the denotation $\{\langle\rangle, \langle\overline{M(a)}\rangle\}$. Here, $\overline{M(a)}$ is the communication corresponding to the interpretation of the constant a

ccspec CSPCASLSPEC =
  data
    sort s
    ops $a, b : s$
  process
    let $P(x : s) = x \rightarrow STOP$
        $Q \quad = b \rightarrow STOP$
    in $P(a) \parallel Q$
end

logic CspCASL
ccspec MULTICSPCASLSPEC =
  data
    sort s
    ops $a, b : s$
  process
    $P(s) : s;\ Q : s;\ System : s$
    $P(x) \quad = x \rightarrow STOP$
    $Q \quad = b \rightarrow STOP$
    $System = P(a) \parallel Q$
end

Figure 4.6: Introduction of process names in *Multi-process* CSP-CASL.

in the model M. It is this point, where CSP-CASL differs from the pure CSP approach: CSP-CASL interprets terms relatively to a model, pure CSP works with communications without using an interpretation. In CASL models N with $N \models \neg a = b$, the terms a and b cannot synchronize, thus the process part is in a deadlock situation and we obtain the denotation $\{\langle\rangle\}$. Overall, the semantics of the CSPCASLSPEC is

$$\begin{array}{ll} & (\{\langle\rangle, \langle\overline{M(a)}\rangle\})_{M \in \{X \in \mathbf{Mod}(D_{\text{CSPCASLSPEC}}) \mid X \models a=b\}} \\ \cup & (\{\langle\rangle\})_{N \in \{X \in \mathbf{Mod}(D_{\text{CSPCASLSPEC}}) \mid X \models \neg a=b\}}. \end{array}$$

Here, $D_{\text{CSPCASLSPEC}}$ denotes the data part of CSPCASLSPEC. Clearly, the process names P and Q are only used to determine how the system behaves, they do not appear on the semantical level.

In contrast to this, the specification MULTICSPCASLSPEC binds denotations to process names. Rather than representing one system, it provides a collection of components. The specification MULTICSPCASLSPEC in Fig. 4.6, is a 'semantically equivalent' version to CSP-CASLSPEC.

Its models are pairs (M, J), where M is a CASL model and J is a 'process model'. Such a process model J maps process names to process denotations over the alphabet of communications derived from M.

Over the traces model $\mathcal{T}$, we obtain as semantics for MULTICSPCASL: For a CASL model M with $M \models a = b$, the process model J is the map

$$\begin{array}{lcl} J(P(\bar{a})) & = & \{\langle\rangle\} \cup \{\langle\bar{a}\rangle \mid \bar{a} \in \overline{\beta(M)(s)}\} \\ J(Q) & = & \{\langle\rangle, \langle\overline{M(b)}\rangle\} \\ J(System) & = & \{\langle\rangle\} \end{array}$$

Here, $\beta(M)$ denotes the extension of the CASL model M by bottom elements, and

$\overline{\beta(M)(s)}$ is the set of communications that is obtained from the carrier set of s in $\beta(M)$. Note, that this carrier set includes an element $\perp$ representing undefinedness, which leads

to a valid communication. For a CASL model N with $N \models \neg a = b$, the process model I is the map

$$\begin{array}{lcl} I(P(\bar{a})) & = & \{\langle\rangle\} \cup \{\langle\bar{a}\rangle, \mid \bar{a} \in \overline{\beta(N)(s)}\} \\ I(Q) & = & \{\langle\rangle, \langle\overline{N(b)}\rangle\} \\ I(System) & = & \{\langle\rangle, \langle\overline{N(a)}\rangle\} \end{array}$$

Note that the type of the functions I and J depends on the models M and N, respectively: the process P takes a value of type s as a parameter; thus, the carrier set of the sort s determines the typing of the process interpretation. The overall model class of MULTICSPCASL is finally

$$\begin{array}{rl} & \{(M,J) \mid M \in \mathbf{Mod}(D_{\text{MULTICSPCASLSPEC}}), M \models a = b\} \\ \cup & \{(N,I) \mid N \in \mathbf{Mod}(D_{\text{MULTICSPCASLSPEC}}), N \models \neg a = b\}. \end{array}$$

The specification MULTICSPCASLSPEC simply adds process name information: hiding the information concerning the system's components, in our example the processes P and Q, leads back to the original semantics of the specification CSPCASLSPEC.

In practice, it often comes handy to work with process names. Semantically, the treatment of process names is a straightforward extension. However, in order to avoid notational complexity, we develop our theoretical results using the original setting.

(CHAPTER ...5)

Related approaches

Contents

THE work presented in this thesis is related to various general research areas, including the combination of process and data specification, system development notions using different formal specification languages as well as specification based testing. This chapter contains references to approaches which define the context of this thesis.

5.1 Combining processes and data specification

Combination of process algebra and algebraic specification to form new formalisms have been studied since the early 80's. Astesiano et. al in [ABR99] presents a survey of methodologies of how algebraic specification can be used to describe concurrent systems. They distinguish four kinds of approaches:

Process algebra Use algebraic specification at the metalevel, for instance, in the definition or in the use of specification languages. A specification will then involve definition of one or more expressions of the language, representing one or more systems. Examples of such approach are for instance: ACP [BK84], CCS [Mil89], and CSP [Hoa85, Ros98].

Process calculi plus algebraic specifications of static data types A particular specification language for concurrent systems is complemented with the possibility of specifying abstract data types using algebraic specification. Examples of such approach are LOTOS [ISO89] (and later became E-LOTOS [JTC01]) and PSF [MV90]. We will describe LOTOS later on in this section.

PSF [MV90] which stands for *Process Specification Formalism*, is a specification language which combines process description based on ACP and algebraic specification of data based on Algebric Specification Formalism [BHK89] (ASF). A PSF specification consists of data modules and process modules. Data modules uses algebraic specification with initial semantics and equational logic with total algebra. Process modules are just ACP specifications of processes. The atomic actions in the process part may have as components some values of the specified data types. An extensive toolset has been developed for PSF (see, e.g., [MV92, PSF]).

Algebraic specification of dynamic-data types Use of particular algebraic specifications that have "dynamic sorts". Those are sorts whose elements correspond to concurrent systems. In this approach there is only one "algebraic model" (for instance a first-order structure or algebra) in which some elements represent concurrent systems. As an example of this technique, the authors presents an approach based on label transition systems called Label Transition Logic (LTL)[AR01].

Algebraic specification of dynamic data-types In this approach (abstract) data types are specified, which dynamically changes with time. Here, different "algebraic" models corresponds to different states of the system. The specialty of this approach is the presence of data types which are dynamic. A comprehensive summary of these approach can be found in [EO01].

There are several combinations of a process algebra with a state-based formalism; a few examples are discussed in [Abr03, Sto97, TA97, Fis98, Smi99, TS99, But99, Fis00, MD00]. In [SAA02a], a formal foundations is presented to make a generic combination of one process algebra language and one algebraic specification language.

Closely related to CSP-CASL are the specification languages LOTOS, μCRL and to a certain extent CIRCUS. In the following, we give a brief description of their main features.

LOTOS and E-LOTOS The Language of Temporal Ordering Specification- LOTOS [ISO89, BSS87, BB88] was the first internationally known (since 1984), algebraic specification formalism for concurrency. LOTOS is a specification language developed within ISO (*International Standards Organization*). Although originally developed for the formal specification of open distributed systems, and in particular for those related to the Open Systems Interconnection (OSI) computer network architecture it is applicable to distributed, concurrent systems in general. LOTOS is very similar to PSF; in the sense that it adds algebraic specifications into a language for concurrency. However, LOTOS uses ACT-ONE [EM85] instead of ASF in the data part; while on the process part uses an extension of CCS instead of the process algebra ACP. In some sense PSF is an improvement of LOTOS (see a discussion in [MV90]), since it allows more freedom in the definition of synchronization mechanism and supports of import/export of action/processes, thus becoming more flexible for stepwise development.

E-LOTOS [JTC01] (*Enhanced* LOTOS) is a new version of LOTOS. This new version enhances the data part with new built-in data types, and on the process part adds the interleaving semantics, plus real time and priorities features. Moreover it adds

modularity of specifications.

LOTOS has been used in several practical applications and comes with an extensive tool set. EUCALYPTUS [Gar96], which stands for *European/Canadian* LOTOS *Protocol Tool Set*, is a tool set developed for LOTOS. The tool comprises among others: a static analyzer, a simulator, a model generator, a model verifier (detection of deadlock and livelock), a C-code generator, a model displayer, a trace analyzer and a test case generator.

As mentioned earlier, in LOTOS, data are specified using ACT-ONE, which uses equational specification of data types with initial algebra semantics. For a relation between CASL and ACT-ONE we refer to [Mos02], which defines a representation of the institution underlying ACT-ONE in first order logic with equality, a sub-language of CASL (see Chapter 3). Furthermore, LOTOS uses initial semantics, while CASL provides both, initial and loose semantics. ACT-ONE does neither includes sub-sorting nor partiality.

μCRL [GP95] (micro *CRL*), where *CRL* stands for Common Representation Language, it is a specification language developed to study processes and data. Here, data types are specified using equational logic with total function. On the process part contains processes described in the usual process algebraic style, in particular the syntax is taken from ACP [BK84]. Processes are represented by process terms, which describe the order in which the actions from a set $\mathcal{A}$ may happen. A process term consists of action names and recursion variables combined by process algebraic operators.

Each μCRL specification determine a labelled transition system, which is defined by the structural operational semantics of μCRL [GP95]. The labelled transition system consist of states which are process terms and the edges are labelled with parameterised actions. Equivalence relations on the states in labelled transition systems is established using *branching bisimulation* [vGW96], which is sound for the proof theory of μCRL [GP94].

μCRL2 is an improved version of μCRL. On the data side, μCRL2 contains a predefined higher order data types, λ- calculus expressions and various other language constructs. The μCRL toolset (see `http://www.cwi.nl/~mcrl`) supports the analysis and manipulation of μCRL specifications.

CIRCUS [WC01, WC02] combines data operation specified in Z [WD96] and interaction specified in CSP, plus a refinement theory [CSW03]. The main motivation behind the development of such language was the need for a language for refinement which can describe programs but also capable of specifying high level models and designs. Programs in CIRCUS are declared as a sequence of paragraphs, which can either be a Z paragraph, a declaration of channels, a channel set declaration, or a process declaration. The semantics of CIRCUS is based on the Unyfing Theories of Programming (UTP) [HJ98] .

The declaration of process is composed by its name and its body specification. A process may be explicitly defined or composed in terms of other processes. An explicit

process definition contains a sequence of process paragraphs and a distinguished nameless main action, which defines its behaviors. Process paragraphs include Z paragraphs and declaration of actions. An action can be a schema, a primitive action like *SKIP*, a guarded command, an invocation to another action, or a combination of these constructs using CSP operators.

There are various approaches of reactive CASL extensions. The definition of CSP-CASL, like CCS-CASL [SAA01, SAA02b] or CASL-CHART [RR00], combines CASL with reactive systems of a particular kind. All these approaches result in specification frameworks able to model actual reactive systems. CASL-LTL [RAC00] and COCASL [MRS03] take a more fundamental approach: they extend CASL internally. In the case of CASL-LTL, the logic is extended by temporal operators, while COCASL dualizes the CASL by co-algebraic constructions.

5.2 System development notions

Formal development by stepwise refinement is one of the most prominent approaches in formal program development. In the literature, one finds an amazing number of approaches, methodological claims and pragmatical claims of stepwise refinement. Here, we mention some of them and give an overview of system development notions for the languages presented on the last section e.g., LOTOS, μCRL and CIRCUS.

LOTOS allows the specification of systems at different descriptive levels. The relationships between different LOTOS specification at different level of abstraction is studied by using a notion of equivalence, proposed in [Par81] and used for a CCS-like calculus in [Mil84]. This equivalence, known as *observational equivalence*, is based on the idea that the behaviour of a system is determined by the way it interacts with external observers. Examples of stepwise refinement in LOTOS can be found in [MV91, PS91], while in [DBBS96] a comparison between LOTOS and Z refinement notion is investigated.

A refinement notion for CIRCUS is described in [SWC02]. It starts from an abstract specification, and gradually, by iterations, it yields an implementation. Each iteration decomposes one process and typically includes three steps: a *simulation* that replaces the state components of the single abstract process with the components of all the distributed process to be derived; *action refinement* [vGG00] that partitions the concrete state and actions in such a way that actions from one partition access only its components; and finally, a process refinement that transforms the partitions in individual processes.

Horizontal development in terms of software product line, has received a great deal of attention for its potential in fostering reuse of software artifacts across the development phases. The concept of a software product lines (SPL) was introduced in the late 1990's (see e.g., [CW98, JRvdL00]), and extensively studied subsequently in [CN01, PBvdL05], with annual conferences and a huge body of engineering literature in [SPLb, SPLa].

5.3 Specification based testing

Traditionally formal methods and software testing have been seen as rivals. That is, they largely failed to inform one another and there was very little interaction between the two communities. In recent years, however, a new consensus has developed. Under this consensus, these approaches are seen as complementary [BBC+02, Hoa96]. A general overview of using formal specification to support software testing can be found in [HBB+09]. In this article, the authors explore the many ways in which the presence of a formal specification can support testing.

The work that will be presented in Chapter 9 on specification based testing for CSP-CASL builds on previous work, mainly in the area of LOTOS, see e.g., [ISO89, GJ99, BHT97]. A first formal treatment of testing was given by M-C. Gaudel [Gau95]. In [Mac99, Mac00], P. Machado presents the work of testing from structured algebraic specification. The main issue investigated here is the so-called oracle problem, that is, whether a decision procedure can be defined for interpreting the results of tests according to a formal specification. In the context of testing from algebraic specification, this consists in checking whether specification axioms are satisfied by programs.

Many research activities have been directed at finding appropriate theories and algorithms to derive test cases from formal specifications such that certain correctness properties can be guaranteed if the system under test passes all test cases of a test suite. Early attempts were contributes by E. Brinksma [Bri88] using the specification language LOTOS. Other approaches to generate test data have been investigated by J. Tretmans [Tre92]. The latter, studies the conformance testing of asynchronous communicating systems, based on general labelled transition systems. A general overview of approaches for the testing of transition systems including an annotated bibliography can be found in E. Brinksma and J. Tretmans [BT01].

Automatic generation of test data from formal specifications, is *not* the topic of this thesis; however, in the literature, one finds a large body of approaches, see e.g., [BJK+05, UL06, BBP96, Bin99].

In [CG07] M-C. Gaudel and A. Cavalcanti presents a model-based testing using CSP. Here, the authors are concentrated on testing for traces and failures refinement. In [Pel96], J. Peleska presents a pioneering work on CSP-based testing.

Testing for software product lines was investigated in [PM06, McG01] and others; the main focus of these papers is the informal or formal derivation of test cases from requirement and feature models.

PART II

A theory of development notions for CSP-CASL

(CHAPTER ...6)

CSP-CASL development notions

Contents

In Chapter 4 we have reported a simple notion of refinement for CSP-CASL based on model class inclusion over the same signature. However, in a refinement step, it is often the case that the signature changes. In this chapter we formulate two directions of system development: a refinement (or *vertical development*) notion for CSP-CASL; and an enhancement (or *horizontal development*) notion for CSP-CASL specifications.

These new notions of CSP-CASL refinement and CSP-CASL enhancement allow to change the signature of the data part. Such change of signature, however, does not "touch" the processes. The notion of process signature has been described in [MR07]; but is far for being a stable notion yet. Thus, we propose CSP-CASL refinement and CSP-CASL enhancement notions with change of signature for the data part only.

The results presented in this chapter have been published in [KR09] and [KRS08].

6.1 Theory of CSP-CASL refinement notion

In this section we define a general refinement notion for CSP-CASL. Such refinement notion is based on the original CSP-CASL theory [Rog06]. There, a specification describes only one unnamed process. Our notions of refinement for CSP-CASL are based on refinements developed in the context of the single languages CSP and CASL. In the context of algebraic specification, e.g., as mentioned in Chapter 3, *Ehrig et al* in [EK99] provide an excellent survey on different approaches. For CSP, each of its semantical models comes with a refinement notion of its own. There are for instance traces refinement, failure/divergences refinement, and stable failures refinement, see Section 2.2.

We now give an example to illustrate the type of refinement we would like to capture in CSP-CASL. Later, we define formally a notion of CSP-CASL refinement with arbitrary change of signature.

EXAMPLE 6.1.1 Let us consider the following two CSP-CASL specifications:

ccspec ABSTRACTSERVICE =
  data
    sort T
    ops $r1, r2, s1, s2 : T$
  process
    AbsSer(x:T) =
      $r1 \rightarrow s1 \rightarrow AbsSer(x)$
    $\Box$ $r2 \rightarrow s2 \rightarrow AbsSer(x)$
end

ccspec CONCRETESERVICE =
  data
    sort U
    ops $r3, r4, ser : U$
    axiom $\neg(r3 = r4)$
  process
    ConcSer(x:U) =
      if $(x = r3)$
        then $r3 \rightarrow ser \rightarrow ConcSer(r4)$
      else $r4 \rightarrow ser \rightarrow ConcSer(r3)$
end

Intuitively, the specification ABSTRACTSERVICE specifies a system which provides a service after a certain type of request has been made. That is, the process $AbsSer(x : T)$ behaves nondeterministically between choosing to offer the service $s1$ after a request $r1$, or offering the service $s2$ after a request $r2$. The order of $r1$ and $r2$ is left open. Only the additional specification of a scheduling mechanism would enforce that. This is done in the specification CONCRETESERVICE. Here, a scheduling mechanism is introduced using the if-then-else constructs of CSP.

The refinement step from ABSTRACTSERVICE to CONCRETESERVICE contains several aspects. On the data part we have change of signature: Let Σ and Σ' be the signature of the data part of ABSTRACTSERVICE and CONCRETESERVICE respectively, then $\sigma : \Sigma \rightarrow \Sigma'$ be a signature morphism such that:

$$\sigma^S(T) = U, \quad \sigma^F(r1) = r3, \quad \sigma^F(r2) = r4, \quad \sigma^F(s1) = ser, \quad \sigma^F(s2) = ser$$

Here, we notice that we have a non injective renaming of the unary operations $s1$ and $s2$. Moreover the class of models of CONCRETESERVICE shrinks with respect to the model classes of ABSTRACTSERVICE; and this is due to the axiom $\neg(r3 = r4)$.

On the process side, the internal non-determinism is resolved by adding a scheduling mechanism. On the CSP traces model, this means that the trace set of CONCRETESERVICE is included in the trace set of ABSTRACTSERVICE. ■

Intuitively a CSP-CASL refinement describes the following development process: On the data part, the possible interpretations (or model classes) are reduced by adding new informations about the data. The refined model classes are then used to construct the alphabet of communications for the process behavior. On the process part, the refined process

description is less internally non-deterministic. That is, the environment in which the process is defined has more control of the process.

In Figure 6.1 we summarize the overall idea of CSP-CASL refinement that we would like to define. Here, the circles denote the model classes of the two data signatures Σ and Σ'. The small circle (I') is included in the bigger circle (I) after applying the reduct to I'. This denotes the inclusion of the model classes, thus *data refinement*.

CSP-CASL semantics constructs the alphabet of communication. Here, every model $M' \in I'$ and $M'|_\sigma$ gives rise to an alphabet $\mathcal{A}lph(M')$ and $\mathcal{A}lph(M'|_\sigma)$. Those are used to construct the CSP semantic domain $\mathcal{D} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$ (see Section 2.2); hence we obtain $\mathcal{D}(\mathcal{A}lph(M'))$ and $\mathcal{D}(\mathcal{A}lph(M'|_\sigma))$. We need to define a mapping $\alpha_\mathcal{D} : \mathcal{D}(\mathcal{A}lph(M'|_\sigma)) \to \mathcal{D}(\mathcal{A}lph(M'))$, and an inverse mapping $\hat{\alpha}_\mathcal{D} : \mathcal{D}(\mathcal{A}lph(M')) \to \mathcal{D}(\mathcal{A}lph(M'|_\sigma))$. The latter represents a reduct definition over the process denotations.

Let $d'_{M'} \in \mathcal{D}(\mathcal{A}lph(M'))$ and $d'_{M'|_\sigma} \in \mathcal{D}(\mathcal{A}lph(M'|_\sigma))$ be two process denotations. In order to "compare" these two process denotations for refinement, we apply an inverse mapping $\hat{\alpha}_\mathcal{D}$ to $d'_{M'}$. Finally, we compute the *process refinement* on these objects – $d_{M'|_\sigma} \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d'_{M'})$.

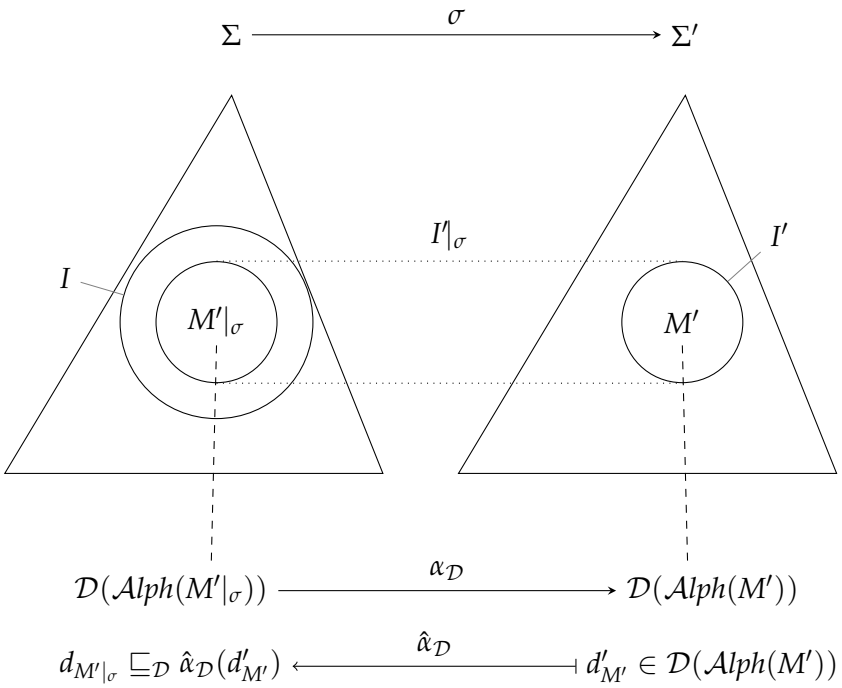

Figure 6.1: CSP-CASL refinement with change of signature.

From the above illustration, there are a number of concepts that we need to address; some of them are:

- definition of signature morphism σ for CSP-CASL,
- proof that the mapping $\alpha_\mathcal{D}$ is well defined and injective, and

- proof that the inverse mapping $\hat{\alpha}_{\mathcal{D}}$ preserves the healthiness condition of the semantical domain.

We start by giving a formal definition of the CSP-CASL data logic signature morphism, following [Rog06], however with a slight modification.

DEFINITION 6.1.2 *Let $\Sigma = (S, TF, PF, P, \leq)$ and $\Sigma' = (S', TF', PF', P', \leq')$ be two* CASL *data logic signatures. A* CSP-CASL *data logic signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ is a many-sorted signature morphism that preserves the subsort relation and the overloading relations, i.e., for σ holds:*

p1 *$s_1 \leq s_2$ implies $\sigma^S(s_1) \leq \sigma^S(s_2)$ for all $s_1, s_2 \in S$*

p2 *$f : w_1 \rightarrow s_1 \sim_F f : w_2 \rightarrow s_2$ implies $\sigma^F_{w_1,s_1}(f) = \sigma^F_{w_2,s_2}(f)$ for all $f \in TF \cup PF$*

p3 *$p : w_1 \sim_P p : w_2$ implies $\sigma^P_{w_1}(p) = \sigma^P_{w_2}(p)$ for all $p \in P$*

refl *$\sigma^S(s_1) \leq \sigma^S(s_2)$ implies $s_1 \leq_S s_2$ for all s_1, $s_2 \in S$ (reflection of the subsort relation) and*

weak non-extension *$s_1 \neq s_2$ and $\sigma^S(s_1) \leq_{S'} u'$ and $\sigma^S(s_2) \leq_{S'} u'$ implies that there exist a sort $t \in S$ with $s_1 \leq t, s_2 \leq t$ and $\sigma^S(t) \leq u'$.*

See Chapter 3 for details about the overloading relations. While the first three conditions come from the CASL subsorted signature morphism, the conditions *refl* and *weak non-extension* are required by CSP-CASL construction. The *refl* condition allows to reflect the subsort relation after the signature morphism. The *weak non-extension* condition allows to make sure that the subsort relation is extended, however, such extension must be restricted. Note that we differ here from the original definition as given in [Rog06]. The weak non extension property by [Rog06] implies the one given here, i.e., here we are more general.

Let us consider the following two examples of data specification:

spec D =
  sort S, T
  ops $a : S$; $b : T$
end

spec D′ =
  sort $S, T \leq U$
  ops $a : S$; $b : T$
end

In D the CSP-CASL alphabet construction yields that in all Σ-models M for the corresponding events $\overline{M(a)} \neq \overline{M(b)}$ hold. However, in D′, where S and T have a common supersort, it depends on the model M whether $\overline{M(a)} = \overline{M(b)}$ or $\overline{M(a)} \neq \overline{M(b)}$. The *weak non- extension* condition ensure that the alphabet transformation over the CSP-CASL data logic signature morphism is injective and well-defined.

In [MR07] it is shown that injectivity of alphabet translation is necessary in making sure that the CSP process properties are preserved after the translation.

The definition of CSP-CASL data logic signature morphism restricts how we can extend the data signature. The restrictions are chosen in a way that the imposed alphabet translation

behave properly. If we consider the 2-step semantics of CSP-CASL defined in Section 4.2, we obtain the picture shown in Figure 6.2.

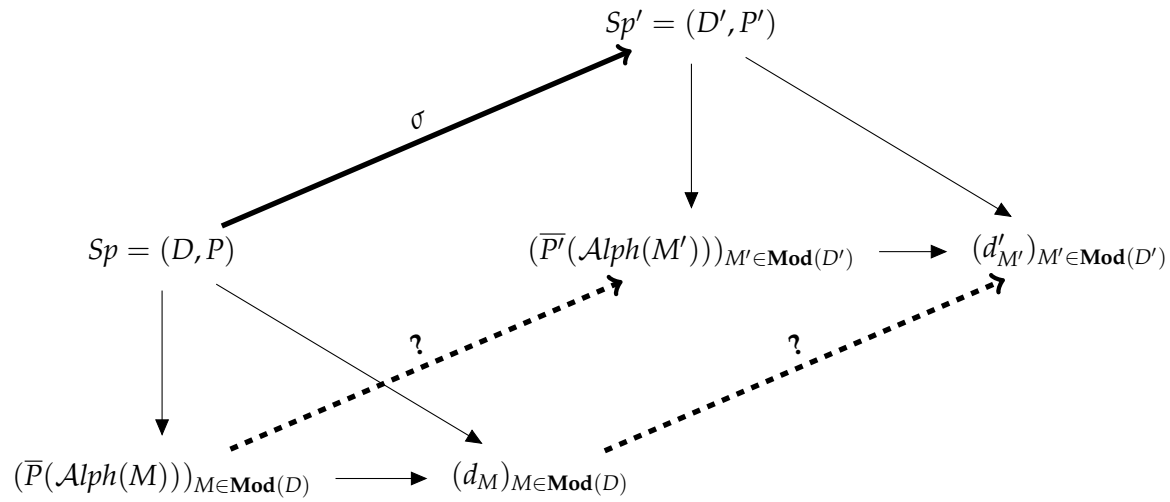

Figure 6.2: Property preserving translation.

Let us for instance consider the following two simple CASL specifications.

spec OLD =
  sort T
end

spec NEW =
  sort $Q \leq U$
end

Let Σ and Σ' be the signature of OLD and NEW. Let $\sigma : \Sigma \to \Sigma'$ be a CSP-CASL data logic signature morphism, such that $\sigma^S(T) = Q$. Let M' be a Σ'-model such that $M'(Q) = \{1, 2\}$ and $M'(U) = \{3, 4, 5, 6\}$, with the following injection $\mathtt{inj}_{(Q,U)}(1) = 3$ and $\mathtt{inj}_{(Q,U)}(2) = 4$. In this setting, we have $M'|_\sigma\ (T) = \{1, 2\}$. The alphabet construction over $M'|_\sigma$ and M' yields:

$$\mathcal{A}lph(\beta(M'\ |_\sigma)) = \{[(T, 1)], [(T, 2)], [(T, \bot)]\}$$
$$\mathcal{A}lph(\beta(M')) = \{[(Q, 1), (U, 3)], [(Q, 2), (U, 4)], [(U, 5)], [(U, 6)], [(Q, \bot), (U, \bot)]\}.$$

Here, we can observe that $\mathcal{A}lph(\beta(M'))$ contains more and different symbols. Our aim is to define a translation from $\mathcal{A}lph(\beta(M'\ |_\sigma))$ to $\mathcal{A}lph(\beta(M'))$ which is injective. In the following we illustrate that an alphabet translation along a CSP-CASL data logic signature morphism is well defined and injective.

LEMMA 6.1.3 *Let $\sigma : \Sigma \to \Sigma'$ be a* CSP-CASL *data logic signature morphism. Let M' be a Σ'-model. Then*

$$\alpha : \begin{array}{l} \mathcal{A}lph(M'\ |_\sigma) \to \mathcal{A}lph(M') \\ [(s, x)]_{\sim_{M'|_\sigma}} \mapsto [(\sigma^S(s), x)]_{\sim_{M'}} \end{array}$$

is well-defined and injective. Furthermore, we have for all $s \in S$

$$\alpha(\overline{\beta(M' \mid_\sigma)(s)}) = \overline{\beta(M')(\sigma^S(s))}$$

where $\overline{\beta(M)(s)} := \{[(s,x)]_{\sim_M} \mid x \in \beta(M)\}$*, i.e.,* $\beta(M)$ *in which partial functions of* M *are totalized.*

PROOF. We now prove the *well-definedness* property of the alphabet transformation. We show that if $(s,x) \sim_{M'|_\sigma} (t,y)$ then $(\sigma^S(s),x) \sim_{M'} (\sigma^S(t),y)$.

Let $(s,x) \sim_{M'|_\sigma} (t,y)$. Following the definition of $\sim_{M'|_\sigma}$ defined in Section 4.2, there are two cases to consider:

Case 1. $[x = y = \bot]$. As $(s,x) \sim_{M'|_\sigma} (t,y)$ holds, then there exists $u \in S$ with $s \leq u$ and $t \leq u$. Thanks to **p1** we obtain $\sigma^S(s) \leq' \sigma^S(u)$ and $\sigma^S(t) \leq' \sigma^S(u)$.

Case 2. $[x \neq \bot, y \neq \bot]$. We need to show that the following two conditions hold:

1. $\exists\, u' \in S'$ such that $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$.
2. for all $u' \in S'$ with $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$ the following holds:

$$(inj_{(\sigma^S(s),u')})_{M'}(x) = (inj_{(\sigma^S(t),u')})_{M'}(y)$$

For condition 1, the proof is identical to the one under Case 1.

For condition 2, we consider the situation in which $s = t$ and $s \neq t$.

- In the case of $s = t$, we set $u = s$ and obtain:

  $$(inj_{(s,s)})_{M'|_\sigma}(x) = (inj_{(s,s)})_{M'|_\sigma}(y)$$

  This has as a consequence that $x = y$. Consequently we obtain

  $$(inj_{(\sigma^S(s),u')})_{M'}(x) = (inj_{(\sigma^S(s),u')})_{M'}(y).$$

- In the case of $s \neq t$, we show that for all $u' \in S'$ with $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$, the following holds:

  $$(inj_{(\sigma^S(s),u')})_{M'}(x) = (inj_{(\sigma^S(t),u')})_{M'}(y) \qquad (**)$$

  Let $u' \in S'$ with $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$. Using the *weak non extension* we know that there exists $v \in S$ such that $s \leq v$, $t \leq v$ and $\sigma^S(t) \leq' u'$. For s, t and v we know that

  $$(inj_{(s,v)})_{M'|_\sigma}(x) = (inj_{(t,v)})_{M'|_\sigma}(y).$$

  Thus,

  $$(inj_{(\sigma^S(s),\sigma^S(v))})_{M'}(x) = (inj_{(\sigma^S(t),\sigma^S(v))})_{M'}(y).$$

  From the third axiom of *inj* [1], it follows:

  $$(inj_{(\sigma^S(s),u')})_{M'}(x) = (inj_{(\sigma^S(t),u')})_{M'}(y).$$

[1] $(inj_{s',s''})_M((inj_{s,s'})_M(x)) = (inj_{s,s''})_M$ for $x \in M_s, s \leq s' \leq s''$.

For the *injectivity* proof, we show that if $[(\sigma^S(s), x)]_{\sim_{M'}} = [(\sigma^S(t), y)]_{\sim_{M'}}$, then $[(s, x)]_{\sim_{M'|_\sigma}} = [(t, y)]_{\sim_{M'|_\sigma}}$, i.e.,

$$(\sigma^S(s), x) \sim_{M'} (\sigma^S(t), y) \Rightarrow (s, x) \sim_{M'|_\sigma} (t, y).$$

Again, following the definition of $\sim_{M'}$ there are two cases to consider:

Case 1 [$x = y = \bot$]. Here, we consider the situation in which $s = t$ and $s \neq t$.

- In the case of $s = t$, its enough to set $u = s$.
- In the case $s \neq t$, we use the *weak non-extension property*. Let $(\sigma^S(s), \bot) \sim_{M'} (\sigma^S(t), \bot)$. Then there exists a $u' \in S'$ such that $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$. Then there exists a $v \in S$ such that $s \leq v$ and $t \leq v$. Thus, $(s, \bot) \sim_{M'|_\sigma} (t, \bot)$.

Case 2 [$x \neq \bot, y \neq \bot$]. Let $(\sigma^S(s), x) \sim_{M'} (\sigma^S(t), y)$. Then we know that $\exists\, u' \in S'$ such that $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$, and for all $u' \in S'$ with $\sigma^S(s) \leq' u'$ and $\sigma^S(t) \leq' u'$ the following holds:

$$(inj_{(\sigma^S(s), u')})_{M'}(x) = (inj_{(\sigma^S(t), u')})_{M'}(y).$$

We show that the following two conditions hold:

1. $\exists\, u \in S$ such that $s \leq u$ and $t \leq u$, and
2. for all $u \in S$ with $s \leq u$ and $t \leq u$ the following holds:

$$(inj_{(s,u)})_{M'|_\sigma}(x) = (inj_{(t,u)})_{M'|_\sigma}(y).$$

In order to prove condition 1, we consider the situations in which $s = t$ and $s \neq t$.

- In the case of $s = t$ its enough to set $u = s$.
- In the case of $s \neq t$, we use the *weak non-extension* property. As there exists $u' \in S'$ such that $\sigma^S(s) \leq' u'$, $\sigma^S(t) \leq' u'$ and $s \neq t$, there exists $v \in S$ such that $s \leq v$ and $t \leq v$.

For condition 2, we show that for all $u \in S$ with $s \leq u$ the following holds:

$$(inj_{(s,u)})_{M'|_\sigma}(x) = (inj_{(t,u)})_{M'|_\sigma}(y).$$

Let $u \in S$ with $s \leq u$. Here, we apply the model reduct definition on both sides:

$$\begin{array}{ll} & (inj_{(s,u)})_{M'|_\sigma}(x) = (inj_{(t,u)})_{M'|_\sigma}(y) \\ \iff & (\sigma^F(inj_{(s,u)}))_{M'}(x) = (\sigma^F(inj_{(t,u)}))_{M'}(y) \\ \iff & (inj_{(\sigma^S(s), \sigma^S(u))})_{M'}(x) = (inj_{(\sigma^S(t), \sigma^S(u))})_{M'}(y). \end{array}$$

Thanks to the preservation property of CSP-CASL data logic signature morphism σ, it follows that $\sigma^S(s) \leq' \sigma^S(u)$. We also know from $\sim_{M'}$ that:

$$(inj_{(\sigma^S(s), \sigma^S(u))})_{M'}(x) = (inj_{(\sigma^S(t), \sigma^S(u))})_{M'}(y)$$

Hence, we obtain $(inj_{(s,u)})_{M'|_\sigma}(x) = (inj_{(t,u)})_{M'|_\sigma}(y)$.

■

We extend the map α canonically to four maps in the following way.

- To include the termination symbol ✓. $\alpha^{\checkmark} : \mathcal{A}lph(M'|_\sigma)^{\checkmark} \to \mathcal{A}lph(M')^{\checkmark}$, defined as:

$$a \mapsto \begin{cases} \alpha(a) & \text{if } a \in \mathcal{A}lph(M'|_\sigma) \\ \checkmark & \text{if } a = \checkmark \end{cases}$$

  Here, $\mathcal{A}lph(M)^{\checkmark} = \mathcal{A}lph(M) \cup \{\checkmark\}$.

- To extend it to strings $\alpha^* : \mathcal{A}lph(M'|_\sigma)^* \to \mathcal{A}lph(M')^*$, defined as:

$$\begin{array}{lcl} \alpha^*(\langle\rangle) & \mapsto & \langle\rangle \\ \alpha^*(a \frown t) & \mapsto & \alpha(a) \frown \alpha^*(t) \end{array}$$

  where $\langle\rangle$ is the empty string and $a \frown t$ is the concatenation of a with the string t.

- To extend it to strings and termination symbol $\alpha^{*\checkmark}$: $\mathcal{A}lph(M'|_\sigma)^{*\checkmark} \to \mathcal{A}lph(M')^{*\checkmark}$, defined as:

$$\begin{array}{l} \alpha^{*\checkmark}(s) = \alpha^*(s) \\ \alpha^{*\checkmark}(s \frown \langle\checkmark\rangle) = \alpha^*(s) \frown \langle\checkmark\rangle \end{array}$$

  Where $s \in \mathcal{A}lph(M'|_\sigma)^*$. Here, $\mathcal{A}lph(M)^{*\checkmark} = \mathcal{A}lph(M)^* \cup \{s \frown \langle\checkmark\rangle \mid s \in \mathcal{A}lph(M)^*\}$.

- To extend it to the power domain, $\alpha^{\checkmark}_{\mathbb{P}} : \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^{\checkmark}) \to \mathbb{P}(\mathcal{A}lph(M')^{\checkmark})$, defined as:

$$X \mapsto \{\alpha^{\checkmark}(x) \mid x \in X\}.$$

- Finally, to apply it to elements of the semantical domains:

$$\alpha_{\mathcal{D}} : \mathcal{D}(\mathcal{A}lph(M'|_\sigma)) \to \mathcal{D}(\mathcal{A}lph(M'))$$

  where $\mathcal{D}$ is one of the CSP semantic models studied in our context, i.e., $\mathcal{D} \subseteq \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$.

  - In the traces model; $\alpha_{\mathcal{T}} : \mathcal{T}(\mathcal{A}lph(M'|_\sigma)) \to \mathcal{T}(\mathcal{A}lph(M'))$, defined as:

$$T_{M'|_\sigma} \mapsto \{\alpha^{*\checkmark}(t) \mid t \in T_{M'|_\sigma}\}.$$

  - In the stable failure model; $\alpha_{\mathcal{F}} : \mathcal{F}(\mathcal{A}lph(M'|_\sigma)) \to \mathcal{F}(\mathcal{A}lph(M'))$, defined in the following way: Let $(T_{M'|_\sigma}, F_{M'|_\sigma}) \in \mathcal{F}(M'|_\sigma)$. We define how the translation goes for the single failures: $\alpha(t, X) = (\alpha^{*\checkmark}(t), \alpha^{\checkmark}_{\mathbb{P}}(X))$ for $(t, X) \in F_{M'|_\sigma}$, then

$$\alpha_{\mathcal{F}}(T_{M'|_\sigma}, F_{M'|_\sigma}) = (\alpha_{\mathcal{T}}(T_{M'|_\sigma}), \{(\alpha^{*\checkmark}(t), \alpha^{\checkmark}_{\mathbb{P}}(X)) \mid (t, X) \in F_{M'|_\sigma}\}).$$

  - In the failures/divergences model; $\alpha_{\mathcal{N}} : \mathcal{N}(\mathcal{A}lph(M'|_\sigma)) \to \mathcal{N}(\mathcal{A}lph(M'))$, defined in the following way: Let $(F^{\perp}_{M'|_\sigma}, D_{M'|_\sigma}) \in \mathcal{N}(\mathcal{A}lph(M'|_\sigma))$. We define how the translation goes for the single failures: $\alpha(t, X) = (\alpha^{*\checkmark}(t), \alpha^{\checkmark}_{\mathbb{P}}(X))$ for $(t, X) \in F^{\perp}_{M'|_\sigma}$, then

$$\alpha_{\mathcal{N}}(F^{\perp}_{M'|_\sigma}, D_{M'|_\sigma}) = (\{(\alpha^{*\checkmark}(t), \alpha^{\checkmark}_{\mathbb{P}}(X)) \mid (t, X) \in F_{M'|_\sigma}\}, \{\alpha^{*\checkmark}(d) \mid d \in D_{M'|_\sigma}\}).$$

While α allow us to translate the alphabet generated by a model M of an abstract data specification to an alphabet generated by a refined model M', we need to define also a mapping that goes the other way around. Given an injective alphabet translation $\alpha : \mathcal{A}lph(M'|_\sigma) \to \mathcal{A}lph(M')$ we define the partial inverse

$$\hat{\alpha} : \mathcal{A}lph(M') \to ? \ \mathcal{A}lph(M'|_\sigma)$$

as:

$$[(\sigma^S(s), x)]_{\sim_{M'}} \quad \mapsto \quad \begin{cases} [(s,x)]_{\sim_{M'|_\sigma}} ; & \text{if } [(s,x)]_{\sim_{M'|_\sigma}} \in \mathcal{A}lph(M'|_\sigma) \\ & \text{such that } \alpha([(s,x)]_{\sim_{M'|_\sigma}}) = [\sigma^S(s), x)]_{\sim_{M'}} \\ \textit{undefined}; & \text{otherwise} \end{cases}$$

In the same way as the alphabet transformation α we extend the inverse translation $\hat{\alpha}$ to four maps:

- To include the termination symbol $\checkmark$. $\hat{\alpha}^\checkmark : \mathcal{A}lph(M')^\checkmark \to ? \ \mathcal{A}lph(M'|_\sigma)^\checkmark$, defined as:

$$a' \quad \mapsto \quad \begin{cases} \hat{\alpha}(a') & \text{if } \hat{\alpha}(a') \text{ is defined} \\ \checkmark & \text{if } a' = \checkmark \\ \textit{undefined} & \text{otherwise} \end{cases}$$

- To extend it to strings $\hat{\alpha}^* : \mathcal{A}lph(M')^* \to ? \ \mathcal{A}lph(M'|_\sigma)^*$, defined as:

$$\begin{array}{lcl} \hat{\alpha}^*(\langle\rangle) & \mapsto & \langle\rangle \\ \hat{\alpha}^*(a' \frown t') & \mapsto & \begin{cases} \hat{\alpha}(a') \frown \hat{\alpha}^*(t) & \text{if } \hat{\alpha}(a') \text{ and } \hat{\alpha}(t') \text{ are defined} \\ \textit{undefined} & \text{otherwise} \end{cases} \end{array}$$

- To extend it to strings and termination symbol; $\hat{\alpha}^{*\checkmark} : \mathcal{A}lph(M')^{*\checkmark} \to ? \ \mathcal{A}lph(M'|_\sigma)^{*\checkmark}$, defined as:

$$\begin{aligned} \hat{\alpha}^{*\checkmark}(s) &= \hat{\alpha}^*(s) \\ \hat{\alpha}^{*\checkmark}(s \frown \langle\checkmark\rangle) &= \hat{\alpha}^*(s) \frown \langle\checkmark\rangle \end{aligned}$$

  for $s \in \mathcal{A}lph(M')^*$.

- To extend it to the power domain, $\hat{\alpha}^\checkmark_{\mathbb{P}} : \mathbb{P}(\mathcal{A}lph(M')^\checkmark) \to ? \ \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark)$, defined as:

$$X \mapsto \{x \in \mathcal{A}lph(M'|_\sigma)^\checkmark \mid \alpha^\checkmark(x) \in X\}.$$

- Finally, to apply it to elements of the semantical domain:

$$\hat{\alpha}_\mathcal{D} : \mathcal{D}(\mathcal{A}lph(M')) \to ? \ \mathcal{D}(\mathcal{A}lph(M'|_\sigma))$$

  where $\mathcal{D}$ is one of the CSP semantic model studied in our context, i.e., $\mathcal{D} \subseteq \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$.

  – In the traces model; $\hat{\alpha}_\mathcal{T} : \mathcal{T}(\mathcal{A}lph(M')) \to ? \ \mathcal{T}(\mathcal{A}lph(M'|_\sigma))$ is defined as:

  $$\hat{\alpha}_\mathcal{T}(T_{M'}) = \{t \in \mathcal{A}lph(M'|_\sigma)^{*\checkmark} \mid \alpha^{*\checkmark}(t) \in T_{M'}\}$$

– In the stable failure model; $\hat{\alpha}_{\mathcal{F}} : \mathcal{F}(\mathcal{A}lph(M')) \rightarrow? \mathcal{F}(\mathcal{A}lph(M'|_\sigma))$, defined as:

$$\begin{aligned}\hat{\alpha}_{\mathcal{F}}(T_{M'}, F_{M'}) = (&\{t \in \mathcal{A}lph(M'|_\sigma)^{*\checkmark} \mid \alpha^{*\checkmark}(t) \in T_{M'}\},\\ &\{(t, X) \in \mathcal{A}lph(M'|_\sigma)^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^{\checkmark}) \mid\\ &\text{exists } (t', X') \in F_{M'} \text{ with } \alpha^{*\checkmark}(t) = t' \text{ and } \alpha^{\checkmark}_{\mathbb{P}}(X) = X' \cap \alpha(\mathcal{A}lph(M'|_\sigma))\})\end{aligned}$$

– In the failures/divergences model; $\hat{\alpha}_{\mathcal{N}} : \mathcal{N}(\mathcal{A}lph(M')) \rightarrow? \mathcal{N}(\mathcal{A}lph(M'|_\sigma))$, defined as:

$$\begin{aligned}\hat{\alpha}_{\mathcal{N}}(F^{\perp}_{M'}, D_{M'}) = &\{(t, X) \in \mathcal{A}lph(M'|_\sigma)^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^{\checkmark}) \mid\\ &\text{exists } (t', X') \in F^{\perp}_{M'} \text{ with } \alpha^{*\checkmark}(t) = t' \text{ and } \alpha^{\checkmark}_{\mathbb{P}}(X) = X' \cap \alpha(\mathcal{A}lph(M'|_\sigma))\},\\ &\{d \in \mathcal{A}lph(M'|_\sigma)^* \mid \alpha^*(d) \in D_{M'}\}).\end{aligned}$$

The definition of the inverse map in the three semantical domains ($\hat{\alpha}_{\mathcal{T}}, \hat{\alpha}_{\mathcal{F}}, \hat{\alpha}_{\mathcal{N}}$) defines the notion of reduct of process denotation.

Note that our definitions subtly differ from the concept of eager abstraction and lazy abstraction as discussed, e.g., in [Ros98]. Eager and lazy abstractions hide the new events in all traces – our approach, however, ignores traces that include new events.

For the definition of the process reduct, we need to prove that such translation is well behaved. That is the healthiness condition of the various semantical model are still valid after the translation. This is necessary when the signature of the data part changes in a refinement step we have to make sure that the semantics of the processes behaves well. For instance, when moving to a larger alphabet of communications the processes defined before the refinement step may only use the "old" alphabet letters. In the next three lemmas, we show that the inverse translation $\hat{\alpha}_{\mathcal{T}}$, $\hat{\alpha}_{\mathcal{F}}$ and $\hat{\alpha}_{\mathcal{N}}$ preserve the healthiness conditions of the traces ($\mathcal{T}$), stable failure ($\mathcal{F}$) and failures/divergences ($\mathcal{N}$) model respectively.

In the following lemmas, let $\sigma : \Sigma \rightarrow \Sigma'$ be the CSP-CASL data logic signature morphism and $M' \in \mathbf{Mod}(D')$ be the model of the data part D'.

LEMMA 6.1.4 *Over the traces model $\mathcal{T}$ the following holds:*

$$T' \in \mathcal{T}(\mathcal{A}lph(M')) \Rightarrow \hat{\alpha}_{\mathcal{T}}(T') \in \mathcal{T}(\mathcal{A}lph(M'|_\sigma)).$$

PROOF. We prove that $\hat{\alpha}_{\mathcal{T}} : \mathcal{T}(\mathcal{A}lph(M')) \rightarrow? \mathcal{T}(\mathcal{A}lph(M'|_\sigma))$ preserves the healthiness conditions of the traces model $\mathcal{T}$.

T.1 We show that $\hat{\alpha}_{\mathcal{T}}(T') \in \mathcal{T}(\mathcal{A}lph(M'|_\sigma))$ is non empty and prefix closed. We know that $\langle\rangle \in T'$. We have that $\hat{\alpha}^*(\langle\rangle) = \langle\rangle$; it follows $\langle\rangle \in \hat{\alpha}_{\mathcal{T}}(T')$.

Let $T = \hat{\alpha}_{\mathcal{T}}(T')$. Let $t \in T$, then there exists $t' \in T'$ with $\hat{\alpha}^*(t') = t$. Let $s \leq t$, then $\alpha^*(s) \leq \alpha^*(t)$ as T' is prefixed closed i.e., $\alpha^*(s) \in T'$. Thus, $\hat{\alpha}^*(\alpha^*(s)) \in T$, i.e., $\hat{\alpha}^*(\alpha^*(s)) = s$. Hence, $s \in T$.

■

LEMMA 6.1.5 *Over the stable failure model $\mathcal{F}$ the following holds:*

$$(T', F') \in \mathcal{F}(\mathcal{A}lph(M')) \Rightarrow \hat{\alpha}_{\mathcal{F}}(T', F') \in \mathcal{F}(\mathcal{A}lph(M'|_\sigma)).$$

PROOF. We prove that $\hat{\alpha}_{\mathcal{F}} : \mathcal{F}(\mathcal{A}lph(M')) \rightarrow? \mathcal{F}(\mathcal{A}lph(M'|_\sigma))$ preserves the healthiness conditions of the stable failure model $\mathcal{F}$.

T.1 $\hat{\alpha}_{\mathcal{T}}(T')$ is non-empty and prefix closed by Lemma 6.1.4.

T.2 Let $(T', F') \in \mathcal{F}(\mathcal{A}lph(M'))$. Let $(s, X) \in \hat{\alpha}(F')$. We show that $s \in \hat{\alpha}_{\mathcal{T}}(T')$.

As $(s, X) \in \hat{\alpha}(F')$, there exists $(s', X') \in F'$ such that $\hat{\alpha}^*(s') = s$, $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(X') = X$. We know that $\mathcal{F}(\mathcal{A}lph(M'))$ fulfills **T.2**, i.e., $(s', X') \in F'$ implies $s' \in T'$. It follows that $\hat{\alpha}^*(s') = s \in \hat{\alpha}_{\mathcal{T}}(T')$. Thus, $\hat{\alpha}_{\mathcal{F}}(\mathcal{F}(\mathcal{A}lph(M')))$ fulfills **T.2**.

T.3 Let $(T', F') \in \mathcal{F}(\mathcal{A}lph(M'))$. Let $s \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(T')$. We show that $(s \frown \langle\checkmark\rangle, X) \in \hat{\alpha}(F')$ for all $X \subseteq \hat{\alpha}(\mathcal{A}lph(M')^{\checkmark})$.

As $s \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(T')$, there exists $(s' \frown \langle\checkmark\rangle) \in T'$ such that $\hat{\alpha}^{*\checkmark}(s' \frown \langle\checkmark\rangle) = s \frown \langle\checkmark\rangle$. We know that $\mathcal{F}(\mathcal{A}lph(M'))$ fulfills **T.3**, i.e., if $s' \frown \langle\checkmark\rangle \in T'$ implies $(s' \frown \langle\checkmark\rangle, X') \in F'$ for all $X' \subseteq \mathcal{A}lph(M')^{\checkmark}$. Then, it follows that $(s \frown \langle\checkmark\rangle, \hat{\alpha}^{\checkmark}_{\mathbb{P}}(X')) \in \hat{\alpha}(F')$ for all $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(X') \subseteq \hat{\alpha}(\mathcal{A}lph(M')^{\checkmark})$. Hence, $\hat{\alpha}_{\mathcal{F}}(\mathcal{F}(\mathcal{A}lph(M')))$ fulfills **T.3**.

F.2 Let $(T', F') \in \mathcal{F}(\mathcal{A}lph(M'))$. Let $(s, X) \in \hat{\alpha}(F')$ and $Y \subseteq X$. We show that $(s, Y) \in \hat{\alpha}(F')$.

As $(s, X) \in \hat{\alpha}(F')$, there exists $(s', X') \in F'$ such that $\hat{\alpha}^*(s') = s$ and $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(X') = X$. We know that $\mathcal{F}(\mathcal{A}lph(M'))$ fulfills **F.2**, i.e., if $(s', X') \in F'$ and $Y' \subseteq X'$ then $(s', Y') \in F'$.

Let $Y \subseteq \hat{\alpha}^{\checkmark}_{\mathbb{P}}(X')$, then there exists $Y' \subseteq X'$ with $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(Y') = Y$. It follows that $(s, Y) \in \hat{\alpha}(F')$. Hence, $\hat{\alpha}_{\mathcal{F}}(\mathcal{F}(\mathcal{A}lph(M')))$ fulfills **F.2**.

F.3 Let $(T', F') \in \mathcal{F}(\mathcal{A}lph(M'))$. Let $(s, X) \in \hat{\alpha}(F')$ and $\forall\, a \in Y : s \frown \langle a\rangle \notin \hat{\alpha}_{\mathcal{T}}(T')$. We show that $(s, X \cup Y) \in \hat{\alpha}(F')$.

As $(s, X) \in \hat{\alpha}(F')$, then there exists $(s', X') \in F'$, such that $\hat{\alpha}^*(s') = s$ and $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(X') = X$. We know that $\mathcal{F}(\mathcal{A}lph(M'))$ fulfills the **F.3**, i.e., if $(s', X') \in F'$ and $\forall\, a' \in Y' : s' \frown \langle a'\rangle \notin T'$ then $(s', X' \cup Y') \in F'$.

Let $\forall\, a \in Y : s \frown \langle a\rangle \notin \hat{\alpha}_{\mathcal{T}}(T')$, there exists Y' such that $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(Y') = Y$. It follows that $(s, X \cup Y) \in \hat{\alpha}(F')$. Hence, $\hat{\alpha}_{\mathcal{F}}(\mathcal{F}(\mathcal{A}lph(M')))$ fulfills **F.3**.

F.4 Let $(T', F') \in \mathcal{F}(\mathcal{A}lph(M'))$. Let $s \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(T')$. We show that $(s, \mathcal{A}lph(M')) \in \hat{\alpha}(F')$.

As $s \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(T')$, then there exists $s' \frown \langle\checkmark\rangle \in T'$ such that $\hat{\alpha}^{*\checkmark}(s' \frown \langle\checkmark\rangle) = s \frown \langle\checkmark\rangle$. We know that $\mathcal{F}(\mathcal{A}lph(M'))$ fulfills the **F.4**, i.e., if $s' \frown \langle\checkmark\rangle \in F'$ then $(s' \frown \langle\checkmark\rangle, \mathcal{A}lph(M')) \in F'$. Then it follows that $(\hat{\alpha}^*(s'), \hat{\alpha}(\mathcal{A}lph(M'))) = (s, \mathcal{A}lph(M'|_\sigma)) \in \hat{\alpha}(F')$. Hence, $\hat{\alpha}_{\mathcal{F}}(\mathcal{F}(\mathcal{A}lph(M')))$ fulfills **F.4**.

■

LEMMA 6.1.6 *Over the failures/divergences model $\mathcal{N}$ the following holds:*

$$(F'^{\perp}, D') \in \mathcal{N}(\mathcal{A}lph(M')) \Rightarrow \hat{\alpha}_{\mathcal{N}}(F'^{\perp}, D') \in \mathcal{N}(\mathcal{A}lph(M'|_\sigma)).$$

PROOF. We show that $\hat{\alpha}_{\mathcal{N}}$ preserves the healthiness conditions of the failures/divergences model $\mathcal{N}$.

F.1, F.2, **F.3** and **F.4**. See the proofs of Lemma 6.1.5.

D.1 Let $(F'^{\perp}, D') \in \mathcal{N}(Alph(M'))$. Let $s \in \hat{\alpha}(D') \cap \hat{\alpha}(Alph(M')^*)$ and $t \in \hat{\alpha}(Alph(M')^{*\checkmark})$. We show that $s \frown t \in \hat{\alpha}(D')$.

As $s \in \hat{\alpha}(D') \cap \hat{\alpha}(Alph(M')^*$ and $t \in \hat{\alpha}(Alph(M')^{*\checkmark})$, there exists $s' \in D' \cap Alph(M')^*$ and $t' \in Alph(M')^{*\checkmark}$ such that $\hat{\alpha}^*(s') = s$ and $\hat{\alpha}^{*\checkmark}(t') = t$. We know that $\mathcal{N}(Alph(M'))$ fulfills **D.1**, i.e., if $s' \in D' \cap Alph(M')^*$ and $t' \in Alph(M')^{*\checkmark}$ then $s' \frown t' \in D'$. It follows that $\hat{\alpha}^*(s') \frown \hat{\alpha}^*(t') = s \frown t \in \hat{\alpha}(D')$. Hence, $\hat{\alpha}_{\mathcal{N}}(\mathcal{N}(Alph(M')))$ fulfills **D.1**.

D.2 Let $(F'^{\perp}, D') \in \mathcal{N}(Alph(M'))$. Let $s \in \hat{\alpha}(D')$. We show that $(s, X) \in \hat{\alpha}(F')$.

As $s \in \hat{\alpha}(D')$, then there exists $s' \in D'$ such that $\hat{\alpha}^*(s') = s$. We know that $\mathcal{N}(Alph(M'))$ fulfills **D.2**, i.e., if $s' \in D'$ then $(s', X') \in F'$. Then it follows that $(\hat{\alpha}^*(s'), \hat{\alpha}^{\checkmark}_{\mathbb{P}}(X')) = (s, X) \in \hat{\alpha}(F')$. Hence, $\hat{\alpha}_{\mathcal{N}}(\mathcal{N}(Alph(M')))$ fulfills **D.2**.

D.3 Let $(F'^{\perp}, D') \in \mathcal{N}(Alph(M'))$. Let $s \frown \langle\checkmark\rangle \in \hat{\alpha}(D')$. We show that then $s \in \hat{\alpha}(D')$.

As $s \frown \langle\checkmark\rangle \in \hat{\alpha}(D')$, then there exists $s' \frown \langle\checkmark\rangle \in D'$ such that $\hat{\alpha}^{*\checkmark}(s' \frown \langle\checkmark\rangle) = s \frown \langle\checkmark\rangle$. We know that $\mathcal{N}(Alph(M'))$ fulfills **D.3**, i.e., if $s' \frown \langle\checkmark\rangle \in D'$ then $s' \in D'$. Then it follows that $\hat{\alpha}^*(s') = s \in D$. Hence, $\hat{\alpha}_{\mathcal{N}}(\mathcal{N}(Alph(M')))$ fulfills **D.3**.

■

We now define the translation of CSP processes operators on the syntactical level.

DEFINITION 6.1.7 (PROCESS TRANSLATION) *Let $\sigma : \Sigma(D) \to \Sigma(D')$ be a* CSP-CASL *data logic signature morphism. We define ρ to denote the translation of process operators defined as follows:*

$$
\begin{array}{lcl}
\rho(STOP) & := & STOP \\
\rho(SKIP) & := & SKIP \\
\rho(DIV) & := & DIV \\
\rho(t \to P) & := & \sigma(t) \to \rho(P) \\
\rho(?x :: s \to P) & := & ?x :: \sigma(s) \to \rho(P) \\
\rho(!x :: s \to P) & := & !x :: \sigma(s) \to \rho(P) \\
\rho(P \fatsemi Q) & := & \rho(P) \fatsemi \rho(Q) \\
\rho(P \,\Box\, Q) & := & \rho(P) \,\Box\, \rho(Q) \\
\rho(P \sqcap Q) & := & \rho(P) \sqcap \rho(Q) \\
\rho(P \,|[\, s \,]|\, Q) & := & \rho(P) \,|[\, \sigma(s) \,]|\, \rho(Q) \\
\rho(P \,|[\, s1 \,|\, s2 \,]|\, Q) & := & \rho(P) \,|[\, \sigma(s1) \,|\, \sigma(s2) \,]|\, \rho(Q) \\
\rho(P \,||\, Q) & := & \rho(P) \,||\, \rho(Q) \\
\rho(P \,|||\, Q) & := & \rho(P) \,|||\, \rho(Q) \\
\rho(P \setminus s) & := & \rho(P) \setminus \sigma(s) \\
\rho(P[[p]]) & := & \rho(P)[[\sigma(p)]] \\
\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) & := & \textbf{if } \sigma(\varphi) \textbf{ then } \rho(P) \textbf{ else } \rho(Q)
\end{array}
$$

In the next theorem we prove that the *reduct property* holds over the CSP models. This ensures that the semantics of a process is frozen when translated to a larger context.

THEOREM 6.1.8 (REDUCT PROPERTY OVER THE CSP MODELS) *Let P be an arbitrary* CSP *process of a* CSP-CASL *specification $Sp = (D, P)$. Moreover, let $\sigma : \Sigma \to \Sigma'$ be a* CSP-CASL *data logic signature morphism and M' a Σ'-model. Then,*

$$\begin{aligned}
&traces([\![P]\!]_{\nu:X\to M'|_\sigma}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}:\sigma(X)\to M'}))\\
&failures([\![P]\!]_{\nu:X\to M'|_\sigma}) = \hat{\alpha}_{\mathcal{F}}(failures([\![\rho(P)]\!]_{\hat{\nu}:\sigma(X)\to M'}))\\
&divergences([\![P]\!]_{\nu:X\to M'|_\sigma}) = \hat{\alpha}_{\mathcal{N}}(divergences([\![\rho(P)]\!]_{\hat{\nu}:\sigma(X)\to M'}))
\end{aligned}$$

where X is the set of free variables in P, $\nu : X \to M'|_\sigma$ and $\hat{\nu} : \sigma(X) \to M'$ are variable evaluations with

$$\nu(x : s) = \hat{\nu}(x : \sigma(s)).$$

PROOF. The proof is by structural induction on the CSP process operator P. Here, we show for each semantical model, how the proof is carried out for a primitive CSP operator such as *STOP* and for the action prefix operator ($t \to P$). The proof for the other CSP process operators is reported in Appendix A.1.

Traces model For the primitive process *STOP* we need to prove the following:

$$traces([\![STOP]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(STOP)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we calculate the trace set, which is $\{\langle\rangle\}$. Applying the inverse translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$, we obtain $\hat{\alpha}^*(\{\langle\rangle\})$, i.e., $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(STOP)]\!]_{\hat{\nu}}))$. Thus, $traces([\![STOP]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(STOP)]\!]_{\hat{\nu}}))$.

For the action prefix process $t \to P$ we need to prove the following:

$$traces([\![t \to P]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(t \to P)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![t]\!]_\nu \to [\![P]\!]_\nu).$$

We then calculate the trace set:

$$\{[\![\langle\rangle]\!]_\nu\} \cup \{[\![t]\!]_\nu \frown q \mid q \in traces([\![P]\!]_\nu)\}.$$

We now unfold the definition of the variable evaluation $[\![t]\!]_\nu$ (details of this definition can be found in [Rog06]):

$$[\![t]\!]_\nu = [(s, \nu^\sharp(t))]_{\sim_{M'|_\sigma}}.$$

In [Rog06] (Lemma 5) proves that $\nu^\sharp(t) = \hat{\nu}^\sharp(\sigma(t))$. Applying the alphabet translation α we obtain:

$$\alpha([(s, \nu^\sharp(t))]_{\sim_{M'|_\sigma}}) = [(\sigma^S(s), \hat{\nu}^\sharp(\sigma(t)))]_{M'}.$$

We now apply the inverse alphabet translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$, we obtain:

$$\{\hat{\alpha}^*([\![\langle\rangle]\!]_{\hat{\nu}})\} \cup \{\hat{\alpha}^{*\checkmark}([(\sigma^S(s), \hat{\nu}^\sharp(\sigma(t)))]_{M'}) \frown q \mid q \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}}))\}.$$

Pulling out the $\hat{\alpha}$ from the above trace set, we obtain:

$$\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(t \rightarrow P)]\!]_{\hat{\nu}})).$$

Thus, $traces([\![t \rightarrow P]\!]_\nu = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(t \rightarrow P)]\!]_{\hat{\nu}}))$.

Stable failure model For the primitive process *STOP* we need to prove the following:

$$\begin{array}{ll} & (traces([\![STOP]\!]_\nu), failures([\![STOP]\!]_\nu)) \\ = & \hat{\alpha}_{\mathcal{F}}(traces([\![\rho(STOP)]\!]_{\hat{\nu}}), failures([\![\rho(STOP)]\!]_{\hat{\nu}})). \end{array}$$

The trace component is identical to the one presented for the traces model. For the failures component we prove the following:

$$failures([\![STOP]\!]_\nu) = \hat{\alpha}_{\mathcal{F}}(failures([\![\rho(STOP)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we calculate the failures set:

$$\{\langle\rangle, X \mid X \subseteq \mathcal{A}lph(M'|_\sigma)^\checkmark\}.$$

We now apply the well-defined and injective alphabet translation α, and we obtain:

$$\{[\![\langle\rangle]\!]_{\hat{\nu}}, X \mid X \subseteq \mathcal{A}lph(M')^\checkmark\}.$$

Applying the inverse translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$, we obtain:

$$\{\hat{\alpha}^*(\langle\rangle), \hat{\alpha}^\checkmark_{\mathbb{P}}(X) \mid X \subseteq \mathcal{A}lph(M')^\checkmark\}.$$

I.e., $\hat{\alpha}_{\mathcal{F}}(failures([\![\rho(STOP)]\!]_{\hat{\nu}}))$. Thus, $failures([\![STOP]\!]_\nu) = \hat{\alpha}_{\mathcal{F}}(failures([\![\rho(STOP)]\!]_{\hat{\nu}}))$.

For the action prefix process $t \rightarrow P$ we need to prove the following:

$$\begin{array}{ll} & (traces([\![a \rightarrow P]\!]_\nu), failures([\![a \rightarrow P]\!]_\nu)) \\ = & \hat{\alpha}_{\mathcal{F}}(traces([\![\rho(a \rightarrow P)]\!]_{\hat{\nu}}), failures([\![\rho(a \rightarrow P)]\!]_{\hat{\nu}})). \end{array}$$

The trace component is identical to the one presented for the traces model. For the failures component we prove the following:

$$failures([\![t \rightarrow P]\!]_\nu) = \hat{\alpha}(failures([\![\rho(t \rightarrow P)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$failures([\![t]\!]_\nu \rightarrow [\![P]\!]_\nu).$$

We then calculate the failures set and we obtain:

$$\begin{array}{rl} & \{([\![\langle\rangle]\!]_\nu, Y) \mid [\![t]\!]_\nu \notin Y, Y \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark)\} \\ \cup & \{(\langle [\![t]\!]_\nu \rangle \frown q, Y) \mid (q, Y) \in \mathit{failures}([\![P]\!]_\nu)\}. \end{array}$$

We now unfold the definition of the variable evaluation, i.e., $[\![t]\!]_\nu = [(s, \nu^\sharp(t))]_{\sim_{M'|_\sigma}}$.

Thanks to Lemma 5 from [Rog06] we have that $\nu^\sharp(t) = \hat{\nu}^\sharp(\sigma(t))$. We now apply the inverse alphabet translation of the stable failure domain $\hat{\alpha}_\mathcal{F}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$, we obtain:

$$\begin{array}{rl} & \{\hat{\alpha}^*([\![\langle\rangle]\!]_{\hat{\nu}}), \hat{\alpha}^\checkmark_\mathbb{P}(Y) \mid \hat{\alpha}([(\sigma^S(s), \hat{\nu}^\sharp(\sigma(t)))]_{M'}) \notin \hat{\alpha}^\checkmark_\mathbb{P}(Y), \hat{\alpha}^\checkmark_\mathbb{P}(Y) \in \mathbb{P}(\mathcal{A}lph(M')^\checkmark)\} \\ \cup & \{\hat{\alpha}^{*\checkmark}([(\sigma^S(s), \hat{\nu}^\sharp(\sigma(t)))]_{M'} \frown q), \hat{\alpha}^\checkmark_\mathbb{P}(Y) \mid (q, Y) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above failure set, we obtain:

$$\hat{\alpha}(\mathit{failures}([\![\rho(t \to P)]\!]_{\hat{\nu}})).$$

Putting together the trace set and the failures set, we obtain the stable failure denotation of $\hat{\alpha}_\mathcal{F}(\mathit{traces}([\![\rho(a \to P)]\!]_{\hat{\nu}}), \mathit{failures}([\![\rho(a \to P)]\!]_{\hat{\nu}}))$.

Failures/divergences model For the primitive process *STOP* we need to prove the following:

$$\begin{array}{rl} & (\mathit{failures}^\perp([\![STOP]\!]_\nu), \mathit{divergences}([\![STOP]\!]_\nu)) \\ = & \hat{\alpha}_\mathcal{N}(\mathit{failures}^\perp([\![\rho(STOP)]\!]_{\hat{\nu}}), \mathit{divergences}([\![\rho(STOP)]\!]_{\hat{\nu}})). \end{array}$$

For the failures component we follow the same argument as in the stable failure model, and obtain:

$$\mathit{failures}^\perp([\![STOP]\!]_\nu = \hat{\alpha}_\mathcal{F}(\mathit{failures}^\perp([\![\rho(STOP)]\!]_{\hat{\nu}})).$$

For the divergences component we prove the following:

$$\mathit{divergences}([\![STOP]\!]_\nu = \hat{\alpha}_\mathcal{F}(\mathit{divergences}([\![\rho(STOP)]\!]_{\hat{\nu}})).$$

This trivially holds, as the divergence set for the process *STOP* is the empty set.

For the action prefix process $t \to P$ we need to prove the following:

$$\begin{array}{rl} & (\mathit{failures}^\perp([\![a \to P]\!]_\nu), \mathit{divergences}([\![a \to P]\!]_\nu)) \\ = & \hat{\alpha}_\mathcal{N}(\mathit{failures}^\perp([\![\rho(a \to P)]\!]_{\hat{\nu}}), \mathit{divergences}([\![\rho(a \to P)]\!]_{\hat{\nu}})). \end{array}$$

For the failures component we follow the same argument as above, and obtain:

$$\mathit{failures}^\perp([\![t \to P]\!]_\nu = \hat{\alpha}_\mathcal{F}(\mathit{failures}^\perp([\![\rho(t \to P)]\!]_{\hat{\nu}})).$$

For the divergences component we prove the following:

$$\mathit{divergences}([\![t \to P]\!]_\nu = \hat{\alpha}(\mathit{divergences}([\![\rho(t \to P)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences([\![t]\!]_\nu \rightarrow [\![P]\!]_\nu).$$

We then calculate the divergence set:

$$\{[\![t]\!]_\nu \frown q \mid q \in divergences([\![P]\!]_\nu)).$$

We now unfold the definition of the variable evaluation, i.e., $[\![t]\!]_\nu = [(s, \nu^\sharp(t))]_{\sim_{M'|_\sigma}}$. Again, thanks to Lemma 5 in [Rog06] we have that $\nu^\sharp(t) = \hat{\nu}^\sharp(\sigma(t))$. We apply the alphabet translation α and obtain: $\alpha([(s, \nu^\sharp(t))]_{\sim_{M'|_\sigma}}) = [(\sigma^S(s), \hat{\nu}^\sharp(\sigma(t)))]_{M'}$.

We now apply the inverse alphabet translation $\hat{\alpha}_\mathcal{N}$ of the failures/divergences model and using the induction hypothesis on $divergences([\![P]\!]_\nu)$, we obtain:

$$\{\hat{\alpha}^{*\checkmark}([(\sigma^S(s), \hat{\nu}^\sharp(\sigma(t)))]_{M'}) \frown q \mid q \in \hat{\alpha}(divergences([\![\rho(P)]\!]_{\hat{\nu}})).$$

Pulling out the $\hat{\alpha}$ from the above divergence set, we obtain:

$$\hat{\alpha}(divergences([\![\rho(t \rightarrow P)]\!]_{\hat{\nu}})).$$

Putting together the failures set and the divergences set, we obtain the failures/divergences denotation of $\hat{\alpha}_\mathcal{N}(failures^\perp([\![\rho(a \rightarrow P)]\!]_{\hat{\nu}}), divergences([\![\rho(a \rightarrow P)]\!]_{\hat{\nu}}))$.

■

This insights allow us to define a refinement notion based on a general data logic signature morphism for CSP-CASL.

DEFINITION 6.1.9 *Let $\sigma : \Sigma \rightarrow \Sigma'$ be a* CSP-CASL *data logic signature morphism as defined in Definition 6.1.2. Let $(d_M)_{M \in I}$ and $(d'_{M'})_{M' \in I'}$ be families of process denotations over Σ and Σ', respectively. Then,*

$$(d_M)_{M \in I} \rightsquigarrow^\sigma_\mathcal{D} (d'_{M'})_{M' \in I'} \iff I'|_\sigma \subseteq I \wedge \forall M' \in I' : d_{M'|_\sigma} \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d'_{M'}).$$

Here, $I'|_\sigma = \{M'|_\sigma \mid M' \in I'\}$, and $\sqsubseteq_\mathcal{D}$ denotes CSP refinement in the chosen semantic, where $\mathcal{D} \subseteq \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$.

Given CSP-CASL specifications $Sp = (D, P)$ and $Sp' = (D', P')$, by abuse of notation we also write (Sp refines to Sp')

$$Sp \rightsquigarrow^\sigma_\mathcal{D} Sp'$$

if the condition of the model class inclusion and the set inclusion of the process denotation holds for Sp and Sp', respectively (see Definition 6.1.9).

On the syntactic level, we additionally define the notion of data refinement and process refinement in order to characterize situations, where one specification part remains constant.

DEFINITION 6.1.10 (DATA REFINEMENT) *Let (D, P) and $(D', \rho(P))$ be two specifications and $\sigma : \Sigma(D) \to \Sigma(D')$ a* CSP-CASL *data logic signature morphism.* CSP-CASL *data refinement is defined as follows:*

$$(D, P) \overset{data}{\rightsquigarrow_\sigma} (D', \rho(P)) \text{ if } \boldsymbol{Mod}(D')|_\sigma \subseteq \boldsymbol{Mod}(D).$$

DEFINITION 6.1.11 (PROCESS REFINEMENT) *Let (D, P) and (D, P') be two specifications.* CSP-CASL *process refinement is defined as follows:*

$$(D, P) \overset{proc}{\rightsquigarrow_\mathcal{D}} (D, P') \text{ if } \forall M \in \boldsymbol{Mod}(D) : [\![[\![P]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_\mathcal{D} \sqsubseteq_\mathcal{D} [\![[\![P']\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_\mathcal{D}$$

for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$.

Clearly, both these refinements are special forms of CSP-CASL refinement in general.

LEMMA 6.1.12 *Let $Sp = (D, P)$, $Sp\_d = (D', \rho(P))$ and $Sp\_p = (D, P')$ be* CSP-CASL *specifications. Let $\sigma : \Sigma(D) \to \Sigma(D')$ is the* CSP-CASL *data logic signature morphism and ρ the process translation. Then, for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$,*

1. *$Sp \overset{data}{\rightsquigarrow_\sigma} Sp\_d$ implies $Sp \rightsquigarrow^\sigma_\mathcal{D} Sp\_d$, and*
2. *$Sp \overset{proc}{\rightsquigarrow_\mathcal{D}} Sp\_p$ implies $Sp \rightsquigarrow^\sigma_\mathcal{D} Sp\_p$.*

PROOF. Let $(d_M)_{M\in\mathbf{Mod}(D)}$, $(d'_{M'})_{M'\in\mathbf{Mod}(D')}$ and $(d''_M)_{M\in\mathbf{Mod}(D)}$ be the families of process denotations of Sp, Sp_d and Sp_p respectively. We prove the implication in (1) and (2).

1. We need to show that:

   $$(d_M)_{M\in\mathbf{Mod}(D)} \rightsquigarrow^\sigma_\mathcal{D} (d'_{M'})_{M'\in\mathbf{Mod}(D')}.$$

   This holds if $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$ and for all $M' \in \mathbf{Mod}(D')$ it holds $d_{M'|_\sigma} \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d_{M'})$. The data refinement $(d_M)_{M\in\mathbf{Mod}(D)} \overset{\mathtt{data}}{\rightsquigarrow_\sigma} (d'_{M'})_{M'\in\mathbf{Mod}(D')}$ establishes the model class inclusion, i.e., $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$.

   Let $M' \in \mathbf{Mod}(D')$ and $d'_{M'}$ be the denotation of $[\![\rho(P)]\!]_{M'}$. Then, thanks to the reduct property we have that

   $$d'_{M'|_\sigma} = \hat{\alpha}_\mathcal{D}(d'_{M'}).$$

   Thus, $d'_{M'|_\sigma} \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d'_{M'})$. Hence, $(d_M)_{M\in\mathbf{Mod}(D)} \rightsquigarrow^\sigma_\mathcal{D} (d'_{M'})_{M'\in\mathbf{Mod}(D')}$; therefore $Sp \rightsquigarrow^\sigma_\mathcal{D} Sp\_d$.

2. Again we need to show that:

   $$(d_M)_{M\in\mathbf{Mod}(D)} \rightsquigarrow^\sigma_\mathcal{D} (d''_M)_{M\in\mathbf{Mod}(D')}.$$

   Here, we work with the same model classes $\mathbf{Mod}(D)$. From the process refinement $(d_M)_{M\in\mathbf{Mod}(D)} \overset{\mathtt{proc}}{\rightsquigarrow_\mathcal{D}} (d''_M)_{M\in\mathbf{Mod}(D')}$ we have: $\forall M \in \mathbf{Mod}(D)$. $d_M \sqsubseteq_\mathcal{D} d''_M$. Thus, $(d_M)_{M\in\mathbf{Mod}(D)} \rightsquigarrow^\sigma_\mathcal{D} (d'_{M'})_{M'\in\mathbf{Mod}(D')}$; therefore, $Sp \rightsquigarrow^\sigma_\mathcal{D} Sp'$.

■

6.2 Theory of CSP-CASL enhancement notion

In the last section we have presented a general theory for the refinement of CSP-CASL specifications. Such theory allows to capture the vertical development of systems. That is, we have a tower of specifications:

$$S_0 \rightsquigarrow^{\sigma}_{\mathcal{D}} S_1 \rightsquigarrow^{\sigma}_{\mathcal{D}} \ldots \rightsquigarrow^{\sigma}_{\mathcal{D}} \ldots S_{n-1} \rightsquigarrow^{\sigma}_{\mathcal{D}} S_n.$$

Here, S_0 is the abstract specification – it contains basic information about the system. In CSP-CASL, this is captured by a loosely specified data and a nondeterministic process description. Such specification is then refined step by step. This means, on the data part, a reduction of the model classes; and on the process part, we have a process behavior which is less nondeterministic. Finally the specification S_n contains a detailed description of the system.

We are now interested in capturing a horizontal development of systems. In a horizontal development new functionality or features are added to an existing systems. For the corresponding software development process, this means that the specification of an advanced product is developed by enhancement and combination of basic specifications. Such concept allows to capture the notion of software product lines.

In this section we elaborate and present a theory of enhancement for CSP-CASL specifications. Here, we would like to capture the notion of horizontal development. That is, how we can extend basic CSP-CASL specifications with new functionality and form new elaborated CSP-CASL specifications.

In CSP-CASL we use the notion of *conservative extension* defined in the context of algebraic specification. Intuitively speaking, an extension is conservative if it does not 'specify away' any models, i.e., if each model of the original specification can be enlarged to a model of the extended specification [Sho67].

DEFINITION 6.2.1 *We say that a signature $\Sigma = (S, TF, PF, P, \leq)$ is **embedded into** a signature $\Sigma' = (S', TF', PF', P', \leq')$ if $S \subseteq S'$, $TF \subseteq TF'$, $PF \subseteq PF'$, $P \subseteq P'$, and the following conditions regarding subsorting hold:*

preservation and reflection $\leq = \leq' \cap (S \times S)$.

weak non-extension *$s_1 \neq s_2$ and $\sigma^S(s_1) \leq_{S'} u'$ and $\sigma^S(s_2) \leq_{S'} u'$ implies that there exist a sort $t \in S$ with $s_1 \leq t, s_2 \leq t$ and $\sigma^S(t) \leq u'$.*

We write $\iota : \Sigma \to \Sigma'$ for the induced map from Σ to Σ', where $\iota^S(s) = s, \iota^{TF \cup PF}(f) = f, \iota^P(p) = p$ for all sort symbols $s \in S$, functions symbols $f \in TF \cup PF$ and predicate symbol $p \in P$.

Obviously, such induced map ι is both, a CASL signature morphism and a CSP-CASL data logic signature morphism. We carry over the notion of a *conservative extension* to our setting.

DEFINITION 6.2.2 *Let D and D' be two* CASL *specifications, with signatures Σ and Σ' respectively, where Σ is embedded into Σ'. D'* **conservatively extends** *D if $\mathbf{Mod}(D) = \mathbf{Mod}(D')|_{\iota}$.*

Therefore, an extension is conservative when no models are lost: every model of the specification being extended is a reduct of some model of the extended specification. CASL provides annotations such as: %**implies**, %**def**, and %**cons** to denote that the model class is not changed, that each model of the specification can be uniquely extended to a model of the extended specification, or that the extension is conservative, respectively. Such annotations have no effect on the semantics of a specification: a specifier may use them to express his intentions, tools may use them to generate proof obligations [RS02].

Extensions with new symbols are not necessarily conservative. For example, consider the following specifications BAS and EXT, where the new symbol c in EXT imposes a constraint on the symbols a and b inherited from BAS. Thus, models with $M(a) \neq M(b)$ of BAS are not included in $\mathbf{Mod}(\text{EXT})|_\iota$ and EXT is not a conservative extension of BAS.

spec BAS =	**spec** EXT =
sort S	**sort** $S < T$
op $a, b : S$	**ops** $a, b : S;\ c : T$
end	**axioms** $c = a;\ c = b$
	end

[RS02] compiles a comprehensive set of proof rules to establish that one specification conservatively extends another. For instance, the extension by the CASL construct 'operation definition' is conservative.

In the semantical construction of CSP-CASL, signature embeddings lead to alphabet embeddings, exactly as proven in Lemma 6.1.3.

Now we define the central notion of *enhancement* between CSP-CASL specifications.

DEFINITION 6.2.3 *Let $Sp = (D, P)$ and $Sp' = (D', P')$ be* CSP-CASL *specifications, and let Σ and Σ' be the signatures of D and D', respectively. Let ι and α be the induced mapping. We say Sp' is an* **enhancement** *of Sp, denoted by $Sp \gg Sp'$, if*

1. *Σ is embedded into Σ',*
2. $\boldsymbol{Mod}(D) = \boldsymbol{Mod}(D')|_\iota$, *and*
3. *for all $M' \in \boldsymbol{Mod}(D')$ it holds that:*

$$traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{T}}(traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')}))$$
$$failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{F}}(failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})).$$

Intuitively, a CSP-CASL enhancement notion asserts that: the meaning of old symbols are preserved (*condition 1 and 2*) and new process symbols use old symbols only without change the meaning. For the latter let us consider the following CSP-CASL specifications.

ccspec HUGO =
  data
    free type $s ::= a$
  process
    $P = a \rightarrow SKIP$
end

ccspec ERNA =
  data
    free type $s ::= a$
    free type $t ::= b$
  process
    $P = a \rightarrow SKIP$
    $Q = b \rightarrow P$
end

ccspec HERMINE =
  data
    free type $s ::= a$
    free type $t ::= b$
  process
    $P = a \rightarrow SKIP$
      $\sqcap\, b \rightarrow SKIP$
    $Q = b \rightarrow P$
end

Here, we have that HUGO $\gg$ ERNA. That is, on the data part we have the embedding of the symbols $\Sigma(\text{HUGO}) \subseteq \Sigma(\text{ERNA})$ and its a conservative extension, i.e., $\mathbf{Mod}(D_{\text{HUGO}}) = \mathbf{Mod}(D_{\text{ERNA}})\,|_\iota$. On the process side the added new symbols don't interfere with the old process denotation, i.e., $\mathit{traces}(\text{HUGO}) = \hat{\alpha}_{\mathcal{T}}(\mathit{traces}(\text{ERNA}))$ and $\mathit{failures}(\text{HUGO}) = \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\text{ERNA}))$.

However, we have that $\neg(\text{HUGO} \gg \text{HERMINE})$. Here, on the process part the old process behavior uses the new added data. We have that $\mathit{traces}(\text{HUGO}) = \hat{\alpha}_{\mathcal{T}}(\mathit{traces}(\text{HERMINE}))$, however $\mathit{failures}(\text{HUGO}) \neq \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\text{HERMINE}))$.

6.3 Summary

In this chapter we have presented two directions of system development: a refinement (or *vertical development*) notion for CSP-CASL; and an enhancement (or *horizontal development*) notion for CSP-CASL specifications.

For the refinement part, we have defined a new notion based on model class inclusion with arbitrary change of signature in the data part. Intuitively a CSP-CASL refinement describes the following development process: On the data part, the model classes are reduced by adding new informations about the data. On the process part, the refined process description is less internally non-deterministic. That is, the environment in which the process is defined has more control of the process.

We also presented a theory of enhancement for CSP-CASL. Intuitively a CSP-CASL enhancement notion asserts that: the meaning of old symbols are preserved and new process symbols use old symbols only without change the meaning. This theory will allow us to capture the notion of *horizontal development*, in which new features (or functions) are added to existing systems.

(CHAPTER ...7)

Proof support for CSP-CASL development notions

Contents

In this chapter we present techniques to discharge proof obligation that arises when proving a development step (*vertical* or *horizontal*) between CSP-CASL specifications. In Section 7.1 we present a proof support for CSP-CASL refinement. Such proof support is based on a decomposition theorem of CSP-CASL refinement. This will allow us to re-use existing tools for CSP and CASL refinement. In Section 7.2 we illustrate a proof support for CSP-CASL enhancement. Here, we provide two enhancement patterns that allow us to prove enhancement relation between CSP-CASL specifications.

The results presented in this chapter have been published in [KR09] and [KRS08].

7.1 Proof support for CSP-CASL refinement

Proof support for CSP-CASL refinement is based on a decomposition theorem. This decomposition theorem gives rise to a proof method for CSP-CASL, namely, we study CSP-CASL refinement in terms of CASL refinement and CSP refinement separately. With regards to CSP-CASL refinement, data turns out to dominate the processes: While any CSP-CASL refinement can be decomposed into first a data refinement followed by a process refinement, there is no such decomposition result possible for the reverse order, i.e., first CSP refinement and then CASL refinement. This insight is in accordance with the 2-step semantics of CSP-CASL, where in the first step we evaluate the data part and only in the second step apply the process semantics.

THEOREM 7.1.1 (CSP-CASL REFINEMENT DECOMPOSITION) *Let (D,P), $(D',\rho(P))$ and (D',P') be a* CSP-CASL *specifications and $\sigma : \Sigma(D) \to \Sigma(D')$ be a* CSP-CASL *data logic signature morphism. Then, for all $\mathcal{D} \in \{\mathcal{T},\mathcal{F},\mathcal{N}\}$,*

$$\begin{array}{c}(D,P) \overset{data}{\rightsquigarrow_\sigma} (D',\rho(P)) \text{ and } (D',\rho(P)) \overset{proc}{\rightsquigarrow_\mathcal{D}} (D',P') \\ \text{implies} \\ (D,P) \rightsquigarrow^\sigma_\mathcal{D} (D',P').\end{array}$$

PROOF. Let (D,P) have denotations $(d_M)_{M\in\mathbf{Mod}(D)}$, $(D',\rho(P))$ have denotations $(d'_{M'})_{M'\in\mathbf{Mod}(D')}$, and (D',P') have denotations $(d''_{M'})_{M'\in\mathbf{Mod}(D')}$. We need to show that:

$$(d_M)_{M\in\mathbf{Mod}(D)} \rightsquigarrow^\sigma_\mathcal{D} (d''_{M'})_{M'\in\mathbf{Mod}(D')}.$$

This holds if $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$ and $\forall M' \in \mathbf{Mod}(D').\ d_{M'|_\sigma} \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d''_{M'})$.

The data refinement ($\overset{data}{\rightsquigarrow_\sigma}$) immediately gives the required model class inclusion. That is, $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$.

Let $M' \in \mathbf{Mod}(D')$ and $d'_{M'}$ be the respective denotation. Then, thanks to the reduct property we have that $d_{M'|_\sigma} = \hat{\alpha}_\mathcal{D}(d'_{M'})$.

The process refinement ($\overset{proc}{\rightsquigarrow_\mathcal{D}}$) yields:

$$\forall M' \in \mathbf{Mod}(D').\ d'_{M'} \sqsubseteq_\mathcal{D} d''_{M'}.$$

Thanks to the monotonicity of the alphabet translation $\hat{\alpha}_\mathcal{D}$ w.r.t. the process refinement, we apply the inverse translation $\hat{\alpha}_\mathcal{D} : \mathcal{D}(\mathcal{A}lph(M'|_\sigma)) \to ? \mathcal{D}(\mathcal{A}lph(M'))$ to both sides of the process denotation:

$$\forall M' \in \mathbf{Mod}(D').\ \hat{\alpha}_\mathcal{D}(d'_{M'}) \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d''_{M'}).$$

This allows us to conclude that

$$\forall M' \in \mathbf{Mod}(D').\ (d_{M'|_\sigma} =_\mathcal{D} \hat{\alpha}_\mathcal{D}(d'_{M'})) \sqsubseteq_\mathcal{D} \hat{\alpha}_\mathcal{D}(d''_{M'}).$$

Thus, the refinement $(d_M)_{M\in\mathbf{Mod}(D)} \rightsquigarrow^\sigma_\mathcal{D} (d''_{M'})_{M'\in\mathbf{Mod}(D')}$ holds, i.e., $(D,P) \rightsquigarrow^\sigma_\mathcal{D} (D',P')$.

■

This result forms the basis for the CSP-CASL tool support developed in [OIR09]. In order to prove that a CSP-CASL refinement $(D,P) \rightsquigarrow^\sigma_\mathcal{D} (D',P')$ holds, first one uses proof support for CASL [MML07] alone in order to establish $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$. Independently of this, one has then to check the process refinement $P \sqsubseteq_\mathcal{D} P'$, for all $\mathcal{D} \in \{\mathcal{T},\mathcal{F},\mathcal{N}\}$. In principle, the latter step can be carried out using CSP-Prover, see e.g. [IR05]. The use of CSP-Prover, however, requires the CASL specification D' to be translated into an alphabet of communications. The tool CSP-CASL-Prover [OIR09] implements this translation and also generates proof support for theorem proving in CSP-CASL.

Changing the order in the above decomposition theorem, i.e., to first perform a process refinement followed by a data refinement, however, is not possible in general. Often, process properties depend on data, as the following counter example illustrates, in which we have

$$(D,P) \rightsquigarrow^{\sigma}_{\mathcal{D}} (D',P') \quad \text{but} \quad (D,P) \not\overset{\text{proc}}{\rightsquigarrow}_{\mathcal{D}} (D,P').$$

Consider the three CSP-CASL specifications ABS, MID and CONC, where MID consists of the data part of ABS and the process part of CONC:

ccspec ABS =
  data
    sorts S
    ops $a, b : S$;
  process
    P $= a \rightarrow STOP$
end

ccspec MID =
  data
    sort S
    ops $a, b : S$;
  process
    Q $= a \rightarrow Stop \,|[\, a \,]|$
      $b \rightarrow STOP$
end

ccspec CONC =
  data
    sort S
    ops $a, b : S$;
    axiom $a = b$
  process
    R $= a \rightarrow STOP \,|[\, a \,]|$
      $b \rightarrow STOP$
end

Let N be a CASL model of the data part D_{ABS} of ABS with $N(S) = \{\#, *\}$, $N(a) = \#$, $N(b) = *$. Concerning the process denotations in the traces model $\mathcal{T}$ relatively to N, for ABS we obtain the denotation[1] $d_{\text{ABS}} = \{\langle\rangle, \langle\#\rangle\}$. In MID, the alphabetized parallel operator requires synchronization only w.r.t. the event a. As $N \models \neg a = b$, the right hand side of the parallel operator, which is prepared to engage in b, can proceed with b, which yields the trace $\langle * \rangle$ in the denotation. The left hand side, however, which is prepared to engage in a, does not find a partner for synchronization and therefore is blocked. This results in the denotation $d_{\text{MID}} = \{\langle\rangle, \langle * \rangle\}$. As $d_{\text{MID}} \not\subseteq d_{\text{ABS}}$, we have ABS $\not\overset{\text{proc}}{\rightsquigarrow}_{\mathcal{T}}$ MID.

In CONC, the axiom $a = b$ prevents N to be a model of the data part. This makes it possible to establish ABS $\overset{\text{proc}}{\rightsquigarrow}_{\mathcal{T}}$ CONC over the traces model $\mathcal{T}$. Using Theorem 7.1.1, we first prove the data refinement: CONC adds an axiom to ABS – therefore, D_{ABS} refines to D_{CONC} with respect to CASL; concerning the process refinement, using the equation $a = b$ and the step law for generalized parallel, we obtain

$$\begin{aligned} a \rightarrow STOP \,|[\, a \,]|\, b \rightarrow STOP \quad &= \quad a \rightarrow STOP \,|[\, a \,]|\, a \rightarrow STOP \\ &=_{\mathcal{T}} \quad a \rightarrow (STOP \,|[\, a \,]|\, STOP) \\ &=_{\mathcal{T}} \quad a \rightarrow STOP \end{aligned}$$

Thus, over D_{CONC} the process parts of ABS CONC are semantically equivalent and therefore in refinement relation over the traces model $\mathcal{T}$. Figure 7.1 illustrates the overall decomposition of CSP-CASL refinement.

In the following example we illustrate how we prove a refinement step of the binary calculator example, using the decomposition theorem implemented in CSP-CASL-PROVER. More challenging refinement steps proof will be presented in Chapter 11.

[1] For the sake of readability, we write the element of the carrier sets rather than their corresponding events in the alphabet of communications.

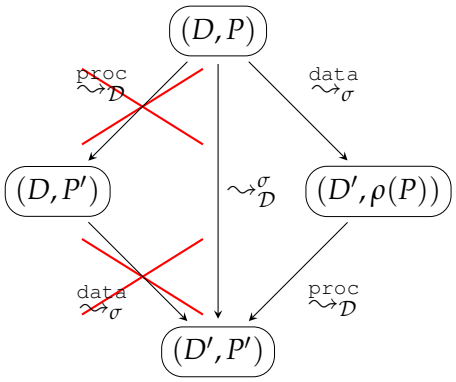

Figure 7.1: Decomposition theorem of CSP-CASL refinement.

EXAMPLE 7.1.2 Here, we show the following refinement of the binary calculator:

$$\text{BCALC0} \rightsquigarrow_{\mathcal{F}} \text{BCALC3}$$

In order to prove this refinement we state, using the keyword **view**, how the refinement goes:

view *Refinement*: BCALC0 **to** BCALC3

A view is a convenient way in CASL to relate two specifications; here, we use it to state the refinement. In general, a view is used in CASL to state a specification morphism (induced by a symbol map) from a (source) specification to an (target) specification.

Figure 7.2 Illustrates a screenshot of CSP-CASL-PROVER for the refinement BCALC0 $\rightsquigarrow_{\mathcal{F}}$ BCALC3.

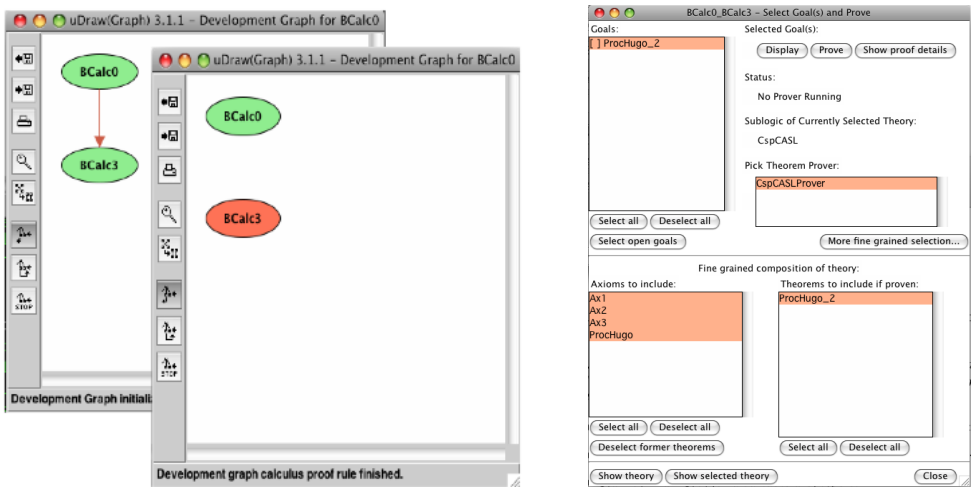

Figure 7.2: Binary calculator refinement in CSP-CASL-PROVER.

The left hand side shows the development graph of our specifications. Here, the back diagram shows the refinement to be proven between the specification BCALC0 and BCALC3; specifically the red arrow indicates the existence of proof obligations. CSP-CASL implements the decomposition theorem and this makes the data refinement to be automatically discharged by HETS. The front diagram illustrates the state of the overall refinement after the data refinement has been proved. Here, the red bubble indicates the presence of a proof obligations to be discharged using CSP-PROVER. To this end, in the right hand side we choose CSP-CASL-PROVER. Here, the tool automatically constructs the alphabets of communications for the process part and generates a *theory file*[2] in which the process description is translated to the input language of CSP-PROVER. At this point we use CSP-PROVER to interactively prove the process refinement. The Isabelle proof script can be found in the Appendix A.2.

■

7.2 Proof support for CSP-CASL enhancement

In order to prove an enhancement step in CSP-CASL, we have identified some *enhancement patterns*. Such patters captures the notion of adding new features to an existing system.

CSP-CASL enhancement guarantees preservation of behaviour up to the first communication that lies outside the original alphabet. This observation is captured in the following proof principle:

THEOREM 7.2.1 (EXTERNAL CHOICE ENHANCEMENT) *Let* $Sp = (D, P =?x :: s \to P')$, *let* $Sp' = (D', P =?x :: s \to P' \ \Box ?y :: t' \to Q')$, *and let* Σ *and* Σ' *be the signatures of* D *and* D', *respectively, let* S *be the set of sorts in* Σ. *If*

1. Σ *is embedded into* Σ', $\boldsymbol{Mod}(D) = \boldsymbol{Mod}(D')|_\iota$, *and*
2. *for all* $u \in S$ *it holds that* $D' \models_{\Sigma'} \forall x : u, y : t' \,.\, x \neq y$,

then $Sp \gg Sp'$.

PROOF. The first two conditions of the enhancement definition hold by assumption. We prove the third condition, that is for all $M' \in \mathbf{Mod}(D')$ we have

$$traces([\![P =?x :: s \to P']\!]_{\nu:X\to\beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![P =?x :: s \to P' \ \Box ?y :: t' \to Q']\!]_{\nu':X\to\beta(M')}))$$

$$failures([\![P =?x :: s \to P']\!]_{\nu:X\to\beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(failures([\![P =?x :: s \to P' \ \Box ?y :: t' \to Q']\!]_{\nu':X\to\beta(M')})).$$

For the traces condition, the trace set of the process $P =?x :: s \to P'$ is given by

$$\{\langle\rangle\} \cup \{\langle a\rangle \frown q \mid q \in traces([\![P'[a/x]]\!]_{\nu:X\to\beta(M'|_\iota)}),\ a \in [s]_{\sim_{\beta(M'|_\iota)}}\}.$$

[2] A theory file is a Isabelle file, in which using different tactics we interactively discharge proof obligations.

Here, $[s]_{\sim_{\beta(M'|_\iota)}}$ is the set of values in the alphabet generated by the sort symbol s relatively to the model $\beta(M'|_\iota)$. The trace set for the extended process $P =?x :: s \to P' \,\Box ?y :: t' \to Q'$ is given by:

$$\begin{array}{rl} & \{\langle\rangle\} \cup \{\langle a\rangle \frown t \mid t \in traces([\![P'[a/x]]\!]_{\nu':X\to\beta(M')}),\ a \in [s]_{\sim_{\beta(M')}}\} \\ \cup & \{\langle b\rangle \frown q \mid q \in traces([\![Q'[b/y]]\!]_{\nu':X\to\beta(M')}),\ b \in [q]_{\sim_{\beta(M')}}\} \end{array}$$

Applying the reduct definition to the trace set of the extended process $\hat{\alpha}_{\mathcal{T}}(traces([\![P =?x :: s \to P' \,\Box ?y :: t' \to Q']\!]_{\nu':X\to\beta(M')}))$ we obtain:

$$\{\langle\rangle\} \cup \{\langle a\rangle \frown t \mid t \in traces([\![P'[a/x]]\!]_{\nu:X\to\beta(M'|_\iota)}),\ a \in [s]_{\sim_{\beta(M'|_\iota)}}\}.$$

Thus, we have that:

$$\begin{array}{rl} traces([\![P =?x :: s \to P']\!]_{\nu:X\to\beta(M'|_\iota)}) = & \hat{\alpha}_{\mathcal{T}}(traces([\![P =?x :: s \to P' \\ & \qquad \Box ?y :: t' \to Q']\!]_{\nu':X\to\beta(M')})) \end{array}$$

For the failures condition, the failure set of the process $?x :: s \to P'$ is given by:

$$\begin{array}{rl} & \{(\langle\rangle, X) \mid [s]_{\sim_{\beta(M'|_\iota)}} \cap X = \emptyset\} \\ \cup & \{(\langle a\rangle \frown p, X) \mid (p, X) \in failures([\![P'([a/x])]\!]_{\nu:X\to\beta(M'|_\iota)}),\ a \in [s]_{\sim_{\beta(M'|_\iota)}}\} \end{array}$$

and the failure set of $?y :: t' \to Q'$ is given by:

$$\begin{array}{rl} & \{(\langle\rangle, Y) \mid [t']_{\sim_{\beta(M')}} \cap Y = \emptyset\} \\ \cup & \{(\langle b\rangle \frown q, Y) \mid (q, Y) \in failures([\![Q'([b/y])]\!]_{\nu':X\to\beta(M')}),\ b \in [t']_{\sim_{\beta(M')}}\} \end{array}$$

The failure set for the extended process $P =?x :: s \to P' \,\Box ?y :: t' \to Q'$ is given by:

$$\begin{array}{rl} & \{(\langle\rangle, X') \mid [s]_{\sim_{\beta(M')}} \cap X' = \emptyset \text{ and } [t']_{\sim_{\beta(M')}} \cap X' = \emptyset\} \\ \cup & \{(\langle a'\rangle \frown p', X') \mid (p', X') \in failures([\![P'([a'/x])]\!]_{\emptyset:\emptyset\to\beta(M')}),\ a' \in [s]_{\sim_{\beta(M')}}\} \\ \cup & \{(\langle b'\rangle \frown q', X') \mid (q', X') \in failures([\![Q'([b'/y])]\!]_{\emptyset:\emptyset\to\beta(M')}),\ b' \in [t']_{\sim_{\beta(M')}}\} \\ \cup & \{(\langle\rangle, X) \mid X \subseteq [s]_{\sim_{\beta(M')}} \cup [t']_{\sim_{\beta(M')}} \\ & \qquad \text{and } \langle\checkmark\rangle \in traces([\![?x :: s \to P']\!]_{\nu':X\to\beta(M')}) \cup traces([\![?y :: t' \to Q']\!]_{\nu':X\to\beta(M')})\} \end{array}$$

We now apply the reduct definition $\hat{\alpha}_{\mathcal{F}}$ of the above failure set. The last set is empty as $\langle\checkmark\rangle$ is not in the traces. For the first set we obtain:

$$\begin{array}{rl} & \hat{\alpha}_{\mathcal{T}}(\{(\langle\rangle, X') \mid [s]_{\sim_{\beta(M')}} \cap X' = \emptyset \text{ and } [t']_{\sim_{\beta(M')}} \cap X' = \emptyset\}) \\ = & \{(\langle\rangle, X) \mid \alpha(\langle\rangle) = \langle\rangle, \alpha(X) = X' \cap \alpha(\mathcal{A}lph(\beta(M'|_\sigma))), [s]_{\sim_{\beta(M')}} \cap X' = \emptyset, [t']_{\sim_{\beta(M')}} \cap X' = \emptyset\} \\ = & \{(\langle\rangle, X) \mid [s]_{\sim_{\beta(M'|_\iota)}} \cap X = \emptyset\} \end{array}$$

For the second set we obtain:

$$\{(\langle a\rangle \frown p, X) \mid (p, X) \in failures([\![P'([a/x])]\!]_{\nu:X\to\beta(M'|_\iota)}), a \in [s]_{\sim_{\beta(M'|_\iota)}}\}.$$

Putting these failures together we obtain the failures of $P =?x :: s \to P'$, i.e.,

$$\begin{array}{rl} failures([\![P =?x :: s \to P']\!]_{\nu:X\to\beta(M'|_\iota)}) = & \hat{\alpha}_{\mathcal{F}}(failures([\![P =?x :: s \to P' \\ & \qquad \Box ?y :: t' \to Q']\!]_{\nu':X\to\beta(M')})). \end{array}$$

Hence, we have that $Sp \gg Sp'$. ∎

Often a specification Sp is enhanced by a specification Sp' by using the overloading functionalities and by adding supersorts. To capture this technique by a characterization theorem, we introduce an extension operation, first on CASL signatures, then on CSP-CASL processes.

DEFINITION 7.2.2 *Given a mapping $\xi : S \to S'$ on sort names, we define*

- $\xi(f) = f : \xi(s_1) \times \cdots \times \xi(s_k) \to \xi(s)$ *for a function symbol* $f : s_1 \times \cdots \times s_k \to t$,
- $\xi(p) = p : \xi(s_1) \times \cdots \times \xi(s_k)$ *for a predicate symbol* $p : s_1 \times \cdots \times s_k$,
- $\xi(x : s) = x : \xi(s)$ *for a variable* x *of type* s *and*
- $\xi(f(t_1, .., t_k)) = \xi(f)(\xi(t_1), \ldots, \xi(t_k)$ *for a* CASL *term* $f(t_1, \ldots, t_k)$.

Σ is embedded into Σ' with a mapping $\xi : S \to S'$ if Σ is embedded into Σ', $TF' = TF \cup \xi(TF)$, $PF' = PF \cup \xi(PF)$, $P' = TF \cup \xi(P)$, and $\leq'$ is the minimal subsort relation with $\leq \subseteq \leq'$ and $(s, \xi(s)) \in \leq'$.

The setting of Definition 7.2.2 ensures that any new function and predicate symbols in Σ' are in overloading relation with the old symbols of Σ. For CSP-CASL processes, ξ is the identity with the exception:

- $\xi(t \to P) = \xi(t) \to \xi(P)$
- $\xi(?x :: s \to P) = ?x :: \xi(s) \to \xi(P)$.

And now we show that enhancement via extension of data using overloading functions and supersorts leads to enhancement of CSP-CASL specifications.

THEOREM 7.2.3 (SUPERSORT ENHANCEMENT) *Let $Sp = (D, P)$ and $Sp' = (D', P')$ be* CSP-CASL *specifications, let Σ and Σ' be the signatures of D and D', respectively. Let S and S' be the sets of sorts in Σ and Σ', respectively, let $\xi : S \to S'$ be a mapping on sort names. If*

1. *Σ is embedded into Σ' with the mapping ξ,*
2. $\boldsymbol{Mod}(D) = \boldsymbol{Mod}(D')|_\iota$, *and*
3. $P' = \xi(P)$,

then $Sp \gg Sp'$.

PROOF. The first two conditions of CSP-CASL enhancement definition holds by assumption. We prove the third condition, that is for all $M' \in \mathbf{Mod}(D')$ it holds that:

$$traces([\![P]\!]_{\nu:X \to \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\xi(P)]\!]_{\nu':X' \to \beta(M')}))$$
$$failures([\![P]\!]_{\nu:X \to \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(failures([\![\xi(P)]\!]_{\nu':X' \to \beta(M')})).$$

First, we show that in the enhanced setting we only have to consider variable bindings to values in the original subsorts. The only introduction of bindings is via the multiple choice operator $?x :: s \to P(x)$. The reduct operator removes traces starting with the value

not present in the original setting:

$$\begin{array}{rl} & \hat{\alpha}_{\mathcal{T}}(traces([\![?x :: \xi(s) \rightarrow \xi(P)(x : \xi(s))]\!]_{\nu':X' \rightarrow \beta(M')}) \\ = & \hat{\alpha}_{\mathcal{T}}(traces([\![?x :: s \rightarrow \xi(P)(x : s)]\!]_{\nu':X' \rightarrow \beta(M')})) \qquad (*) \end{array}$$

where $\nu' : X' \rightarrow \beta(M')$ is an arbitrary variable evaluation. This holds as $s \leq' \xi(s)$ and therefore $[s]_{\sim_{\beta(M')}} \subseteq [\xi(s)]_{\sim_{\beta(M')}}$. Furthermore, for $b \in [\xi(s)]_{\sim_{\beta(M')}} \cap [s]_{\sim_{\beta(M')}}$ we have $b \notin \alpha(\mathcal{A}lph(\beta(M'|_\iota)))$.

We now prove by structural induction on the process operator P that the following holds:

$$traces([\![P]\!]_{\nu:X \rightarrow \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\xi(P)]\!]_{\nu':X' \rightarrow \beta(M')})).$$

where $\nu'(x : \xi(s)) = \alpha(x : s)$, i.e., ν' evaluates only to values reachable under α.

On the CSP-CASL process operator, the mapping ξ is defined as the identity, with the exception of two operators mentioned above.

Let $P = ?x :: s \rightarrow P$. Then $P' = ?x :: \xi(s) \rightarrow \xi(P)$. We now show the following:

$$traces([\![?x :: s \rightarrow P]\!]_{\nu:X \rightarrow \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![?x :: \xi(s) \rightarrow \xi(P)]\!]_{\nu':X' \rightarrow \beta(M')})).$$

By the argument illustrated in (*), we can replace the right hand side:

$$traces([\![?x :: s \rightarrow P]\!]_{\nu:X \rightarrow \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![?x :: s \rightarrow \xi(P)(x : s)]\!]_{\nu':X' \rightarrow \beta(M')})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![?x :: s]\!]_{\nu:X \rightarrow \beta(M'|_\iota)} \rightarrow [\![P]\!]_{\nu:X \rightarrow \beta(M'|_\iota)}).$$

We then calculate the trace set:

$$\{\langle\rangle\} \cup \{\langle a \rangle \frown t \mid t \in traces([\![P[a/x]]\!]_{\nu:X \rightarrow \beta(M'|_\iota)}), a \in [s]_{\sim_{\beta(M'|_\iota)}}\}.$$

We unfold the right hand side of the equation, and we obtain:

$$\hat{\alpha}_{\mathcal{T}}(\{\langle\rangle\} \cup \{\langle a \rangle \frown t \mid t \in traces([\![\xi(P)[a/x]]\!]_{\nu':X' \rightarrow \beta(M')}), a \in [s]_{\sim_{\beta(M')}}\}).$$

The claim follows by induction hypothesis and applying the reduct definition.

Let $P = t \rightarrow P$. Then we have that $P' = \xi(t) \rightarrow \xi(P)$. We prove the following:

$$traces([\![t \rightarrow P]\!]_{\nu:X \rightarrow \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\xi(t) \rightarrow \xi(P)]\!]_{\nu':X' \rightarrow \beta(M')})).$$

As all symbols in t and $\xi(t)$ are in overloading relation, and ν' is restricted to values in the original setting, we have that

$$\nu' \models t = \xi(t).$$

The claim follows by induction hypothesis and applying the reduct definition.

The reduct operator over the failures set ($\hat{\alpha}_{\mathcal{F}}$) removes the failures which are not present in the original setting:

$$\begin{array}{rl} & \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket ?x :: \xi(s) \to \xi(P)(x : \xi(s)) \rrbracket_{\nu':X' \to \beta(M')})) \\ = & \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket ?x :: s \to \xi(P)(x : s) \rrbracket_{\nu':X' \to \beta(M')})) \qquad (**) \end{array}$$

where $\nu' : X' \to \beta(M')$ is an arbitrary variable evaluation. This is a consequence of the condition presented in the traces condition (see (*)) – the healthiness conditions of the stable failure model requires that the trace component in the failures needs to be present in the traces.

For the failures condition, we show the following by structural induction on the process operator P:

$$\mathit{failures}(\llbracket P \rrbracket_{\nu:X \to \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket \xi(P) \rrbracket_{\nu':X \to \beta(M')})).$$

where $\nu'(x : \xi(s)) = \alpha(x : s)$, i.e., ν' evaluates only to values reachable under α.

Again we consider the two cases: Let $P = ?x :: s \to P$. Then $P' = ?x :: \xi(s) \to \xi(P)$. We show the following:

$$\mathit{failures}(\llbracket ?x :: s \to P \rrbracket_{\nu:X \to \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket ?x :: \xi(s) \to \xi(P) \rrbracket_{\nu':X \to \beta(M')})).$$

By the argument in (**) we unfold the right hand side, and obtain

$$\mathit{failures}(\llbracket ?x :: s \to P \rrbracket_{\nu:X \to \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket ?x :: s \to \xi(P)(x : s) \rrbracket_{\nu':X' \to \beta(M')})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}(\llbracket ?x :: s \rrbracket_{\nu:X \to \beta(M'|_\iota)} \to \llbracket P \rrbracket_{\nu:X \to \beta(M'|_\iota)}).$$

We then calculate the failures set and we obtain:

$$\begin{array}{rl} & \{(\langle\rangle, X) \mid [s]_{\sim_{\beta(M'|_\iota)}} \cap X = \emptyset\} \\ \cup & \{(\langle a \rangle \frown q, X) \mid (q, X) \in \mathit{failures}(\llbracket P([a/x]) \rrbracket_{\nu:X \to \beta(M'|_\iota)}), a \in [s]_{\sim_{\beta(M'|_\iota)}}\} \end{array}$$

The claim follows by induction hypothesis and applying the reduct definition.

For the other case, we show the following:

$$\mathit{failures}(\llbracket t \to P \rrbracket_{\nu:X \to \beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket \xi(t) \to \xi(P) \rrbracket_{\nu':X \to \beta(M')})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}(\llbracket t \rrbracket_{\nu:X \to \beta(M'|_\iota)} \to \llbracket P \rrbracket_{\nu:X \to \beta(M'|_\iota)}).$$

We then calculate the failures set and we obtain:

$$\begin{array}{rl} & \{(\llbracket \langle\rangle \rrbracket_{\emptyset:\emptyset \to \beta(M')}, Y) \mid \llbracket t \rrbracket_{\nu:X \to \beta(M'|_\iota)} \notin Y, Y \in \mathbb{P}(\mathcal{A}lph(M'|_\iota)^\checkmark)\} \\ \cup & \{(\langle \llbracket t \rrbracket_{\emptyset:\emptyset \to \beta(M')} \rangle \frown q, Y) \mid (q, Y) \in \mathit{failures}(\llbracket P \rrbracket_{\nu:X \to \beta(M'|_\iota)})\}. \end{array}$$

We unfold the right hand side and calculate the failures set:

$$\begin{aligned} & \hat{\alpha}_{\mathcal{F}}(\{([\![\langle\rangle]\!]_{\nu':X\to\beta(M')}, Y') \mid [\![\xi(t)]\!]_{\nu':X\to\beta(M')} \notin Y', Y' \in \mathbb{P}(\mathcal{A}lph(M')^{\checkmark})\}) \\ \cup \quad & \hat{\alpha}_{\mathcal{F}}(\{(\langle [\![\xi(t')]\!]_{\nu':X\to\beta(M')}\rangle \frown q', Y') \mid (q', Y') \in \mathit{failures}([\![P]\!]_{\nu':X\to\beta(M')})\}). \end{aligned}$$

As above for the traces condition we have that a $\nu' \models t = \xi(t)$. Then the claim follows by induction hypothesis and applying the reduct definition. ■

In Section 10.3 we will illustrate, through a case study, how we use the enhancement patterns introduced in this section.

7.3 Summary

Establishing the theoretical framework for CSP-CASL development notions is not useful in practice if not accompanied by tool support. In this chapter we have presented techniques to discharge proof obligations that could arise from the development notions of CSP-CASL.

On the refinement side of CSP-CASL specifications, we established an approach based on a decomposition theorem. Such decomposition theorem allows us to prove CSP-CASL refinement, first by reasoning about data refinement and then by process refinement. Based on this approach we are able to re-use existing tools to discharge proof obligations.

On the enhancement side of CSP-CASL specifications, we have proposed two enhancement patterns that allow us to capture the notions of adding new features to existing specifications.

(CHAPTER ...8)

Property verification in CSP-CASL

Contents

In this chapter we present techniques to prove interesting properties of CSP-CASL specifications. In particular we study deadlock and livelock analysis in the CSP-CASL context. We show how we use the CSP-CASL refinement notion to prove that properties specified in the abstract specification are inherited by the refined specification.

Throughout this chapter, on the syntactical level we make the following assumption: Let $Sp = (D, P)$ and $Sp = (D', P')$ be two CSP-CASL specifications and $\sigma : \Sigma(D) \rightarrow \Sigma(D')$ be the CSP-CASL data logic signature morphism. Let $Alphabet(P)$ and $Alphabet(P')$ be the set of the communication alphabets used in the processes P and P' respectively. We assume that $Alphabet(P') \subseteq \alpha(Alphabet(P))$, where α is the alphabet translation (see Section 6.1). Such assumption is necessary in order to make sure that the inverse translation $\hat{\alpha}$ is always defined. We call such property the *alphabet condition*.

8.1 Deadlock analysis in CSP-CASL

In this section we show how to analyze deadlock freeness in the context of CSP-CASL. To this end, first we recall how deadlock is characterized in CSP (see Section 2.2). Then, we define what it means for a CSP-CASL specification to be deadlock free. Finally, we establish a proof technique for deadlock freeness based on CSP-CASL refinement, which turns out to be complete.

We recall that in the CSP context, the stable failures model $\mathcal{F}$ is best suited for deadlock analysis. Deadlock is represented by the process *STOP*. Let A be the alphabet. Then the

process *STOP* has denotation

$$(\{\langle\rangle\}, \{(\langle\rangle, X) \mid X \subseteq A^{\checkmark}\}) \in \mathbb{P}(A^{*\checkmark}) \times \mathbb{P}(A^{*\checkmark} \times \mathbb{P}(A^{\checkmark}))$$

in the stable failure model $\mathcal{F}$, i.e., the process *STOP* can perform only the empty trace, and after the empty trace the process *STOP* can refuse to engage in all events. In CSP, a process P is by definition deadlock free if and only if

$$\forall s \in A^*.(s, A^{\checkmark}) \notin \mathit{failures}(P).$$

In other words: Before termination, the process P can never refuse all events; there is always some event that P can perform.

8.1.1 Deadlock definition in CSP-CASL

A CSP-CASL specification has a family of process denotations as its semantics. Each of these denotations represents a possible implementation. We consider a CSP-CASL specification to be deadlock free, if it enforces all its possible implementations to have this property. On the semantical level, we capture this idea as follows:

DEFINITION 8.1.1 *Let $(d_M)_{M \in I}$ be a family of process denotations over the stable failures model, i.e., $d_M = (T_M, F_M) \in \mathcal{F}(\mathcal{A}lph(M))$ for all $M \in I$.*

- *d_M is deadlock free if $(s, X) \in F_M$ and $s \in \mathcal{A}lph(M)^*$ implies that $X \neq \mathcal{A}lph(M)^{\checkmark}$.*
- *$(d_M)_{M \in I}$ is deadlock free if for all $M \in I$ it holds that d_M is deadlock free.*

Deadlock can be analyzed trough refinement checking; that is an implementation is deadlock free if it is the refinement of a deadlock free specification.

THEOREM 8.1.2 *Let $Sp = (D, P)$ and $Sp = (D', P')$ be two CSP-CASL specifications. Let $Sp \rightsquigarrow^{\sigma}_{\mathcal{F}} Sp'$. Let the alphabet condition holds, i.e., $\mathit{Alphabet}(P') \subseteq \alpha(\mathit{Alphabet}(P))$. If Sp is deadlock-free, then so is Sp'.*

PROOF. Let $(d_M)_{M \in \mathbf{Mod}(D)}$ and $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$ be the family of process denotations of $Sp = (D, P)$ and $Sp = (D', P')$ respectively. The proof is carried out by contraposition.

Let $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$ contains a denotation with deadlock. From the refinement argument we know that:

$$\begin{aligned} & \mathbf{Mod}(D')|_{\sigma} \subseteq \mathbf{Mod}(D) \\ \text{and} \quad & \forall M' \in \mathbf{Mod}(D').\, d_{M'|_{\sigma}} = (T_{M'|_{\sigma}}, F_{M'|_{\sigma}}) \sqsubseteq_{\mathcal{F}} \hat{\alpha}_{\mathcal{F}}(T'_{M'}, F'_{M'}) = \hat{\alpha}_{\mathcal{F}}(d'_{M'}). \end{aligned}$$

We show that $d_{M'|_{\sigma}} = (T_{M'|_{\sigma}}, F_{M'|_{\sigma}})$ contain a denotation with deadlock. Let $\overline{M} \in I'$ such that $d'_{\overline{M}} = (T'_{\overline{M}}, F'_{\overline{M}})$ with $(s', \mathcal{A}lph(\overline{M})^{\checkmark}) \in F'_{\overline{M}}$, i.e., $d'_{\overline{M}}$ is a deadlocked process denotation. We unfold the reduct definition over the stable failure model $\mathcal{F}$:

$$\begin{aligned} \hat{\alpha}_{\mathcal{F}}(T'_{\overline{M}}, F'_{\overline{M}}) = (&\{s \in \mathcal{A}lph(M'|_{\sigma})^{*\checkmark} \mid \alpha^{*\checkmark}(s) \in T'_{\overline{M}}\}, \\ &\{(s, X) \in \mathcal{A}lph(M'|_{\sigma})^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M'|_{\sigma})^{\checkmark}) \mid \\ &\text{exists } (s', X') \in F'_{\overline{M}} \text{ with } \alpha^{*\checkmark}(s) = s' \text{ and } \alpha^{\checkmark}_{\mathbb{P}}(X) = X' \cap \alpha(\mathcal{A}lph(M'|_{\sigma}))\}). \end{aligned}$$

Since $d'_{\overline{M}}$ is a deadlocked process denotation, we have that $X' = \mathcal{A}lph(\overline{M})^{\checkmark}$. Unfolding the definition of stable failure refinement we have that

$$\hat{\alpha}_{\mathcal{T}}(T'_{\overline{M}}) \subseteq T_{\overline{M}|_\sigma} \text{ and } \hat{\alpha}(F'_{\overline{M}}) \subseteq F_{\overline{M}|_\sigma}.$$

It follows that $(s, \mathcal{A}lph(\overline{M})^{\checkmark}) \in F_{\overline{M}|_\sigma})$, this means that $d'_{\overline{M}}$ is a deadlocked process denotation. Hence, $(d_M)_{M\in I}$ contains a denotation with deadlock. ∎

Following an idea from the CSP context, we formulate the most abstract deadlock free CSP-CASL specification over a subsorted CASL signature $\Sigma = (S, TF, PF, P, \leq)$ – see [Mos04] for the details – with a set of sort symbols $S = \{s_1, \ldots, s_n\}, n \geq 1$:

ccspec DF_Σ =
  data
    ... declaration of Σ ...
  process
    $DF_S = \bigsqcap_{s:S}(!x :: s \rightarrow DF_S) \sqcap SKIP$
end

Here, the process DF_S can either internally choose to successfully terminate, or behave like $!x :: s \rightarrow DF_S$. The latter, internally chooses an element x from the sort s, engages in it, and then recursively behaves like DF_S. We observe:

LEMMA 8.1.3 DF_Σ *is deadlock free.*

PROOF. Let $(df_M)_{M\in I}$ be the denotation of DF_Σ over the stable-failures model, where $df_M = (T_M, F_M)$. For all $M \in I$ holds:

$$\begin{array}{lcl} T_M & = & \mathcal{A}lph(M)^{*\checkmark} \\ F_M & = & \{(t, X) \mid t \in \mathcal{A}lph(M)^*, X \subseteq \mathcal{A}lph(M) \vee \exists a \in \mathcal{A}lph(M).\, X \subseteq \mathcal{A}lph(M)^{\checkmark} - \{a\}\} \\ & & \cup\{(t \frown \langle\checkmark\rangle, Y) \mid t \in \mathcal{A}lph(M)^*, Y \subseteq \mathcal{A}lph(M)^{\checkmark}\}. \end{array}$$

That is after a non-terminating trace t, DF_Σ never has $\mathcal{A}lph(M)^{\checkmark}$ as its refusal set. Hence, DF_Σ is deadlock free. ∎

This result on DF_Σ extends to a complete proof method for deadlock freeness in CSP-CASL:

THEOREM 8.1.4 *A* CSP-CASL *specification* (D, P) *is deadlock free if and only if* $\mathrm{DF}_\Sigma \rightsquigarrow_{\mathcal{F}} (D, P)$.

PROOF. Let $(df_M)_{M\in\mathbf{Mod}(\Sigma)}$ and $(d_{M'})_{M'\in\mathbf{Mod}(D)}$ be the family of process denotations of DF_Σ and (D, P) respectively. We show both sides of the equivalence:

⇒) Let $(d_{M'})_{M'\in\mathbf{Mod}(D)}$ be deadlock free. We apply the decomposition theorem (Theorem 7.1.1) and prove first the data refinement and then the process refinement. The data refinement holds, as the model class of DF_Σ consists of all CASL models over Σ.

For the process refinement we show that for all $M' \in \mathbf{Mod}(D)$ it holds that:

$$df_{M'} = (T_{M'}, F_{M'}) \sqsubseteq_{\mathcal{F}} (T'_{M'}, F'_{M'}) = d_{M'}$$

To this end we show that $T'_{M'} \subseteq T_{M'}$ and $F'_{M'} \subseteq F_{M'}$ holds. The trace inclusion is trivial, as $T'_M = \mathcal{A}lph(M')^{*\checkmark}$, i.e., the set of traces of DF_Σ consists of all possible traces. The inclusion of the failures set holds for the similar reason: In the case of t being a non-terminating trace, i.e, $t \in \mathcal{A}lph(M')^*$, then the refusal set X needs to be a proper subset of $\mathcal{A}lph(M')^\checkmark$ – otherwise $(d'_{M'})_{M' \in \mathbf{Mod}(D)}$ contains a deadlocked denotation.

Thus, the failure set is also included in the failure of DF_Σ.

$\Leftarrow$) If $\mathrm{DF}_\Sigma \rightsquigarrow_{\mathcal{F}} (D, P)$, Lemma 8.1.3 and Theorem 8.1.2 imply that (D, P) is deadlock free.

■

8.2 Livelock analysis in CSP-CASL

For concurrent systems, divergence (or livelock) is regarded as an individual starvation, i.e., a particular process is prevented from engaging in any actions. As described in Chapter 2, in CSP, the failures/divergences model $\mathcal{N}$ is considered best to study systems with regard to divergence. The CSP process *DIV* represents this phenomenon: immediately, it can refuse every event, and it diverges after any trace. *DIV* is the least refined process in the $\sqsubseteq_{\mathcal{N}}$ model.

In the failures/divergences model $\mathcal{N}$, a process is modeled as a pair (F, D). Here, F represents the *failures*, while D collects all *divergences*. Let A be the alphabet. The process *DIV* has

$$(A^{*\checkmark} \times \mathbb{P}(A^\checkmark), A^{*\checkmark}) \in \mathbb{P}(A^{*\checkmark} \times \mathbb{P}(A^\checkmark)) \times \mathbb{P}(A^{*\checkmark})$$

as its semantics over the failure/divergences model $\mathcal{N}$.

Following these ideas, we define what it means for a CSP-CASL specification to be divergence free: Essentially, after carrying out a sequence of events, the denotation shall be different from *DIV*.

DEFINITION 8.2.1 *Let $(d_M)_{M \in I}$ be a family of process denotations over the failure divergence model, i.e, $d_M = (F_M, D_M) \in \mathcal{N}(\mathcal{A}lph(M))$ for all $M \in I$.*

- *A denotation d_M is* **divergence free** *if and only if :*

  C.1 $\forall s \in \mathcal{A}lph(M)^*.\{(t, X) \mid (s \frown t, X) \in F_M\} \neq \mathcal{A}lph(M)^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M)^\checkmark)$ *or*

  C.2 $\forall s \in \mathcal{A}lph(M)^*.\{t \mid (s \frown t) \in D_M\} \neq \mathcal{A}lph(M)^{*\checkmark}$.

- $(d_M)_{M \in I}$ *is* **divergence free** *if for all $M \in I$ it holds that d_M is divergence free.*

As in the case of analysis for deadlock freeness, also the analysis for divergence freeness can be checked trough refinement, this time over the model $\mathcal{N}$.

THEOREM 8.2.2 *Let $Sp = (D, P)$ and $Sp = (D', P')$ be two* CSP-CASL *specifications. Let the alphabet condition holds, i.e., $Alphabet(P') \subseteq \alpha(Alphabet(P))$. Let $Sp \rightsquigarrow^{\sigma}_{\mathcal{N}} Sp'$. If Sp is divergence free, then Sp' is divergence free.*

PROOF. Let $(d_M)_{M \in \mathbf{Mod}(D)}$ and $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$ be the family of process denotations of $Sp = (D, P)$ and $Sp = (D', P')$ respectively. The proof goes by contraposition.

Let $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$ contains a denotation with divergence, i.e., there exists a model $\overline{M} \in \mathbf{Mod}(D')$ such that $d'_{\overline{M}} = (F'_{\overline{M}}, D'_{\overline{M}})$ is divergent. Then, conditions **C.1** and **C.2** do not hold for $d'_{\overline{M}}$.

We show that $(d_M)_{M \in \mathbf{Mod}(D)}$ contains a denotation with divergence, i.e., there exists a model $\overline{N} \in \mathbf{Mod}(D')|_{\sigma}$ such that $d_{\overline{N}} = (F_{\overline{N}}, D_{\overline{N}})$ is divergent. From the refinement argument we know that:

$$\begin{aligned} & \mathbf{Mod}(D')|_{\sigma} \subseteq \mathbf{Mod}(D) \\ \text{and} \quad & \forall M' \in \mathbf{Mod}(D').\, d_{M'|_{\sigma}} = (F^{\perp}_{M'|_{\sigma}}, D_{M'|_{\sigma}}) \sqsubseteq_{\mathcal{N}} \hat{\alpha}_{\mathcal{N}}(F'^{\perp}_{M'}, D'_{M'}) = \hat{\alpha}_{\mathcal{N}}(d'_{M'}). \end{aligned}$$

Let $\overline{M} \in \mathbf{Mod}(D')$ such that $d'_{\overline{M}} = (F'^{\perp}_{\overline{M}}, D'_{\overline{M}})$ is divergent, i.e., there is $s' \in \mathcal{A}lph(\overline{M})$ with

$$\begin{aligned} & \{(t', X') \mid (s' \frown t', X') \in \hat{\alpha}(F'^{\perp}_{\overline{M}})\} = \mathcal{A}lph(\overline{M})^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(\overline{M})^{\checkmark}) \\ \text{and} \quad & \{t' \mid (s' \frown t') \in \hat{\alpha}(D'_{\overline{M}})\} = \mathcal{A}lph(\overline{M})^{*\checkmark}. \end{aligned}$$

Such trace s should already exist in $d_{\overline{M}|_{\sigma}}$, due to the refinement argument. Hence, in $d_{\overline{M}|_{\sigma}}$ there exist $s \in \mathcal{A}lph(\overline{M}\,|_{\sigma})$ with $\hat{\alpha}^{*\checkmark}(s') = s, \hat{\alpha}^{\checkmark *}(t') = t$ and $\hat{\alpha}^{\checkmark}_{\mathbb{P}}(X') = X$ such that

$$\begin{aligned} & \{(t, X) \mid (s \frown t, X) \in F^{\perp}_{\overline{M}|_{\sigma}}\} = \mathcal{A}lph(\overline{M}\,|_{\sigma})^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(\overline{M}\,|_{\sigma})^{\checkmark}) \\ \text{and} \quad & \{t \mid (s \frown t) \in D_{\overline{M}|_{\sigma}}\} = \mathcal{A}lph(\overline{M}\,|_{\sigma})^{*\checkmark}). \end{aligned}$$

Here, the translation $\hat{\alpha}$ gives the full alphabet, thanks to the reduct property (Theorem 6.1.8). Therefore, $(d_M)_{M \in \mathbf{Mod}(D)}$ contains a divergent process denotation.

■

As for the analysis of deadlock freeness we formulate the least refined divergence free CSP-CASL specification over a CASL signature Σ with a set of sort of symbols $S = \{s_1, \ldots, s_n\}$ with $n \geq 1$.

ccspec DIVF_{Σ} =
  data
    ... declaration of Σ ...
  process
    $DivF = (STOP \sqcap SKIP) \sqcap (\bigsqcap_{s:S} !x :: s \rightarrow DivF)$
end

DivF may deadlock at any time, it may terminate successfully at any time, or it may perform any event at any time, however, it will not diverge.

LEMMA 8.2.3 DIVF_Σ *is divergence free.*

PROOF. Let $(d_M)_{M\in I}$ be the semantics of DIVF_Σ over the failures/divergences model $\mathcal{N}$ where $d_M = (F_M^\perp, D_M) \in \mathcal{N}(\mathcal{A}lph(M))$.

We compute the failures and divergence component of d_M. Here, we need to compute the fixed point of DIVF_S. For the model $\mathcal{N}$ this is given by the componentwise intersection

$$\sqcap S = (\{\cap F \mid (F,D) \in S\}, \{\cap D \mid (F,D) \in S\})$$

S is the directed set of process under the refinement in the model $\mathcal{N}$. For the failures component, we have for all models M that: $F_M^\perp = \mathcal{A}lph(M)^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M)^\checkmark)$. As for the divergences component the intersection gives the empty set. Hence, we have that $d_M = (\mathcal{A}lph(M)^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M)^\checkmark), \emptyset)$. Such a denotation fulfills the condition **C.2** of the *divergence free* definition. Thus, DIVF_Σ is divergence free. ∎

Putting things together, we obtain a complete proof method for divergence freedom of CSP-CASL specifications:

THEOREM 8.2.4 *A* CSP-CASL *specification* (D,P) *is divergence free if and only if* $\text{DIVF}_\Sigma \rightsquigarrow_{\mathcal{F}} (D,P)$. *Here* Σ *is the signature of* D.

PROOF. We show both directions of the equivalence:

⇒) Now let (D,P) be divergence free. Assume that $\text{DIVF}_\Sigma \not\rightsquigarrow_{\mathcal{N}} (D,P)$. As the data part of DIVF_Σ refines to D, with our decomposition Theorem 7.1.1 we can conclude that $(D, DivF) \overset{\text{proc}}{\not\rightsquigarrow}_{\mathcal{N}} (D,P)$.

Let $(d_M)_{M\in\mathbf{Mod}(D)}$ be the semantics of $(D, DivF)$, where $d_M = (F_M, D_M)$, and $(d'_M)_{M\in\mathbf{Mod}(D)}$ be the semantics of (D,P), where $d'_M = (F'_M, D'_M)$. By definition of process refinement there exists a model $M \in \mathbf{Mod}(D)$ such that $F'_M \not\subseteq F_M$ or $D'_M \not\subseteq D_M$.

As $F_M = \mathcal{A}lph(M)^{*\checkmark} \times \mathbb{P}(\mathcal{A}lph(M)^\checkmark)$, see the proof of Lemma 8.2.3, we know that $F'_M \subseteq F_M$ holds. Therefore, we know that $D'_M \not\subseteq D_M$. As $D_M = \emptyset$, there exists a trace $t \in D'_M$ not ending with $\checkmark$, as the healthiness condition **D3** of the failures/divergences model asserts that for any trace $u' = u \frown \langle\checkmark\rangle \in D'_M$ also $u \in D'_M$. Applying healthiness condition **D1** we obtain $t \frown t' \in D'_M$ for all $t' \in \mathcal{A}lph(M)^{*\checkmark}$. With healthiness condition **D2** this results in

$$\{(t \frown t', X) \mid t' \in \mathcal{A}lph(M)^{*\checkmark}, X \in \mathbb{P}(\mathcal{A}lph(M)^\checkmark)\} \subseteq F'_M.$$

Hence, d'_M is not divergence free, as D'_M violates **C.2** – contradiction to (D,P) divergence free.

⇐) If $\text{DIVF}_\Sigma \rightsquigarrow_{\mathcal{N}} (D,P)$, Lemma 8.2.3 and Theorem 8.2.2 imply that (D,P) is divergence free.

∎

8.3 Summary

In this chapter we have presented proof techniques for the verification of properties of CSP-CASL specifications. Specifically, we have shown that refinement over certain CSP denotational models preserve some properties. This concept allows to verify properties already on abstract specifications – which in general are less complex than the more concrete ones. The properties, however, are preserved over the design steps.

We have illustrated how to analyze deadlock and livelock freeness in the context of CSP-CASL. To this end, we have first defined what it means for a CSP-CASL specification to be deadlock or livelock free . Finally, we have established a proof technique for deadlock and livelock freeness based on CSP-CASL refinement, which turns out to be complete.

PART III

CSP-CASL *based testing*

(CHAPTER ...9)

Theory of testing from CSP-CASL

Contents

In this chapter we describe the theory of testing from CSP-CASL specifications. We first illustrate the general idea of our approach. Here, we present the 'main challenges' of setting up the theory for CSP-CASL using the example of a binary calculator. In Section 9.2 and 9.4 we describe the theoretical framework of our approach.

The notions and results presented in this chapter have been published in [KRS07].

9.1 Challenges for CSP-CASL based testing

Software testing is recognized as a necessary means of program verification. Even when other program verification techniques such as static analyses and formal proofs are employed, testing is still considered necessary to complement these techniques, and to build greater confidence in the system being developed.

In contrast to the approaches mentioned in the background Chapter 5, here we are using a specification language with loose semantics which allows under-specification, refinement and enhancement. Hence, a proper testing theory which exploits such aspects is needed.

When dealing with testing based on formal specifications, there are some inherent challenges to be solved. We believe that these challenges are common to all sufficiently abstract specification formalisms. In order to illustrate such challenges, we consider as an SUT the binary calculator example described in Section 4.1 (Example 4.1.1). Here, the abstract specification (BCALC0) is step by step refined to a more concrete specification.

A testing framework should be capable to deal with incomplete and nondeterministic specifications. Basically, for CSP-CASL based testing we shall define:

- what is a CSP-CASL test case,
- the expected result of a test case with respect to a CSP-CASL specification (the *test evaluation*), and
- the execution result of a test cases with respect to a (black box) SUT (the *test verdict*).

A first challenge is to therefore to define these notions such that the evaluation of test cases reflect the development notions of CSP-CASL specifications. This means that test suites can be incrementally *extended* and *refined* according to the development notions as presented in Chapter 6.

For the calculator example, we recall the first high-level specification (specified in Chapter 4):

ccspec BCALC0 =
  data sort *Number*
    ops 0, 1 : *Number*;
      $\_+\_$: *Number* × *Number* →? *Number*
  channels *Button* : *Number*;
    Display : *Number*
  process *P0* : *Button*, *Display* ;
    $P0 = Button\ ?\ x :: Number \rightarrow P0 \sqcap Display\ !\ y :: Number \rightarrow P0$
end

Even for such loosely specified systems we would like to be able to derive meaningful tests. For example, we could design test cases which are used for setting up the interface between testing system and SUT. The testing framework should be able to cope with such a situation.

A more refined specification could require that the pressing of buttons and the display of digits strictly alternates:

$$P1 \ = \ Button?x :: Number \ \rightarrow \ Display!y :: Number \rightarrow P1$$

In the process $P1$ each input is directly followed by some output. For such a specification, we would like to be able to test exactly the mentioned property, namely that after each press of a button some digit is displayed.

An even more refined version requires that the first displayed digit is echoing the input, and the second displays the result of the computation:

$$\begin{aligned} P2 \ = \ & Button?x :: Number \rightarrow Display!x \rightarrow \ Button?y :: Number \\ & \rightarrow Display!(x + y) \rightarrow P2 \end{aligned}$$

Here, we would like to test for instance that after input of x the display shows x, and if after input of x and y the display shows the value of the term $x + y$. Such refinement

steps could occur, for example, when use cases which are derived from customer's wishes are integrated into the formal specification. Ideally, we would like to be able to re-use test cases on a more detailed level which have been designed for a more abstract level; since the refined specification is more precise than the abstract one, the outcome of testing should also be more precise. In particular, each test case developed for $P1$ should be reusable for $P2$.

In $P2$ it is still left open what the value of $x + y$ shall be. We haven't yet specified the arithmetic properties of addition. Such situations of under-specifications occur, e.g., in object-oriented design. Here, it is often the case that library functions are used whose exact functionality is specified at a later stage. As presented in Chapter 4 (Example 4.2.4), we add some suitable axioms to the data part.

ccspec BCALC3 =
  data sort $Number$
    ops $0, 1 : Number$;
      $\_ + \_ : Number \times Number \to ? Number$
    axioms $0 + 0 = 0;\ 0 + 1 = 1;\ 1 + 0 = 1$
  channels $Button : Number$;
    $Display : Number$
  process $P3 : Button, Display$;
    $P3 = Button?x :: Number \to Display!x$
      $\to Button?y :: Number \to Display!(x + y) \to P3$
end

In BCALC3, the specification does not constrain the SUT in the result of the $+$ operator. This is because we haven't specified yet what happened in the case of $1 + 1$. Such a situation might for example arise when the functionality of border cases or exceptions is not constrained in the basic specification. E.g., in many programming languages the value of an integer variable in case of overflow is not defined. However, we want to design test cases which cover the normal, non-exceptional behaviour, and to re-use these test cases later on. With such a specification, we expect to be able to test whether the calculator behaves correctly, e.g., for the input of 0 and 1.

Taking the standard arithmetic CARDINAL from the CASL library of Basic Datatypes [Mos04], we specify a one bit calculator where $1 + 1$ is seen as an arithmetic overflow and therefore is an undefined term.

ccspec BCALC4 =
  data CARDINAL [**op** $WordLength = 1 : Nat$]
    with sort $CARDINAL \mapsto Number$
  channels $Button : Number$;
    $Display : Number$
  process $P3 : Button, Display$;
    $P3 = Button?x :: Number \to Display!x$
      $\to Button?y :: Number \to Display!(x + y) \to P3$
end

For this specification all models of the data part are isomorphic, and in the process part there is no internal non-determinism. Such a specification can be completely tested.

9.2 Test case evaluation

As mentioned in the previous section each test case reflects some intentions described in the specification. Here, we introduce a *colouring* scheme to reflect the intentions of test cases. Intuitively, *green* test cases reflect *required* behaviour of the specification. *Red* test cases reflect *forbidden* behaviour of the specification. A test is coloured *yellow* if it depends on an *open design decision*, i.e., if the specification does neither require nor disallow the respective behaviour.

DEFINITION 9.2.1 *Let $Sp = (D, P)$ be a* CSP-CASL *specification such that D is consistent, and let $X = (X_s)_{s \in S}$ be a variable system over the signature Σ of the data part D. A* test case *T is any* CSP-CASL *process in the signature of Sp and the variable system X. The* colour *of a test case T with respect to Sp is a value $c \in \{green, red, yellow\}$, such that*

- *$colour_{Sp}(T) = green$ iff for all models $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to M$ it holds that:*
  1. *(**Traces condition**) $traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset \to \beta(M)})$ and*
  2. *(**Failures condition**) for all $tr = \langle t_1, \ldots t_n \rangle \in traces([\![T]\!]_\nu)$ and for all $1 \leq i \leq n$ it holds that:*
     $(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset \to \beta(M)})$
- *$colour_{Sp}(T) = red$ iff for all models $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to M$ it holds that:*

$$traces([\![T]\!]_\nu) \not\subseteq traces([\![P]\!]_{\emptyset:\emptyset \to \beta(M)})$$

- *$colour_{Sp}(T) = yellow$ otherwise.*

In other words: a test case T is *green*, if all models agree (1) that all its traces are possible system runs, and (2) the execution of such traces can't be refused. A test case T is *red*, if all models agree that not all of its traces are possible system runs and, finally, a test case T is *yellow*, if the execution of some possible system run can also lead to failure, or the process T has a trace which some models consider as a possible system run while others don't. In analysing red test cases, we only consider the traces condition, because the CSP semantics only consider refusals for possible system runs. A CSP-CASL specification with an inconsistent data part D does not lead to any colouring of test cases. The following proposition lists some simple properties of our colouring scheme.

PROPOSITION 9.2.2 *Let $Sp = (D, P)$ be a* CSP-CASL *specification. The following holds:*

1. *$colour_{Sp}(STOP) = green$.*

2. *With the specification* $Sp = (D, STOP)$ *all test cases different from STOP are coloured red.*
3. *If* T *and* T' *are test cases such that* $colour_{Sp}(T) = c$, $colour_{Sp'}(T') = c'$, *and* $traces([\![T']\!]_\nu) \subseteq traces([\![T]\!]_\nu)$ *for all models* $M \in \mathbf{Mod}(D)$ *and all variable evaluations* $\nu : X \to M$, *then* $c = green$ *implies* $c' = green$ *and* $c' = red$ *implies* $c = red$.

PROOF.

1. The test process *STOP* gives rise to the empty observation $\langle\rangle$. Hence, the conditions for the *green* colour, (1) and (2) trivially hold.
2. The trace set of $Sp = (D, STOP)$ is $\{\langle\rangle\}$, hence $[\![STOP]\!]_\nu \not\sqsubseteq_{\mathcal{T}} [\![T]\!]_\nu$ for any $T \neq_{\mathcal{T}} STOP$. Here, T being semantically different from *STOP* over the traces model ($=_{\mathcal{T}}$), carries over also to the other CSP models; i.e., $\neq_{\mathcal{T}} \Rightarrow \neq_{\mathcal{F}}$, and $\neq_{\mathcal{N}}$.
3. For all models $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to M$:
   - If $c = green$, i.e., $traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. Then, it follows
     $$traces([\![T']\!]_\nu) \subseteq traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}).$$
     For every $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T']\!]_\nu)$, we have $tr \in traces([\![T]\!])$, so that $(\langle t_1, \ldots, t_{j-1}\rangle, \{t_j\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$ follows for every $1 \leq j \leq k$. Thus $c' = green$.
   - If $c' = red$, then there exists $tr \in traces([\![T']\!]_\nu)$ such that $tr \notin traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. But then, since $traces([\![T']\!]_\nu) \subseteq traces([\![T]\!]_\nu)$, we have $tr \in traces([\![T]\!]_\nu)$ and $traces([\![T]\!]_\nu) \notin traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. Thus, $c = red$.

■

We now discuss the main sources of yellow test cases. Typical examples of open design decisions which lead to yellow test cases are the following:

Internal nondeterminism This means, that one action may have multiple outcomes. Let us consider the following CSP-CASL specification:

ccspec NONDETERMINISM =
  data sort S
    ops $a, b : S$
  process
    $Choice = a \to STOP \sqcap b \to STOP$
end

The order of a and b as a first action is left open. Consider the test case $T = a \to STOP$. The traces condition holds, as $traces([\![T]\!]_\nu) \subseteq traces([\![Choice]\!]_{\emptyset:\emptyset\to\beta(M)})$. However, at the first step of *Choice* neither the execution of a nor the execution of b can be guaranteed. This is due to the fact that *Choice* has the failure $(\langle\rangle, \{a\})$, this means that it can refuse to run the event a of T; hence the colour of T is yellow.

Loose specification In the same way, a heterogeneous model class of the data part can lead to a yellow test. Take for example the following CSP-CASL specification:

ccspec LOOSE =
data sorts *Signal*, *Number*
ops f : *Number* $\rightarrow$ *Number*;
0, 1 : *Number*;
continue, *shutDown* : *Signal*
process
$P =$ **if** $f(0) = 1$
then *continue* $\rightarrow$ *SKIP*
else *shutDown* $\rightarrow$ *STOP*
end

Here, the signal *continue* is sent in models where f is, e.g., the successor function. In other models, f might be the predecessor function and *shutDown* is sent. If the interpretation of f is still an open design decision in the current specification, there are two correct implementations behaving differently.

The second example illustrates also that the classification of a test process as *green*, *red*, or *yellow* is in general undecidable, as CASL includes full first order logic and arbitrary CASL predicates are allowed in case distinctions. As mentioned in Section 5.3, this is called the *test oracle problem* (see e.g., [Mac00][Mac99]).

9.3 Syntactic characterization for colouring CSP-CASL test cases

Using techniques originally developed in the context of full abstraction proofs for CSP [Ros98], the semantical definition of colouring test processes presented in the previous section has an equivalent syntactical characterisation for certain test processes. Here, we first show a syntactic encoding for the traces condition and then for the failures condition. Finally, we illustrate some properties of such syntactic encodings in the context of colouring CSP-CASL test cases.

9.3.1 Traces condition

A test processes T is called *linear* if it can be written as $T = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow STOP$. Concerning trace inclusion with respect to a linear test process T of length $n \geq 0$, we define the following system of process equations:

$$
\begin{array}{lcl}
Check_T & = & ((P \parallel T)[[R_{s_1}]] \ldots [[R_{s_h}]] \\
 & & \quad |[\, A \,]| \\
 & & count(n)) \setminus \{A\} \\
 & & \\
count(n : Nat) & = & \textbf{if } n = 0 \\
 & & \quad \textbf{then } OK \rightarrow STOP \\
 & & \textbf{else } a \rightarrow \ count(n-1)
\end{array}
$$

Here, $h \geqslant 0$ is the number of sorts in which T can possibly communicate.

To make this a valid process part of a CSP-CASL specification, we take the original data part and extend it conservatively, i.e., without losing any model of the data part. To this end we add a datatype *Nat* with the standard operations, a free type A consisting only of the constant a, a free type OK consisting only of the constant OK, and for each of the finitely many sorts $s_1 \ldots s_h$ in which T could possibly communicate a renaming predicate $R_{s_i} : s_i \times A$ with the axiom $\forall\, x : s_i \bullet R(x, a)$.

The following CASL specification illustrates how we extend the data specification in order to make $Check_T$ a valid CSP-CASL specification. We first import the NAT specification from the CASL library. DATA is the data specification of P, with only one sort S.

from BASIC/NUMBERS **get** NAT

spec CHECKTDATA =
  NAT **and** DATA
    then **%cons**
      free type $A ::= a$
      free type $OK ::= ok$
      pred $R : S \times A$
      $\bullet\ \forall\, x : S \bullet R(x, a)$
end

The idea behind $Check_T$ definition is as follows: The synchronous parallel operator $\parallel$ forces P and T to agree on all communications. Should P agree to execute the communications $t_1, \ldots, t_n$ of T in exactly this order, this results in a sequence of n communications. All these communications are renamed into a via the predicates $R_{s_i,A}$. The process *count* communicates OK after the execution of n $a's$. Hiding the communication a makes only this OK visible.

We now show that in order to check the first condition for the *green* test case we need to prove that $Check_T =_{\mathcal{T}} OK \rightarrow STOP$.

THEOREM 9.3.1 *Given a model $M \in \mathbf{Mod}(D)$ and a variable evaluation $v : X \rightarrow M$, then the*

following holds:

$$traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}) \iff [\![Check_T]\!]_\nu =_{\mathcal{T}} [\![OK \to STOP]\!]_\nu$$

PROOF. We prove both direction of the equivalence:

$\Rightarrow$) Let M be the model and $\nu : X \to M$ be the variable evaluation such that $traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. Then the trace set of the synchronous parallel in $Check_T$ is given by the intersection of the traces of T and P.

Using the assumption $traces([\![T]\!]_\nu) \subseteq traces[\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$, we have that,

$$traces([\![T]\!]_\nu \mid\mid [\![P]\!]_{\emptyset:\emptyset\to\beta(M)}) = traces([\![T]\!]_\nu).$$

Clearly, $traces([\![T]\!]_\nu[[R_{s_1}]] \dots [[R_{s_h}]]) = \underbrace{a \to \dots \to a}_{n \text{ times}} \to STOP$. Thus, we have that,

$$traces([\![(T[[R_{s_1}]] \dots [[R_{s_h}]]) \,|[\, A \,]|\, count(n)]\!]_\nu) = traces([\![\underbrace{a \to \dots \to a}_{n \text{ times}} \to OK \to STOP]\!]_\nu).$$

The next step is to hide all the a's; it follows that,

$$traces([\![(\underbrace{a \to \dots \to a}_{n \text{ times}} \to OK \to STOP) \setminus \{a\}]\!]_\nu) = traces([\![OK \to STOP]\!]_\nu).$$

Hence, $traces([\![Check_T]\!]_\nu) = traces([\![OK \to STOP]\!]_\nu)$.

$\Leftarrow$) The proof is done by contraposition. Let M be the data model such that $traces([\![T]\!]_\nu) \not\subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$ for some variable evaluation $\nu : X \to M$.

We have that $traces([\![T]\!]_\nu) = \{p \mid p \leq \langle [\![t_1]\!]_\nu, \dots, [\![t_n]\!]_\nu \rangle\}$. Then there exists a $k \in \mathbb{N}$ such that $0 \leq k < n$ with $\langle [\![t_1]\!]_\nu, \dots, [\![t_k]\!]_\nu \rangle \in traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$ and $\langle [\![t_1]\!]_\nu, \dots, [\![t_{k+1}]\!]_\nu \rangle \notin traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. This means in the synchronous parallel we have:

$$traces([\![T]\!]_\nu \mid\mid [\![P]\!]_{\emptyset:\emptyset\to\beta(M)}) = traces([\![t_1 \to \dots \to t_k \to STOP]\!]_\nu).$$

Applying the renaming operators we obtain,

$$traces(([\![T]\!]_\nu \mid\mid [\![P]\!]_{\emptyset:\emptyset\to\beta(M)})[[R_{s_1}]] \dots [[R_{s_h}]]) = traces([\![\underbrace{a \to \dots \to a}_{k \text{ times}} \to STOP]\!]_\nu).$$

Thus, we have that,

$$traces([\![\overbrace{a \to \dots \to a}^{k \text{ times}} \to STOP \,|[\, A \,]|\, count(n)]\!]_\nu) = traces([\![STOP]\!]_\nu).$$

Hence, $[\![STOP]\!]_\nu \neq_{\mathcal{T}} [\![OK \to STOP]\!]_\nu$.

■

9.3.2 Failures condition

Concerning the characterisation of the failures condition, we assume that the traces condition for the linear test process T already has been established. Here, we use a technique based on a refinement checking between two process equations to establish if each event in the test case is refused or not by a CSP-CASL process.

Let $Sp = (D, P)$ be a CSP-CASL specification and $T = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow STOP$ a liner test process. Here, we assume the trace condition holds; that is for all $M \in \mathbf{Mod}(D)$ and all variable evaluation $\nu : X \rightarrow M$ it holds $traces(\llbracket T \rrbracket_\nu) \subseteq traces \llbracket P \rrbracket_{\emptyset:\emptyset \rightarrow \beta(M)})$. Let $\mathcal{A}lph(M)$ be the alphabet of communication constructed over the model $M \in \mathbf{Mod}(D)$. The main idea is to test locally if each event t_i of the test case T is accepted or refused by the process P. We call this test *Local Refusal Test* (LRT).

In order to perform a local refusal test at position i of the test case T defined over a CSP-CASL specification $Sp = (D, P)$, $1 \leq i \leq n$, we construct the following processes:

$$
\begin{array}{l}
\mathrm{T}_i = t_1 \rightarrow \ldots \rightarrow t_i \rightarrow STOP \\
\\
\mathrm{RTEST}_i = (\mathrm{P} \mid\mid \mathrm{PRETEST}_i) \mid\mid \mathrm{T}_i \\
\mathrm{PRETEST}_i = t_1 \rightarrow \ldots \rightarrow t_{i-1} \rightarrow \mathrm{RUN}_D \\
\\
\mathrm{CHECKER}_i(n) = \textbf{if } n = 1 \\
\qquad\qquad\qquad \textbf{then } t_i \rightarrow STOP \\
\qquad\qquad\quad \textbf{else}((\sqcap_{s \in S} x :: s \rightarrow Checker_i(n-1)) \sqcap STOP)
\end{array}
$$

The process T_i represents the so far tested events of T. This mean that the trace $\langle t_1, \ldots, t_{i-1} \rangle$ is a possible trace of P and that each event $t_1, \ldots, t_{i-1}$ is not in the refusal set of P. Now, the purpose is to test if the event t_i is refused or not by P. In order to do this, we need to check the refinement $\mathrm{CHECKER}_i(i) \overset{\mathtt{proc}}{\rightsquigarrow}_{\mathcal{F}} \mathrm{RTEST}_i$. If the refinement holds, it means the event t_i cannot be refused by P otherwise the event t_i is in the refusal set of P.

In RTEST_i we run the process P in parallel with $\mathrm{PRETEST}_i$ and the process T_i. The process $\mathrm{PRETEST}_i$ allow us to make progress in the process P until the point we would like to perform the local refusal test. Here, the process RUN_D is of the form:

$$\mathrm{RUN}_D = \Box_{s \in S} ?x :: s \rightarrow \mathrm{RUN}_D \,\Box\, SKIP$$

It is a process which is always prepared to communicate an event from $s \in S$ or to terminate successfully. The process $\mathrm{CHECKER}_i(n)$ is parameterised by the trace length n of the test case T. Such process, recursively chooses internally an event $x :: s$ and engages in it; otherwise, it has the possibility to internally choose to deadlock. Here S is the sort set of the signature Σ of the data part D.

Assuming the traces condition for the test case T holds, the local refusal test is determined by the decision procedure described in Figure 9.1.

$$
\begin{array}{l}
\textbf{for}(i = 1, i \leqslant n, i{+}{+})\{ \\
\quad \textbf{if}(\textsc{Checker}_i(i) \overset{proc}{\not\rightsquigarrow_{\mathcal{F}}} \textsc{RTest}_i \textit{ holds}) \\
\qquad\qquad \textbf{then}(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \in \textit{failures}(P) \\
\quad \textbf{else}(\langle t_1, \ldots, t_i - 1\rangle, \{t_i\}) \notin \textit{failures}(P) \\
\}
\end{array}
$$

Figure 9.1: Local refusal test – LRT.

DEFINITION 9.3.2 (SUCCESSFUL LRT) *We say that a local refusal test (*LRT*) of a test case T for a* CSP-CASL *process P is* **successful**, *if for all events t_i of T the refinement* $\textsc{Checker}_i(i) \overset{proc}{\rightsquigarrow_{\mathcal{F}}} \textsc{RTest}_i$ *holds. We say an* LRT *of a test case T for a* CSP-CASL *process P is* ***not successful*** *if exists an event t_i of T such that the refinement* $\textsc{Checker}_i(i) \overset{proc}{\rightsquigarrow_{\mathcal{F}}} \textsc{RTest}_i$ *doesn't hold.*

In the following theorem, we prove that in order to check for the failures condition of a test case we need to check the local refusal test. Here, we illustrate for a single event t_i at the position i of the test case.

THEOREM 9.3.3 *Let $Sp = (D, P)$ be a* CSP-CASL *specification.*
Let $T = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow STOP$ be a CSP-CASL *test case for Sp. Then for all $1 \leq i \leq n$, the following holds:*

$$
\begin{array}{c}
(D, \textsc{Checker}_i(i)) \overset{proc}{\rightsquigarrow_{\mathcal{F}}} (D, \textsc{RTest}_i) \\
\Longleftrightarrow \\
\textit{for all } M \in \textbf{\textit{Mod}}(D) \textit{ and } \nu : X \rightarrow M.\ (\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\} \notin \textit{failures}([\![P]\!]_{\emptyset:\emptyset\rightarrow\beta(M)}).
\end{array}
$$

PROOF. We show both directions of the equivalence.

$\Longrightarrow$) (By contradiction.) Let $(D, \textsc{Checker}_i(i)) \overset{proc}{\rightsquigarrow_{\mathcal{F}}} (D, \textsc{RTest}_i)$. Let $M \in \mathbf{Mod}(D)$ and variable evaluation $\nu : X \rightarrow M$. Then, from the process refinement over the stable failure model we have $\textit{failures}([\![\textsc{RTest}_i]\!]_\nu) \subseteq \textit{failures}([\![\textsc{Checker}_i(i)]\!]_\nu)$.

We now calculate the failures set of $\textsc{Checker}_i(i)$:

$$
\begin{array}{rl}
\textit{failures}([\![\textsc{Checker}_i(i)]\!]_\nu) = & \bigcup_{1\leq j\leq i-2}\{(\langle a_1, \ldots, a_{j-1}\rangle, X) \mid \langle a_1, \ldots, a_{j-1}\rangle \in \mathcal{A}lph(M)^* \\
& \qquad\qquad \text{and } X \subseteq \mathcal{A}lph(M)^{\checkmark}\} \\
& \cup\{(\langle a_1, \ldots, a_{i-1}\rangle, X) \mid \langle a_1, \ldots, a_{i-1}\rangle \in \mathcal{A}lph(M)^* \\
& \qquad\qquad \text{and } t_i \notin X \subseteq \mathcal{A}lph(M)^{\checkmark}\} \\
& \cup\{(\langle a_1, \ldots, a_{i-1}, t_i\rangle, X) \mid \langle a_1, \ldots, a_{i-1}\rangle \in \mathcal{A}lph(M)^* \\
& \qquad\qquad \text{and } X \subseteq \mathcal{A}lph(M)^{\checkmark}\}
\end{array}
$$

We have that $(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin \textit{failures}([\![\textsc{RTest}_i]\!]_\nu$, since we have that $\textit{failures}([\![\textsc{RTest}_i]\!]_\nu) \subseteq \textit{failures}([\![\textsc{Checker}_i(i)]\!]_\nu)$ and the event t_i is not in the refusal set of $\textsc{Checker}_i(i)$.

Assume that $(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \in \mathit{failures}(\llbracket P \rrbracket_{\emptyset:\emptyset \to \beta(M)})$, then it follows that $(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \in \mathit{failures}(\llbracket \text{RTEST}_i \rrbracket_\nu)$. The latter is a contradiction to our earlier assertion. Hence, $(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin \mathit{failures}(\llbracket P \rrbracket_{\emptyset:\emptyset \to \beta(M)})$.

$\Longleftarrow$) (By contradiction.) Let $(\langle \nu^\sharp(t_1), \ldots, \nu^\sharp(t_{i-1})\rangle, \{\nu^\sharp(t_i)\}) \notin \mathit{failures}(\llbracket P \rrbracket_{\emptyset:\emptyset \to \beta(M)})$.

Assume that $(D, \text{CHECKER}_i(i)) \overset{\text{proc}}{\not\rightsquigarrow_{\mathcal{F}}} (D, \text{RTEST}_i)$. It follows that $\mathit{failures}(\llbracket \text{RTEST}_i \rrbracket_\nu) \not\subseteq \mathit{failures}(\llbracket \text{CHECKER}_i(i) \rrbracket_\nu)$. This means that there is a failure set in RTEST_i which is not present in the failures set of $\text{CHECKER}_i(i)$.

This is the case, when $(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \in \mathit{failures}(\llbracket \text{RTEST}_i \rrbracket_\nu)$. It follows that $(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \in \mathit{failures}(\llbracket P \rrbracket_{\emptyset:\emptyset \to \beta(M)})$, which contradicts our assertion. Hence, $(D, \text{CHECKER}_i(i)) \overset{\text{proc}}{\not\rightsquigarrow_{\mathcal{F}}} (D, \text{RTEST}_i)$.

■

From the above theorem we obtain the following corollary, which basically states that checking for the failures condition boils down to test the 'successfulness' of all local refusal test.

COROLLARY 9.3.4 *Let $Sp = (D, P)$ be a* CSP-CASL *specification.*
Let $T = t_0 \to \ldots \to t_n \to STOP$ be a CSP-CASL *test case for Sp. Let the traces condition holds for T, i.e., for all $M \in \mathbf{Mod}(D)$ and all variable evaluation $\nu : X \to M$ it holds $\mathit{traces}(\llbracket T \rrbracket_\nu) \subseteq \mathit{traces}(\llbracket P \rrbracket_{\emptyset:\emptyset \to \beta(M)})$. Then the following holds:*

1. LRT *is successful if and only if $\mathit{colour}_{Sp}(\llbracket T \rrbracket_\nu) = \mathit{green}$.*
2. LRT *is not successful if and only if $\mathit{colour}_{Sp}(\llbracket T \rrbracket_\nu) = \mathit{yellow}$.*

The following corollary summarizes the colouring of a CSP-CASL test process using the syntactic characterization introduced in the last sections.

COROLLARY 9.3.5 *Let $Sp = (D, P)$ a* CSP-CASL *specification. Let $T = t_1 \to \ldots \to t_n \to STOP$ be a linear test process for Sp, and let $Var(T) = \{x_1 : s1, \ldots, x_k : s_k\}$ be the variables occurring in T. Then the colour of T is:*

- **Green** *iff the following two conditions holds:*

$$a) \quad (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, \mathit{Check}_T) \\ =_{\mathcal{T}} \\ (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, \mathit{OK} \to \mathit{STOP})$$

and

$$b) \quad a) \Rightarrow (D, \text{CHECKER}_i(i)) \overset{\text{proc}}{\rightsquigarrow_{\mathcal{F}}} (D, \text{RTEST}_i) \text{ holds for all } 1 \leq i \leq n.$$

- **Yellow** *iff the following two conditions holds:*

$$\begin{array}{ll} a) & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, Check_T) \\ & \qquad =_{\mathcal{T}} \\ & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, OK \rightarrow STOP) \\ \textit{and} & \\ b) & a) \Rightarrow \textit{exists an } i \textit{ such that } (D, \text{CHECKER}_i(i)) \overset{proc}{\not\rightsquigarrow_{\mathcal{F}}} (D, \text{RTEST}_i). \end{array}$$

Moreover, for a monomorphic data D, the colour of T is:

1. **Green** *iff the following two conditions holds:*

$$\begin{array}{ll} a) & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, Check_T) \\ & \qquad =_{\mathcal{T}} \\ & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, OK \rightarrow STOP) \\ \textit{and} & \\ b) & a) \Rightarrow (D, \text{CHECKER}_i(i)) \overset{proc}{\rightsquigarrow_{\mathcal{F}}} (D, \text{RTEST}_i) \textit{ holds for all } 1 \leq i \leq n. \end{array}$$

2. **Red** *iff the following condition hold:*

$$\begin{array}{ll} a) & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, Check_T) \\ & \qquad \neq_{\mathcal{T}} \\ & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, OK \rightarrow STOP) \end{array}$$

3. **Yellow** *iff the following two conditions holds:*

$$\begin{array}{ll} a) & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, Check_T) \\ & \qquad =_{\mathcal{T}} \\ & (\{\text{CHECKTDATA } \textbf{then op } x_1 : s_1, \ldots, x_k : s_k\}, OK \rightarrow STOP) \\ \textit{and} & \\ b) & a) \Rightarrow \textit{exists an } i \textit{ such that } (D, \text{CHECKER}_i(i)) \overset{proc}{\not\rightsquigarrow_{\mathcal{F}}} (D, \text{RTEST}_i). \end{array}$$

Having a monomorphic data specification and a deterministic process, test cases for such specifications are coloured either green or red and not yellow. Intuitively this means that the specification has already resolved all the open design decisions, and for the process part means we do not have an internal non-determinism.

Colouring a test case is performed using CSP-CASL-PROVER. Here, we use syntactic encoding for the traces and failures condition to prove the colour of a test case with respect to a CSP-CASL specification. In the following example we illustrate a colouring proof script in CSP-CASL-PROVER. Here, we use the binary calculator specification.

EXAMPLE 9.3.6 Let us consider BCALC4, and the following test case $T1$:

$$T1 = \mathit{Button}!0 \rightarrow \mathit{Display}!0 \rightarrow \mathit{Button}!1 \rightarrow \mathit{Display}!1 \rightarrow \mathit{STOP}$$

We show a script of CSP-CASL-PROVER that proves $colour_{BCalc4}(T1) = Green$. For the latter we need to show that the traces condition and the failures condition holds.

For the traces condition, we use Theorem 9.3.1 to prove the traces condition. Here, we need to show:

$$Check_T =_{\mathcal{T}} OK \rightarrow STOP$$

In order to encode the syntactic encoding $Check_T$, we enrich the event set by adding the events 'A' and 'OK':

```
datatype Event = Button Number | Display Number | A | OK
```

Here, *Button Number* and *Display Number* are the two channels used for the communication. In the following we encode $Check_T$:

```
consts CheckT :: "('p, Event) proc"
defs CheckT_def : " TestColour ==
          ( ((P3 || T1 )  [[ MyRenaming ]] )
                  |[ {A} ]|
                Count4) -- {A}"
```

Here, $MyRenaming = \{(x, A) \mid x \in \Sigma\}$ is the process which renames every event to 'A', and $Count4$:

```
consts Count4 ::  "('p, Event) proc"
defs Count4_def: " Count4 == A -> A -> A -> A -> OK -> STOP"
```

The equations $Check_T =_T OK \rightarrow STOP$ can be shown by systematically proving some auxiliary lemmas:

1)	$P3 \,\|\|\, T1 =_{\mathcal{T}} T1$	Parallel_one
2)	$T1[[MyRenaming]] =_{\mathcal{T}} A \rightarrow A \rightarrow A \rightarrow A \rightarrow STOP$	Renaming
3)	$A \rightarrow A \rightarrow A \rightarrow A \rightarrow STOP =_{\mathcal{T}} Count4$	Parallel_two
4)	$Count4 \setminus \{A\} =_{\mathcal{T}} OK \rightarrow STOP$	Hiding

In step 1, we prove that indeed the process $P3$ agree on the events prescribed in the test case $T1$. In step 2, we rename all the actions to A's. In step 3, the process $Count4$ verifies that there are four agreed actions and communicates the event OK. In the last step we hide all the A's in order to make the event OK visible.

All these steps are formalized in CSP-CASL-PROVER in the following script:

```
theorem TraceCondition : "CheckT =T OK -> STOP"
   apply(simp add: CheckT_def)
   apply(cspT_simp Parallel_one)
   apply(simp add: T1_def)
   apply(cspT_simp Renaming)
   apply(simp add: Count4_def)
   apply(cspT_simp Parallel_two)
   apply(cspF_simp Hiding)
done
```

The proof is discharged using tactics developed in the context of CSP-PROVER. Such tactics use CSP algebraic laws in order to prove the equivalence of processes and the process refinement by syntactically rewriting process expressions.

In theorem *TraceCondition*, the command (`simp add:__`) allows us to unfold the definition of the different processes, e.g., *CheckT_def*. The tactic (`cspT_simp __`) takes care of the rewriting process. This tactic takes as a parameter the name of a lemma to be applied, e.g., *Parallel_one*. These four lemmas can be proven following a systematic approach. The CSP-PROVER tactic (`cspT_simp __`) is usually able to prove simple equations. Adding a '+' to a proof command triggers its repeated execution till it fails. The detailed proof script of these lemmas is reported in Appendix A.3.

We then prove that the failures condition holds. Here, we have to make sure that the LRT for each event is successful. Let us consider the LRT for the first event (*Button*!0), the following proof script illustrate the steps to prove that the LRT is successful.

```
theorem LRT_Button0: "Check_Button0  <=F RTest_Button0"
   apply(unfold Check_Button0_def RTest_Button0_def)
   apply(cspF_auto | auto | cspF_hsf | rule cspF_decompo)+
   apply(rule cspF_rw_right)
   apply(rule cspF_decompo)
   apply(simp)
   apply(rule cspF_IF)
   apply(cspF_auto | auto | rule cspF_decompo | rule cspF_Int_choice_left1)+
done
```

Here, in *Check_Button*0 and *RTest_Button*0 we encode the processes CHECKER and RTEST respectively for the LRT of the first event. ■

9.4 Test case execution

Here, we define the execution of a test process with respect to a particular SUT. The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test process.

A *point of control and observation* (PCO) $\mathcal{P} = (\mathcal{A}, \|...\|, \mathcal{D})$ of a SUT consists of

- an alphabet $\mathcal{A}$ of primitive events which can be communicated at this point,
- a mapping $\|\cdot\| : \mathcal{A} \longrightarrow T_\Sigma(X)$ which returns for each $a \in \mathcal{A}$ a term (usually a constant) over Σ, and
- a direction $D : \mathcal{A} \longrightarrow \{ts2sut, sut2ts\}$.

*ts*2*sut* stands for signals which are sent from the testing system to the system under test, and *sut*2*ts* stands for signals which are sent in the other direction. In telecommunications, the mapping $\|...\|$ is called a *coding rule*. For the data type definition language ASN.1 (Abstract Syntax Notation One) [Dub00] there are standardized coding rules for many frequently used PCOs. In Section 12.4 we show a concrete use of ASN.1 as coding rules.

We now define conditions for a test case to be executable.

DEFINITION 9.4.1 (EXECUTABLE TEST CASE) *A test case T is* executable *at a PCO $\mathcal{P}$ with respect to a specification $Sp = (D, P)$, if*

1. *for each term t occurring in T there is exactly one $a_t \in \mathcal{A}$ such that a_t and t are equal,*
2. *for $a, b \in \mathcal{A}$, if $\|a\|$ equals $\|b\|$ then a and b are the same primitive event.*

Condition 1 ensures that each term in the test case corresponds to some observable or controllable event in the SUT. This is the case if a_t and t are of the same sort and

$$D \models (\forall X. \|a_t\| = t)$$

Here, X is a variable system including all variables of $\|a_t\|$ and t. Since in general equality of CASL terms is undecidable, in general it is undecidable if an arbitrary test case is executable with respect to a PCO. However, for all practical purposes equality is easily decidable. Condition 2 ensures that different observations or control events represent different values.

For test execution, we consider the SUT to be a process over the alphabet $\mathcal{A}$, where the internal structure is hidden. Hence, the SUT can engage in communications at the PCO. Communications a with $D(a) = sut2ts$ are initiated by the SUT and are matched by the testing system with the expected event from the test case. Communications a with $D(a) = ts2sut$ are initiated by the testing system and cannot be refused by the SUT. Figure 9.2 illustrate the notion of test direction.

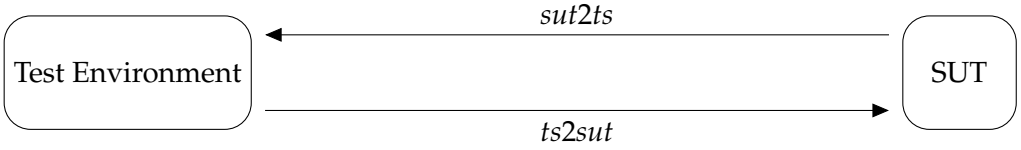

Figure 9.2: Direction of test events.

If the SUT sends an event without a stimulus, the SUT deviates from the specified behaviour. If the SUT internally refuses some communication, this can only be observed by the fact that it doesn't answer, i.e., the testing system waits for some event *sut*2*ts*, but this event does not happen. Testing is concerned with safety properties only; thus we say that in such a case a *timeout* happens.

The *test verdict* of a test case is defined relatively to a particular CSP-CASL specification and a particular SUT. The verdict is either *pass*, *fail* or *inconclusive*. Intuitively, the verdict *pass* means that the test execution increases our confidence that the SUT is correct with respect to the specification. The verdict *fail* means that the test case exhibits a fault in the SUT, i.e., a violation of the intentions described in the specification. The verdict *inconclusive* means that the test execution neither increases nor destroys our confidence in the correctness of the SUT.

Let $T = (t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow STOP)$ be a linear test case. Assume $colour(T) = c$ with respect to a specification (D, P). Assume further that T is executable at a PCO $\mathcal{P} = (\mathcal{A}, \|...\|, \mathcal{D})$. The test verdict of the test case T with colour c at the PCO $\mathcal{P}$ relatively to an execution of the SUT is defined inductively as follows:

- If $n = 0$ the colour c of the test case yields the test verdict as follows: if $c = green$ the test verdict is *pass*, if $c = red$ the test verdict is *fail*, if $c = yellow$ the test verdict is *inconclusive*.
- If $n > 0$, let a be the primitive event with $\|a\|$ equals t_1. Assume that the colour c is
  - *green*: If the direction $D(a) = sut2ts$ and we receive a, then we inductively determine the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow STOP)$.

    If the direction $D(a) = sut2ts$ and we receive some b different from a or if a timeout occurs, then the test verdict is *fail*.

    If the direction $D(a) = ts2tsut$ and we receive an event from the SUT within the timeout period, then the test verdict is *fail*.

    If the direction $D(a) = ts2sut$ and we do not receive an event during the timeout period, then we send a to the SUT and obtain the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow STOP)$.
  - *red*: If the direction $D(a) = sut2ts$ and we receive a we obtain the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow STOP)$. If the direction $D(a) = sut2ts$ and we receive some b different from a or if a timeout occurs, then the test verdict is *pass*.

    If the direction $D(a) = ts2tsut$ and we receive an event from the SUT within the timeout period, then the test verdict is *pass*.

    If the direction $D(a) = ts2sut$ and we do not receive an event during the timeout period, then we send a to the SUT and obtain the test verdict by continuing to execute the SUT against the remaining linear test case $(t_2 \rightarrow \ldots \rightarrow t_n \rightarrow STOP)$.
  - *yellow*: the test verdict is *inconclusive*.

The verdict of a yellow test case is always *inconclusive* and does not require any execution of the SUT. Recall that a yellow test case reflects an open design decision. Consequently, such a test case can neither reveal a deviation from the intended behaviour, nor can it increase the confidence that the system is apt to its intended use. After taking this design decision, however, i.e., turning the property into an intended or a forbidden one, the colour of the test will change and we will obtain *pass* or *fail* as a verdict.

9.5 Summary

In this chapter, we have presented a theory for the evaluation of test cases with respect to CSP-CASL specifications. The major innovations are the separation of the test oracle and the test evaluation problem by defining:

- the expected result (*green*, *red* and *yellow*) and,
- the verdict (*pass*, *fail* and *inconclusive*) of a test case.

The CSP-CASL specification determines the alphabet of the test suite, and the expected result of each test case. The expected result of a test case, in terms of the colouring scheme, is proved using CSP-CASL-PROVER.

The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test processes. Here, we have defined an algorithm which allows to determine the verdict of the test case on the fly. Figure 9.3 illustrate the general overview of our testing approach.

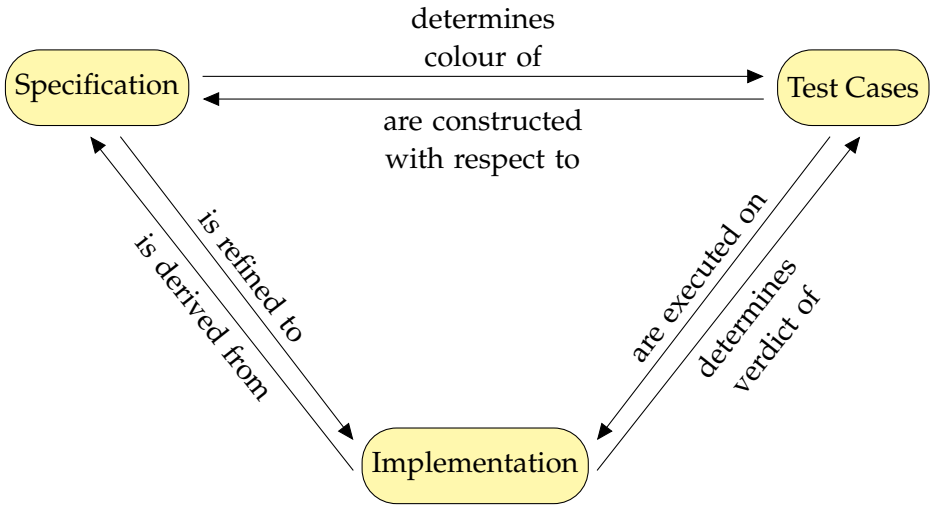

Figure 9.3: CSP-CASL validation triangle.

Specification, *Implementation* and *Test Cases* are mutually related artifacts. *Specifications* and *Test Cases* are written in CSP-CASL, the *Implementation* is treated as a black box. *Test cases* can be constructed either from the specification – as shown in the triangle – or independently from it. The specification determines the alphabet of the test suite, and the expected result of each test case. The expected result is coded in a colouring scheme of test cases. If a test case is constructed which checks for the presence of a required feature (according to the specification), we define its colour to be *green*. If a test case checks for the absence of some unwanted behaviour, we say that it has the colour *red*. If the specification does neither require nor disallow the behaviour tested by the test case, i.e., if a SUT may or may not implement this behaviour, the colour of the test case is defined to be *yellow*. Here, we

have defined a syntactic characterization in order to colour a CSP-CASL test process. The colouring of a CSP-CASL test process is done in CSP-CASL-PROVER.

During the execution of a test on a particular SUT, the *verdict* is determined by comparing the colour of the test case with the actual behaviour. A test *fails*, if the colour of the test case is green but the SUT does not exhibit this behaviour, or if the colour is red but the behaviour can be observed in the SUT. The execution of a yellow test case yields an inconclusive verdict. A test *passes*, if the colour is green and the SUT exhibit this behavior, or if the colour is red and the SUT doesn't exhibit this behavior.

(CHAPTER ...10)

Testing and CSP-CASL development notions

Contents

In this chapter we study how the CSP-CASL testing theory relates to the notion of vertical and horizontal development of CSP-CASL specifications. In Section 10.1 we study how refinement of specifications and testing relate to each other. In Section 10.2 we illustrate how the notion of specification enhancement relate to testing and how we can re-use test cases in a software product line. In Section 10.3 we illustrate these notions with a simple case study of remote control unit for home appliances.

The notions and results presented in this chapter have been published in [KRS07] and [KRS08].

10.1 Testing and CSP-CASL refinement

In this section, we show the relation between CSP-CASL refinement notion and the evaluation of test cases. In particular we show the preservation of the colour of a test case under a well-behaved refinement notion.

Let $\leq$ be a binary relation over CSP-CASL specifications such that $(D, P) \leq (D', P')$. We call such a relation to be $\leq$ *well-behaved*, if, given specifications $(D, P) \leq (D', P')$ with consistent data parts D and D' and a variable system X over the signature of D, the following holds for any test process T over D:

1. $colour_{(D,P)}(T) = green$ implies $colour_{(D',P')}(T) = green$, and

2. $colour_{(D,P)}(T) = red$ implies $colour_{(D',P')}(T) = red$.

Interpreting $\leq$ as a development step, this means: If a test case T reflects a desired behavioural property in (D, P), i.e., $colour_{(D,P)}(T) = green$, after a well-behaved development step from (D, P) to (D', P') this property remains a desired one and the colour of T is green. If a test case reflects a forbidden behavioural property in (D, P), i.e., $colour_{(D',P')}(T) = red$, after a well-behaved development step from (D, P) to (D', P') this property remains a forbidden one and the colour of T is red. A well-behaved development step can change only the colour of a test case T involving an open design decision, i.e., $colour_{(D,P)}(T) = yellow$.

In the following, we study for various refinement relations if they are well-behaved.

THEOREM 10.1.1 (DATA REFINEMENT IS WELL-BEHAVED) *Let $Sp = (D, P)$ and $Sp' = (D', \rho(P))$ be* CSP-CASL *specifications such that $Sp \overset{data}{\leadsto_\sigma} Sp'$ hold via data refinement, i.e., $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$. Then, $\overset{data}{\leadsto_\sigma}$ is well-behaved in the chosen* CSP *model $\mathcal{D} \subseteq \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$.*

PROOF. We need to show that the colour of a test case T over Sp remains unchanged over Sp' after a data refinement.

Let $\sigma : \Sigma \to \Sigma'$ be the CSP-CASL data logic signature morphism. We consider the cases of the *green* and *Red* test cases.

- Let $colour_{Sp}(T) = green$. We show that $colour_{Sp'}(\rho(T)) = green$.

  Let $M' \in \mathbf{Mod}(D')$. From the data refinement argument we know that $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$. This implies that M' is also a model of D. For all models of D, the *traces* and *failures* conditions for a green test case hold by assumption. For the traces condition we have that:

  $$traces([\![T]\!]_{\nu:X\to\beta(M'|_\sigma)}) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\sigma)}).$$

  Thanks to the reduct property (Theorem 6.1.8) we have that:

  $$\begin{aligned} traces([\![T]\!]_{\nu:X\to\beta(M'|_\sigma)}) &= \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')}) \\ traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\sigma)}) &= \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')}). \end{aligned}$$

  This implies that $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')}) \subseteq \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')})$. Eliminating the inverse alphabet translation $\hat{\alpha}_{\mathcal{T}}$ from both sides we obtain:

  $$traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')} \subseteq traces([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')}. \qquad (*)$$

  For the failures condition we have that for all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_{\nu:X\to\beta(M'|_\sigma)})$ and for all $1 \leq i \leq n$ it holds that:

  $$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\sigma)}).$$

  Thanks to the reduct property we have that:

  $$failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\sigma)}) = \hat{\alpha}_{\mathcal{F}}(failures([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')})).$$

This implies that for all $tr = \langle t_1, \ldots t_n \rangle \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')}))$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \notin \hat{\alpha}_{\mathcal{F}}(failures([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M'|_\sigma)})). \qquad (**)$$

Putting together the results of the traces (*) and failures (**) condition it follows that $colour_{Sp'}(\rho(T)) = green$.

- Let $colour_{Sp}(T) = red$. We show that $colour_{Sp'}(\rho(T)) = red$.

  Let $M' \in \mathbf{Mod}(D')$. From the data refinement argument we know that $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$. This implies that M' is also a model of D. Moreover, we know that:

  $$traces([\![T]\!]_{:X\to\beta(M'|_\sigma)}) \not\subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}).$$

  Thanks to the reduct property we have that:

  $$traces([\![T]\!]_{\nu:X\to\beta(M'|_\sigma)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')})$$
  $$traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\sigma)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')}).$$

  Then it follows that $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')}) \not\subseteq \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')})$. Thus, $traces([\![\rho(T)]\!]_{\nu:X\to\beta(M')} \not\subseteq traces([\![\rho(P)]\!]_{\emptyset:\emptyset\to\beta(M')}$, i.e., $colour_{Sp'}(\rho(T)) = red$.

■

10.1.1 Model $\mathcal{T}$: Process refinement is not well-behaved

As the CSP trace refinement does not guarantee the preservation of behaviour, it is to be expected that the CSP-CASL notion of process refinement based on $\mathcal{T}$ fails to be well-behaved. This is illustrated by the following counter-example:

ccspec DOONEA =
  data
    sorts S
    op $a : S$;
  process
    P $= a \to STOP$
end

ccspec DONOTHING =
  data
    sort S
    op $a : S$;
  process
    $STOP$
end

As $STOP$ refines any process in the CSP traces model $\mathcal{T}$, we have:

$$\text{DOONEA} \overset{\text{proc}}{\rightsquigarrow}_{\mathcal{T}} \text{DONOTHING}$$

Let us consider the following test case:

$$T = a \to STOP$$

The colour of T with respect to DOONEA is green. However, T is coloured red over DONOTHING, i.e., $traces([\![T]\!]_\nu) \not\subseteq traces([\![\text{DONOTHING}]\!]_{\emptyset:\emptyset\to\beta(M)})$, for all models $M \in \mathbf{Mod}(D)$ and all variables evaluations $\nu : X \to M$, where D is the data specification of DONOTHING.

10.1.2 Models $\mathcal{F}$ and $\mathcal{N}$: Process refinement is well-behaved for divergence-free processes

Here, we show that CSP-CASL process refinement based on CSP models $\mathcal{F}$ and $\mathcal{N}$ is well-behaved, provided the processes involved are divergence-free.

THEOREM 10.1.2 *Let $Sp = (D, P)$ and $Sp' = (D, P')$ be two* CSP-CASL *specifications, with $Sp \overset{proc}{\rightsquigarrow}_{\mathcal{D}} Sp'$. Then, $\overset{proc}{\rightsquigarrow}_{\mathcal{F}}$ and $\overset{proc}{\rightsquigarrow}_{\mathcal{N}}$ are well-behaved provided the processes are divergence-free.*

PROOF. Concerning $\mathcal{F}$ and $\mathcal{N}$ it is sufficient to prove this for $\mathcal{F}$ only, as failures/divergences refinement and stable failures refinement are equivalent on divergence-free processes.

We need to show that the colour of a test case T over Sp remain unchanged over Sp' after a process refinement. Let $(D, P) \overset{proc}{\rightsquigarrow}_{\mathcal{F}} (D, P')$, we consider the cases of the *green* and *red* test cases.

Green Let the test process T be green with respect to Sp. Let M be a model and $\nu : X \to \beta(M)$ be a variable evaluation. We prove the two conditions:

1. We prove by induction on the length n of traces $t \in traces([\![T]\!]_\nu)$ that $t \in traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M)})$.

   For $n = 0$, this is obviously the case. Let $t = \langle t_1, \ldots, t_n, t_{n+1}\rangle \in traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. Then $\langle t_1, \ldots, t_n\rangle \in traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$ and thus by induction hypothesis also $\langle t_1, \ldots, t_n\rangle \in traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M)})$.

   Let us assume that $\langle t_1, \ldots, t_n, t_{n+1}\rangle \notin traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M)})$. As a divergence-free process, P' has the failure $(\langle t_1, \ldots, t_n\rangle, \emptyset)$. Thus, by healthiness of the stable failures domain[1], P' has also failure $(\langle t_1, \ldots, t_n\rangle, \{t_{n+1}\})$.
   This is a contradiction to $failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M)}) \subseteq failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$ and $(\langle t_1, \ldots, t_n\rangle, \{t_{n+1}\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.

2. We show that for all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_\nu)$ and for all $1 \leq i \leq n$ it holds that:
   $$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M)}).$$
   This trivially holds, as $failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M)}) \subseteq failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$, and by assumption we know that:
   $$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}).$$

Red Let the test process T be red with respect to Sp. Let M be a model, and ν be a variable evaluation. We prove the non-inclusion of the traces of T with respect to Sp'. We show that exists a trace $t \in traces([\![T]\!]_\nu)$ such that $t \notin traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M)})$. This follows directly, since we know that there exists $t \in traces([\![T]\!]_\nu)$ such that $t \notin traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M)})$ and $traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M)}) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.

[1] Specifically the healthiness condition **F3**, see Chpater2.

■

Besides the question whether refinements are well behaved with respect to test case colouring, one can also ask the other way round: *Is there a refinement between the specification and the test processes?* Here, we have the relation: Given a green test T over a CSP-CASL specification (D, P). Then $(D, P) \rightsquigarrow_{\mathcal{T}} (D, T)$, i.e. every green test process is a CSP-CASL process refinement with respect to the traces model $\mathcal{T}$.

LEMMA 10.1.3 *Let $Green_{(D,P)}$ be the set of all green test processes with respect to (D, P). Then $(D, P) \overset{proc}{\rightsquigarrow_{\mathcal{T}}} (D, \sqcap Green_{(D,P)})$.*

PROOF. We show that $\forall M \in \mathbf{Mod}(D) : [\![[\![P]\!]_M]\!]_{\mathcal{T}} \sqsubseteq_{\mathcal{T}} [\![[\![\sqcap Green]\!]_M]\!]_{\mathcal{T}}$.
As the $traces([\![\sqcap Green]\!]_M)$ is the union of all the green traces; all of which are subsets of $traces([\![P]\!]_M)$, we have $traces([\![\sqcap Green]\!]_M) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.
Thus, $(D, P) \overset{proc}{\rightsquigarrow_{\mathcal{T}}} (D, \sqcap Green_{(D,P)})$. ■

10.2 Testing and CSP-CASL enhancement

In Section 6.2 we have defined the notion of enhancement or horizontal development of CSP-CASL specifications. Here, we show that such enhancement relations allow the reuse of results established w.r.t. the original specification. That is, test cases preserve their colour after an enhancement step. Therefore, test cases which have been designed for basic features can be re-used whenever a more advanced product is conceived which includes these features.

THEOREM 10.2.1 *Let $Sp = (D, P)$ and $Sp' = (D', P')$ be CSP-CASL specifications with $Sp \gg Sp'$. Let T be a test process over Sp. Then, $colour_{Sp}(T) = colour_{Sp'}(T)$.*

PROOF. Let T be a test case over Sp. Let signature Σ and Σ' be the signature of the data part D and D' respectively. Let $\iota : \Sigma \to \Sigma'$ be the induced mapping from Σ to Σ'. We consider the cases of the *green*, *red* and *yellow* test cases.

For *green* test cases we show the following:

$$colour_{Sp}(T) = green \iff colour_{Sp'}(T) = green.$$

We prove both direction of the equivalence.

$\Longrightarrow$) Let $colour_{Sp}(T) = green$, i.e.,

1. For all $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to \beta(M)$ we have $traces([\![T]\!]_\nu) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.
2. For all $tr = \langle t_1, \ldots t_n \rangle \in traces([\![T]\!]_\nu)$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}).$$

We show that $colour_{Sp'}(T) = green$. i.e.,

1. For all $M' \in \mathbf{Mod}(D')$ and all variable evaluations $\nu' : X' \to \beta(M')$ it holds $traces([\![T]\!]_{\nu'}) \subseteq traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$.
2. For all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_{\nu'})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')}).$$

Let $M' \in \mathbf{Mod}(D')$. From the enhancement argument we know that there exists $M \in \mathbf{Mod}(D)$ such that $M = M'|_\iota$. Let $\alpha : \mathcal{A}lph(\beta(M'|_\iota)) \to \mathcal{A}lph(\beta(M'))$ be the injective alphabet translation. For the traces condition, we apply the alphabet translation on both sides:

$$\alpha_{\mathcal{T}}(traces([\![T]\!]_{\nu:X\to\beta(M'|_\iota)})) \subseteq \alpha_{\mathcal{T}}(traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)})).$$

This results in, $traces([\![T]\!]_{\nu':X\to\beta(M')}) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M')})$. From the enhancement argument we know that:

$$traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{T}}(traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})).$$

It follows that $traces([\![T]\!]_{\nu':X\to\beta(M')}) \subseteq traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$.

The same argument holds for the failures condition. Again, we apply the alphabet translation and obtain $failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M')})$. From the enhancement argument we know that:

$$failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{F}}(failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})).$$

It follows that for all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_{\nu':X\to\beta(M')})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')}).$$

Hence the traces and failures condition holds for $colour_{Sp'}(T) = green$.

$\Longleftarrow$) Let $colour_{Sp'}(T) = green$, i.e.,

1. For all $M' \in \mathbf{Mod}(D')$ and all variable evaluations $\nu' : X \to \beta(M')$ we have $traces([\![T]\!]_{\nu'}) \subseteq traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$.
2. For all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_{\nu'})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')}).$$

We show that $colour_{Sp}(T) = green$, i.e.,

1. For all $M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to \beta(M)$ we have $traces([\![T]\!]_{\nu}) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.
2. For all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_{\nu})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}).$$

Let $M \in \mathbf{Mod}(D)$. Again, from the enhancement argument we know that there exists $M' \in \mathbf{Mod}(D')$ such that $M = M' \mid_\iota$.

Then, for the traces condition we apply the reduct over the processes on both sides:

$$\hat{\alpha}_{\mathcal{T}}(traces([\![T]\!]_{\nu':X\to\beta(M')})) \subseteq \hat{\alpha}_{\mathcal{T}}(traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})).$$

From the enhancement argument we have

$$traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{T}}(traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)})).$$

It follows that

$$traces([\![T]\!]_{\nu:X\to\beta(M'|_\iota)}) \subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)}).$$

For the failures condition we directly obtain that for all $tr = \langle t_1, \ldots t_n\rangle \in traces([\![T]\!]_{\nu:X\to\beta(M'|_\iota)})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1}\rangle, \{t_i\}) \notin failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)}).$$

This follows by applying the reduct and the enhancement argument, i.e.,

$$failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)}) = \hat{\alpha}_{\mathcal{F}}(failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)})).$$

Thus, the traces and failures condition holds for $colour_{Sp}(T) = green$.

For *red* test cases, we show the following:

$$colour_{Sp}(T) = red \iff colour_{Sp'}(T) = red$$

We prove both direction of the equivalence.

$\Longrightarrow$**)** Let $colour_{Sp}(T) = red$, i.e., $\forall M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to M$ it holds $traces([\![T]\!]_\nu) \not\subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.

We show that $colour_{Sp'}(T) = red$, i.e., $\forall M' \in \mathbf{Mod}(D')$ and all variable evaluations $\nu' : X' \to M'$ it holds $traces([\![T]\!]_{\nu'}) \not\subseteq traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$.

Let $M' \in \mathbf{Mod}(D')$. From the enhancement argument we know that there exists $M \in \mathbf{Mod}(D)$ such that $M = M' \mid_\iota$.

We show that exists a trace $t' \in traces([\![T]\!]_{\nu':X\to\beta(M')})$ such that $t' \notin traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$.

We know that there exists a $t \in traces([\![T]\!]_{\nu:X\to\beta(M'|_\iota)})$ such that $t \notin traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)})$. Let $\alpha^*(t) = t'$, where $\alpha : \mathcal{A}lph(M' \mid_\iota) \to \mathcal{A}lph(M')$ is the injective alphabet translation. From the enhancement argument we have

$$traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{T}}(traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})).$$

Thus, it follows that $t' \in traces([\![T]\!]_{\nu':X\to\beta(M')})$ and $t' \notin traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$. Hence, $traces([\![T]\!]_{\nu':X\to\beta(M')}) \not\subseteq traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$, i.e., $colour_{Sp'}(T) = red$.

$\Longleftarrow$) Let $colour_{Sp'}(T) = red$. We show that $colour_{Sp}(T) = red$, i.e., $\forall M \in \mathbf{Mod}(D)$ and all variable evaluations $\nu : X \to M$ it holds $traces([\![T]\!]_\nu) \not\subseteq traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$.

Let $M \in \mathbf{Mod}(D)$. Again, from the enhancement argument we know that there exists $M' \in \mathbf{Mod}(D')$ such that $M = M' \mid_\iota$.

We show that exists a trace $t \in traces([\![T]\!]_\nu)$ such that $t \notin traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M)})$. This follows directly, since we know that there exists $t \in traces([\![T]\!]_{\nu'})$ such that $t \notin traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$ and from the enhancement argument we have that

$$traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{T}}(traces([\![P]\!]_{\emptyset:\emptyset\to\beta(M')})).$$

Hence, $traces([\![T]\!]_{\nu'}) \not\subseteq traces([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})$, i.e., $colour_{Sp}(T) = red$.

For *yellow* test cases, we show the following equivalence holds

$$colour_{Sp}(T) = yellow \Longleftrightarrow colour_{Sp'}(T) = yellow$$

In both direction of the equivalence, for the trace condition, the same proof argument as in the case of green test case holds. Here, we show how it goes for the failures condition.

Let $colour_{Sp}(T) = yellow$, i.e., for all $M \in \mathbf{Mod}(D)$ and variable evaluations $\nu : X \to M$ the following holds: for all $tr = \langle t_1, \ldots t_n \rangle \in traces([\![T]\!]_\nu)$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \in failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M)}).$$

We show that $colour_{Sp'}(T) = yellow$ i.e., for all $M' \in \mathbf{Mod}(D')$ and variable evaluations $\nu' : X' \to M'$ the following holds: for all $tr = \langle t_1, \ldots t_n \rangle \in traces([\![T]\!]_{\nu'})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \in failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')}).$$

Let $M' \in \mathbf{Mod}(D')$. From the enhancement argument we know that there exists $M \in \mathbf{Mod}(D)$ such that $M = M'|_\iota$. Let $\alpha : \mathcal{A}lph(\beta(M'|_\iota)) \to \mathcal{A}lph(\beta(M'))$ be the injective alphabet translation. We apply the alphabet translation on the failures: $\alpha_{\mathcal{F}}(failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M'|_\iota)}))$. This results in, $failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M')})$. From the enhancement argument we know that:

$$failures([\![P]\!]_{\emptyset:\emptyset\to\beta(M')}) = \hat{\alpha}_{\mathcal{F}}(failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')})).$$

Then, we directly obtain, that for all $tr = \langle t_1, \ldots t_n \rangle \in traces([\![T]\!]_{\nu':X\to\beta(M')})$ and for all $1 \leq i \leq n$ it holds that:

$$(\langle t_1, \ldots, t_{i-1} \rangle, \{t_i\}) \in failures([\![P']\!]_{\emptyset:\emptyset\to\beta(M')}).$$

Hence, $colour_{Sp'}(T) = yellow$.

■

Summarizing, the enhancement notion developed for CSP-CASL allows us to re-use test cases developed for basic specification, to experiment the same behavior in enhanced specification. That is, *green*, *red* and *yellow* test cases remain unchanged after an enhancement step.

10.3 Case study: remote control unit

In this section we develop an example from the embedded systems domain: an infrared remote control unit used for home appliances such as TV, VCR, DVD player etc. Here we give abstract and concrete specifications and show the various CSP-CASL specifications at different levels of abstraction. We show how the refinement and enhancement of the various CSP-CASL specifications influence the testing procedure of such device.

10.3.1 Specification of a remote control unit

Figure 10.1: Basic Remote Control Unit (BRCU)

On an abstract level, a remote control unit (*RCU*) can be described as follows: there are a number of *buttons* which can be pressed (one at a time), and a light emitting diode (*LED*) which is capable of sending *signals* (bitvectors of a certain length). The *RCU* has an internal table which signal correspond to which button. Whenever a button is pressed, it sends a corresponding signal via the LED. Such an Abstract Remote Control Unit can be specified in CSP-CASL as follows:

ccspec ABSRCU =
data
  sort $Button, Signal$
  op $codeOf : Button \to Signal;$
process
  AbsRCU $= ?x : Button \to codeOf(x) \to$ AbsRCU

end

Basic remote control units (BRCU) as they were produced in the 1970's had e.g., 12 buttons

(i.e., $b_0 \ldots b_9, b_{OnOff}, b_{Mute}$), where the corresponding signals are 16-bit key-codes; for example:

$$0000.01010.0000001 \text{ is a signal for } b_1$$

There are various standards for remote controls; one of these defines that the first 4 bits identify the company ID, the next 5 bits represent the device ID (i.e., TV, DVD, etc.), while the last 7 bits identify which button was pressed. This can be specified in CSP-CASL as follows:

ccspec BRCU =
data
  sort $Button, Signal$
  ops $b_0, b_1, \ldots, b_9, b_{OnOff}, b_{Mute} : Button;$
  free type $Bit ::= 0 \mid 1$
  then LIST[**sort** Bit]
    then
      sort $Signal = \{l : List[Bit] \bullet \#l = 16\}$
      op $codeOf : Button \rightarrow Signal;$
        $prefix : List[Bit] = [0000] ++ [01010]$
      axioms
        $codeOf(b_0) = prefix ++ [0000000];$
        $\ldots$
        $codeOf(b_9) = prefix ++ [0001001];$
        $codeOf(b_{Mute}) = prefix ++ [0001111];$
        $codeOf(b_{OnOff}) = prefix ++ [1111111];$
        $\forall b : Button \bullet \exists l : List[Bit] \bullet codeOf(b) = prefix ++ l$
process
  BRCU $= ?x : Button \rightarrow codeOf(x) \rightarrow$ BRCU

end

Soon after the first generation, the market demanded more comfortable devices with more functionality and, thus, more buttons. Modern remote control units have about 50-200 buttons. For the example, we assume that in the Extended specification the buttons b_{volup} and b_{voldn} for controlling the volume and b_{chup} and b_{chdn} for zapping though channels were added, with appropriate key-codes. In CSP-CASL, such an extension can be specified by defining a sort *EButton* which is an extension (superset) of sort *Button*. Of course, in the extended specification, the domain of operation *codeOf* must be suitably extended. Here is the abstract version of an extended remote control unit:

ccspec ABSERCU =
data
  sorts $Button < EButton;\ Signal$
  ops $codeOf : Button \rightarrow Signal;$
    $codeOf : EButton \rightarrow Signal$

process
AbsERCU $= ?x : EButton \rightarrow codeOf(x) \rightarrow$ AbsERCU

end

For a concrete implementation of the abstract extended specification, we use the supersorting and overloading features built into CASL. To this end, we import the data part of BRCU, named BRCUDATA, and define a supertype *EButton* of *Button*, which includes four new buttons. The function $codeOf : EButton \rightarrow Signal$ is in overloading relation with the function $codeOf : Button \rightarrow Signal$. Therefore, the CASL semantics ensure that both functions yield the same result for elements of type *Button*.

ccspec ERCU =
data BRCUDATA **then**
free type $EButton ::= \mathbf{sort}\ Button \mid b_{volup} \mid b_{voldn} \mid b_{chup} \mid b_{chdn}$
op $codeOf : EButton \rightarrow Signal$
axioms
$codeOf(b_{volup}) = prefix ++ [0010000];$
$codeOf(b_{voldn}) = prefix ++ [0100000];$
$codeOf(b_{chup}) = prefix ++ [1000000];$
$codeOf(b_{chdn}) = prefix ++ [1100000];$
process
ERCU $= ?x : EButton \rightarrow codeOf(x) \rightarrow$ ERCU

end

If more and more functions are added to a device, buttons need to be reused. That is, some buttons have a main and alternate inscription and there is a special button b_{alt}; if this button is pressed the key-code of the subsequently pressed button changes according to the alternate inscription. Basically, the button b_{alt} serves as a modifier of the next button. This enhancement differs from the previous one, since it requires the device to distinguish between two states (whether the b_{alt} modifier button has been pressed or has not been pressed), and it enforces a modification in the process part of the specification. The enhancement from BRCU to ERCU makes use of overloading and added supersorts. The following is an (abstract) specification of a *RCU* with Modifier enhancing the extended *RCU*. To this end, we import the data part of ERCU, named ERCUDATA.

ccspec MERCU =
data ERCUDATA **then**
free type $MButton ::= \mathbf{sort}\ EButton \mid b_{alt}$
sort $AltButton = \{x : MButton \bullet x = b_{alt}\}$
op $codeOfAlt : EButton \rightarrow Signal$
process
MERCU $= ?x : EButton \rightarrow codeOf(x) \rightarrow$ MERCU
$\Box\ b_{Alt} \rightarrow ?x : EButton \rightarrow codeOfAlt(x) \rightarrow$ MERCU
end

As the *codeOfAlt* is not in overloading relation with *codeOf*, after pressing the button b_{Alt}

the remote control can send out different signals for the buttons pressed. The specification MERCU is abstract in so-far, as the functionality of *codeOfAlt* is not further specified. In order to demonstrate the integration of features in a software product line development, we show how to reuse specification modules. A universal remote control is a device which can be used for *TV*, *VCR*, and *DVD* players. For this purpose, it has a button *mode*, which allows the user to cycle through the three operation modes (*TV*, *VCR*, *DVD*). The specification URCU builds onto the specification ERCU, as well as on similar built specifications ERCUDDVD and ERCUVCR.

ccspec URCU =
data { ERCUDATA **and** ERCUDATADVD **and** ERCUDATAVCR }
then **sort** *NewButton*
op *mode* : *NewButton*
process
let TV $= ?x : Button \rightarrow codeOf(x) \rightarrow$ TV
$\Box\ mode \rightarrow$ DVD
DVD $= ?x : Button \rightarrow codeOfDVD(x) \rightarrow$ DVD
$\Box\ mode \rightarrow$ VCR
VCR $= ?x : Button \rightarrow codeOfVCR(x) \rightarrow$ VCR
$\Box\ mode \rightarrow$ TV
in TV
end

10.3.2 Refinement and enhancement of the remote control unit

Here, we formally relate all the specifications described in the previous section. Figure 10.2 summarizes the development of the remote control unit specification in CSP-CASL.

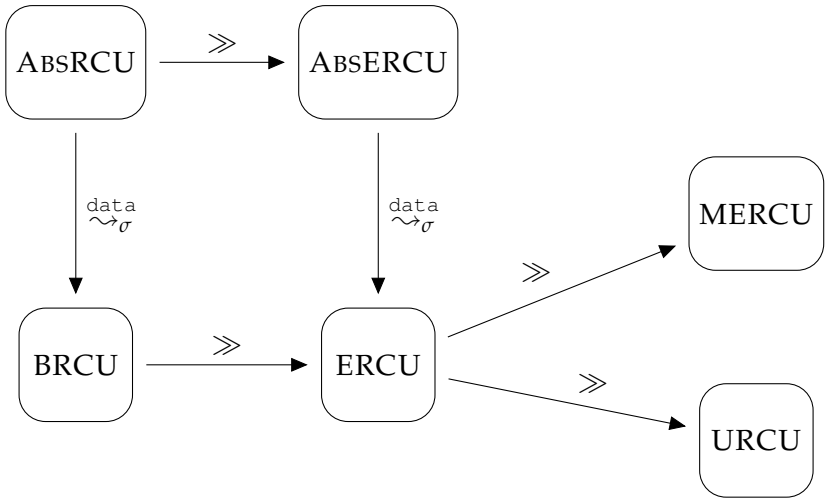

Figure 10.2: Remote control unit in CSP-CASL specifications development

In the following lemmas we prove each of the development notion (*refinement* or *enhancement*) illustrated in Figure 10.2.

LEMMA 10.3.1 *The following holds:* ABSRCU $\overset{data}{\rightsquigarrow_\sigma}$ BRCU *and* ABSERCU $\overset{data}{\rightsquigarrow_\sigma}$ ERCU.

PROOF. ABSRCU $\overset{data}{\rightsquigarrow_\sigma}$ BRCU holds if $\mathbf{Mod}(D_{BRCU})|_\sigma \subseteq \mathbf{Mod}(D_{AbsRCU})$, where $\sigma : \Sigma(D_{AbsRCU}) \to \Sigma(D_{BRCU})$ is the signature morphism. This trivially holds as every model of D_{BRCU} is a model of D_{ABSRCU}. The same arguments holds for ABSERCU $\overset{data}{\rightsquigarrow_\sigma}$ ERCU. Both data refinements are simply proved using HETS. ∎

LEMMA 10.3.2 *The following holds:* ABSRCU $\gg$ ABSERCU.

PROOF. The process *AbsERCU* communicates in a richer data structure. Here, we observe that all models of the data part D_{AbsRCU} of ABSRCU can be extended to models of the data part $D_{AbsERCU}$ of ABSERCU:

$$\mathbf{Mod}(D_{AbsRCU}) = \mathbf{Mod}(D_{AbsERCU})|_\iota$$

where $\iota : \Sigma(D_{AbsRCU}) \to \Sigma(D_{AbcERCU})$ is the induced map from the signature of D_{AbsRCU} to $D_{AbcERCU}$. Here, the signature of D_{AbsRCU} is embedded into the signature $D_{AbcERCU}$, i.e.,

$$\begin{gathered}\Sigma(D_{AbsRCU}) = (\{Signal, Button\}, \{codeOf_{Button \times Signal}\}, \emptyset, \emptyset, \emptyset) \\ \subseteq \\ \Sigma(D_{AbsERCU}) = (\{Signal, Button, EButton\}, \{codeOf_{Button \times Signal}, codeOf_{EButton \times Signal}\}, \\ \emptyset, \emptyset, \{<_{Button,EButton}\})\end{gathered}$$

Here, we use the *supersort enhancement pattern* (Theorem 7.2.3) to show that ABSRCU$\gg$ABSERCU. To this end we define a mapping $\xi : S \to S'$ from the set of sorts of $\Sigma(D_{AR})$ to the set of sort of $\Sigma(D_{ER})$, which is simply the identity with the exception:

$$\xi(Button) = EButton$$

As we define *codeOf* only for the new values, we have a conservative model extension. Obviously, ξ maps the process of ABSRCU to the process ABSERCU, i.e., we map the process name $\xi(\text{AbsRCU}) = \text{AbsERCU}$, and we obtain:

$$\begin{aligned}\xi(\text{AbsRCU}) &= ?x : \xi(Button) \to \xi(codeOf(x)) \to \xi(\text{AbsRCU}) \\ &= ?x : EButton \to codeOf(x) \to \text{AbsERCU}\end{aligned}$$

This proves the three conditions for *supersort enhancement pattern*. Therefore, we have that ABSRCU $\gg$ ABSERCU. ∎

LEMMA 10.3.3 *The following holds:* BRCU $\gg$ ERCU.

PROOF. Here again we use the *supersort enhancement pattern* to prove that BRCU$\gg$ERCU: To this end we define the map ξ to be the identity on all sorts with the exception of $\xi(Button) = EButton$. Clearly, the signatures are embedded with ξ. As we define *codeOf* only for the new values, we have a conservative model extension. Obviously, ξ maps the process of BRCU to the process of ERCU. Thus, all three conditions are true and therefore BRCU$\gg$ERCU. ∎

LEMMA 10.3.4 *The following holds:* ERCU $\gg$ MERCU.

PROOF. Here, we use the *external choice enhancement patter* (Theorem 7.2.1) to prove that ERCU$\gg$MERCU. First, we have to adjust the process part of MERCU to the syntactic pattern stated in the theorem. To this end, we use the following law:

$$(*) \quad a \rightarrow R = ?x : \{a\} \rightarrow R$$

This allows us to make the following transformation:

$$\begin{array}{lcl} \text{MERCU} & = & ?x : \mathit{EButton} \rightarrow \mathit{codeOf}(x) \rightarrow \text{MERCU} \\ & & \Box\ b_{\mathit{Alt}} \rightarrow ?y : \mathit{EButton} \rightarrow \mathit{codeOfAlt}(y) \rightarrow \mathit{MERCU} \\ & \Downarrow & \\ \text{MERCU} & = & ?x : \mathit{EButton} \rightarrow \mathit{codeOf}(x) \rightarrow \text{MERCU} \\ & & \Box ?z : \mathit{AltButton} \rightarrow ?y : \mathit{EButton} \rightarrow \mathit{codeOfAlt}(y) \rightarrow \mathit{MERCU}. \end{array}$$

Concerning the data part, MERCU is a conservative extension of ERCU, as all added symbols are new, and, if they relate to old ones, they follow a definitional extension pattern. Thanks to the CASL free type b_{alt} is different from all values of *EButton*. Thus, both conditions of the *external choice enhancement pattern* hold, hence ERCU$\gg$MERCU. ■

LEMMA 10.3.5 *The following holds:* ERCU$\gg$URCU.

PROOF. Again, we use *external choice enhancement pattern* to establish ERCU$\gg$URCU. Using the rule $(*)$, we adjust the process part of URCU, and we obtain the following process:

$$\begin{array}{lcl} \textbf{let}\ \text{TV} & = & ?x : \mathit{Button} \rightarrow \mathit{codeOf}(x) \rightarrow \text{TV} \\ & & \Box ?y : \mathit{ModeButton} \rightarrow \text{DVD} \\ \text{DVD} & = & ?x : \mathit{Button} \rightarrow \mathit{codeOfDVD}(x) \rightarrow \text{DVD} \\ & & \Box ?y : \mathit{ModeButton} \rightarrow \text{VCR} \\ \text{VCR} & = & ?x : \mathit{Button} \rightarrow \mathit{codeOfVCR}(x) \rightarrow \text{VCR} \\ & & \Box ?y : \mathit{ModeButton} \rightarrow \text{TV} \\ \textbf{in}\ \text{TV} & & \end{array}$$

On the data part, we have that $\Sigma(D_{ERCU}) \subseteq \Sigma(D_{URCU})$. The added symbol in D_{URCU} don't interfere with the old symbols, hence $\mathbf{Mod}(D_{ERCU}) = \mathbf{Mod}(D_{URCU})\ |_{\iota}$, where ι is the induced mapping. ■

10.3.3 Testing the remote control unit

In this section we design some test cases for the *RCU* specifications and show the re-use of test cases as well as the preservation of colours described in the previous section. The first set of test cases is designed to test ABSRCU:

$$A_0 : u : Button \rightarrow codeOf(u) \rightarrow STOP$$
$$A_1 : u : Button \rightarrow v : Signal \rightarrow STOP$$
$$A_3 : u : Button \rightarrow w : Button \rightarrow STOP$$

Here, u, v and w are variable over the indicated sorts.

Thanks to the refinement and the enhancement results summarized in Figure 10.2, test cases T over AbsRCU are also test cases over all the other specifications. With respect to their colouring we obtain e.g., the following inheritance relations:

- $colour_{\text{AbsERCU}}(T) = colour_{\text{AbsRCU}}(T)$ thanks to AbsRCU $\gg$ AbsERCU.
- $colour_{\text{BRCU}}(T) = colour_{\text{AbsRCU}}(T)$ thanks to AbsRCU $\overset{\texttt{data}}{\rightsquigarrow}$ BRCU.
- $colour_{\text{ERCU}}(T) = colour_{\text{AbsRCU}}(T)$, where we can either use the connection over BRCU or over AbsERCU.

This means for our three test cases A_0, A_1 and A_2 that their colour is the same in all specification mentioned in Figure 10.2, where their colouring can be determined by looking at AbsRCU only, i.e., the specification with the smallest number of axioms. For the colouring we obtain the following result respect to AbsRCU:

	A_0	A_1	A_2
AbsRCU	*Green*	*Yellow*	*Red*

A next set of test cases is designed to test BRCU:

$$T_0 : STOP$$
$$T_1 : b_1 \rightarrow STOP$$
$$T_2 : b_1 \rightarrow codeOf(b_1) \rightarrow b_6 \rightarrow codeOf(b_6) \rightarrow STOP$$
$$T_3 : b_1 \rightarrow b_6 \rightarrow STOP$$
$$T_4 : b_0 \rightarrow (prefix +\!+ [0000101]) \rightarrow STOP$$

The following table shows how these test process are coloured with respect to BRCU.

	T_0	T_1	T_2	T_3	T_4
BRCU	Green	Green	Green	Red	Red

The empty observation T_0 is green with respect to all specifications (see Proposition 9.2.2). T_1 is green for *BRCU* as *BRCU* cannot refuse the event b_1 after the empty trace. The same holds for T_2, since *BRCU* cannot refuse the signal of b_1 after the event of b_1. T_3 consists of a sequence of two button presses and therefore is red for *BRCU*. T_4 however is red for *BRCU* due to a wrong signal event, i.e., $codeOf(b_0) \neq codeOf(b_5)$. Similarly to the result above, these test cases preserve these colours w.r.t. ERCU, MERCU and URCU.

In order to test the new features available in a the product line, new test cases have to be designed which use the new symbols. E.g., for ERCU the following test cases do this:

$$T_5 : b_1 \rightarrow codeOf(b_1) \rightarrow b_{VolUp} \rightarrow codeOf(b_{VolUp}) \rightarrow STOP$$
$$T_6 : b_{ChUp} \rightarrow codeOf(b_{ChUp}) \rightarrow STOP$$
$$T_7 : b_{ChDn} \rightarrow codeOf(b_{ChDn}) \rightarrow b_1 \rightarrow STOP$$
$$T_8 : b_{ChUp} \rightarrow b_{VolDn} \rightarrow STOP$$
$$T_9 : b_{ChUp} \rightarrow codeOf(b_{VolUp}) \rightarrow STOP$$

These test process are coloured with respect to ERCU in the following way:

	T_5	T_6	T_7	T_8	T_9
ERCU	*Green*	*Green*	*Green*	*Red*	*Red*

These test cases preserve these colours w.r.t. MERCU and URCU.

As BRCU $\gg$ ERCU, the colour of the test cases $T_0 \ldots T_4$ over ERCU is inherited from their colour w.r.t. BRCU. As ERCU$\gg$MERCU, the colour of the test cases $T_0 \ldots T_9$ over ERCU is inherited from their colour w.r.t. ERCU. However, the testing of the new features of MERCU requires new test cases, for instance:

$$T_{10} : b_1 \rightarrow codeOf(b_1) \rightarrow b_{Alt} \rightarrow b_2 \rightarrow codeOfAlt(b_2) \rightarrow STOP$$
$$T_{11} : b_{Alt} \rightarrow b_1 \rightarrow codeOfAlt(b_1) \rightarrow STOP$$
$$T_{12} : b_2 \rightarrow codeOfAlt(b_2) \rightarrow STOP$$
$$T_{13} : b_{Alt} \rightarrow b_{VolDn} \rightarrow codeOf(b_{VolDn}) \rightarrow STOP$$

The following table shows how these test process are coloured with respect to MERCU:

	T_{10}	T_{11}	T_{12}	T_{13}
MERCU	*Green*	*Green*	*Red*	*Red*

Again, we design test cases to experiment the new features specified in URCU:

$$\begin{aligned} T_{14} : b_1 &\rightarrow codeOf(b_1) \rightarrow mode \rightarrow b_2 \rightarrow codeOfDVD(b_2) \\ &\rightarrow mode \rightarrow b_5 \rightarrow codeOfVCR(b_5) \\ &\rightarrow mode \rightarrow b_7 \rightarrow codeOf(b_7) \rightarrow STOP \end{aligned}$$
$$T_{15} : mode \rightarrow b_1 \rightarrow codeOfDVD(b_1) \rightarrow STOP$$
$$T_{16} : b_2 \rightarrow codeOfVCR(b_2) \rightarrow STOP$$
$$T_{17} : mode \rightarrow b_{VolDn} \rightarrow codeOf(b_{VolDn}) \rightarrow STOP$$

And their colour:

	T_{14}	T_{15}	T_{16}	T_{17}
URCU	*Green*	*Green*	*Red*	*Red*

10.3.3.1 Test case evaluation and execution

In this section we consider the evaluation of test cases w.r.t. CSP-CASL specifications from an implementation point of view. We also demonstrate how we run test cases on an implementation of the remote control. Our testing framework essentially consist of two parts which all have tool support:

1. We use CSP-CASL-prover [OIR09] to verify the colour of a test case. To this end we use the syntactic characterization of the test colouring as defined in Section 9.3. We also verify that a test case T is executable for the chosen PCO, see Section 9.4 for the definition.

2. Given a coloured test case and a particular SUT, our Test Execution and test Verdict program (TEV), automatically runs a test against the SUT and automatically determines the test verdict.

In terms of executing tests against the SUT, we have implemented in Java some of the remote control specifications discussed in the previous sections. In the following we illustrate the essential parts of the Java implementation for the BRCU.

```
public class BasicRemoteControl extends JFrame
                                implements ActionListener{

  public BasicRemoteControl() {  // Constructor
     ...
     jlbOutput = new JTextField(12);
     jlbOutput.setEditable(false);
     ...
     jbnButtons = new JButton[13];

     for (int i=0; i<=9; i++){ // Create numeric Jbuttons
          jbnButtons[i] = new JButton(String.valueOf(i));
      }}

  public void actionPerformed(ActionEvent e){ //Perform action
      for (int i=0; i<jbnButtons.length; i++){
          if(e.getSource() == jbnButtons[i]){
              switch(i) {
                   case 0:
                    codeOf("0000010100000000" )
                          break;
                      case 1:
                    codeOf("0000010100000001" );
                     break;
     ....}}}}

   void codeOf(String s){   // Set the signal
        jlbOutput.setText(s);
   }

   public static void main(String args[]) {   // Main method
         BasicRemoteControl brc = new BasicRemoteControl(); ...}}
```

Let $\mathcal{P}_{BRCU} = (\mathcal{A}_{BRCU}, \|...\|, \mathcal{D})$, be the PCO of BRCU with respect to the Java program *BasicRemoteControl.java*, in Listing 2.1. Here, the alphabet $\mathcal{A}_{BRCU}$ of primitive events which can be communicated at this point are the various AWT components of the buttons and the text field where the signal is shown, e.g.,

$$\mathcal{A}_{BRCU} = \{jplOutput, jbnButtons[0], jpnButtons[1], \ldots, jpnButtons[9]\}.$$

We establish the following mapping:

$$\|jplOutput\| = Signal, \|jbnButtons[0]\| = b_0, \ldots \|jbnButtons[9]\| = b_9$$

In order to make the connection between the SUT and the testing system we use `abbot` [abb], which is a Java package that enables to test Java AWT components. Here we establish also the direction of testing $\mathcal{D}$.

In order to make such connection, we develop a new program called *TeVBRCU* which resides in the package of the BRCU Java implementation (*BasicRemoteControl.java*). *TeVBRCU* is able to make the connection with the SUT, in this case *BasicRemoteControl* and to access the different objects to be tested. Moreover, in *TeVBRCU* we program the decision procedure to determine the verdict of a test case (see Section 9.4).

In the initial part of the *TeVBRCU* class, we declare private objects which are going to be tested.

```
public class TeVBRCU extends ComponentTestFixture{

  private BasicRemoteControl brc;
  private JTextField display;
  private JButton button0, button1,button2,button3,button4, button5,
                  button6, button7, button8, button9
  private JButtonTester bt0, bt1, bt2, bt3, bt4, bt5, bt6, bt7, bt8, bt9
  private JTextFieldTester d;
  boolean hasEventOccured = false;

  private String Colour = ""; // colour of test case
  private Timer timer;       // length of timeout

    public TeVBRCU(String name) {
        super(name);
    }
 ...
 /* Set up the SUT and Run test cases*/
 ...
}
```

Here, *ComponentTestFixture* is an abstract class which extends the *TestCase* class of *JUnit*[2]. *ComponentTestFixture* ensures proper setup and cleanup for a GUI environment, it provides methods for automatically placing a GUI component within a frame and properly handling window showing/hiding operations. In this initial part we declare two type of

[2] http://www.junit.org/

objects: objects that refers to the actual SUT, e.g., *display*, *button*0 etc; and objects that will be used to stimulate the SUT, e.g., *bt*0, *bt*1, etc. Here, we declare also the colour of a test case as a string.

In a second part of the program, we setup a method that initialize the system under test, in our case the Java program *BasicRemoteControl*. Here, we make sure that the SUT is visible:

```
protected void setUp() throws Exception {
  brc = new BasicRemoteControl();
  brc.setSize(241, 217);
  brc.pack();
  brc.setLocation(400, 250);
  brc.setVisible(true);
  showWindow(brc, null, false);
  ...
}
```

The `abbott` package allows us to find automatically the components of the system to be tested. In the case of the remote control unit it finds the different buttons. For instance, the following code finds *button* 1, and binds it to the private object *button*1:

```
button1 = (JButton)getFinder().
          find(new ClassMatcher(JButton.class) {
  public boolean matches(Component c) {
    return c instanceof JButton &&
           ((JButton)c).getText().equals("1");}});
```

The same procedure we adopt to find the text field where the signal is displayed:

```
display = (JTextField) getFinder().
          find(new ClassMatcher(JTextField.class));
display.addCaretListener(new CaretListener(){
 public void caretUpdate(CaretEvent caretEvent){
     hasEventOccured = true;
  }
 });
```

In order to execute test cases we instantiate robot-like objects, which automatically stimulate the SUT.

```
bt0 = new JButtonTester();   ....   bt9 = new JButtonTester();
d1 = new JTextFieldTester();
```

Here, the objects *bt*0 of *JButtonTester*() provides action methods and assertion for objects declared as *JButton*.

Our decision procedure for determining the verdict of a test case, makes use of the timeout in order to make sure certain events can really happen. In our program we make use of the Java class *Timer* (**import java.util.Timer**;) in order to set a timeout and make sure that events are performed within this timeout.

For the execution of a test, we set a timeout of 1 secs as the period of time in which a signal is expected from the *RCU*. Depending on the colour of the test case and the response from the SUT, TEV determines automatically the test verdict. Here, we have encoded the verdict algorithm within each test case.

The decision procedure for determining the verdict of a test case is encoded with in each test case. In our a program, a test case is a unit test, e.g, *testFirstCase*(). In the following test units we encode some test cases developed for BRCU.

- $T_1 = b_1 \rightarrow STOP$

```
public void testFirstCase() throws Exception {
  colour = "Green";
  System.out.println("Start time of Test Case: " + getTime());

  bt1.actionClick(button1);  // click button 1 -- ts2sut
  System.out.println("Button 1 pressed at: " + getTime());

    if(hasEventOccured){
      System.out.println("Has Event Occured: " + hasEventOccured);
      System.out.println("Signal" + display.getText()
                         + "showed at:" + getTime());
      System.out.println("Test Result: FAIL");
  } else {
     System.out.println("Test Result: PASS");
   }
     hasEventOccured = false;
}
```

- $T_2 = b_1 \rightarrow codeOf(b_1) \rightarrow b_6 \rightarrow codeOf(b_6) \rightarrow STOP$

```
public void testSecondCase() throws Exception {
  colour = "Green";
  System.out.println("Start time of Test Case: " + getTime());

  bt1.actionClick(button1);  // click button 0 -- ts2sut
  System.out.println("Button 0 pressed at: " + getTime());

 if(hasEventOccured
    && display.getText().equals("0000010100000001")){
    System.out.println("Has Event Occured: " + hasEventOccured);
    System.out.println("Signal" + display.getText()
                       + "showed at:" + getTime());
     // proceed to test button 6
     testButton6();
    } else {
      System.out.println("Test Result: FAIL");
   }
     hasEventOccured = false;
}

public void testButton6() throws Exception {
  bt6.actionClick(button6);  // click button 6 -- ts2sut
  System.out.println("Button 6 pressed at: " + getTime());
```

```
  if(hasEventOccured
   && display.getText().equals("0000010100000110")){
      System.out.println("Has Event Occured: " + hasEventOccured);
      System.out.println("Signal" + display.getText()
                         + "showed at:" + getTime());
      System.out.println("Test Result: PASS");
   } else {
      System.out.println("Test Result: FAIL");
   }
     hasEventOccured = false;
}
```

10.4 Summary

In this chapter, we have demonstrated the connection between the testing framework presented in Chapter 9 and the CSP-CASL development notion presented in Chapter 6. The major innovations are:

Testing and vertical development In our approach, we can build test suites for any level of abstraction in this process. In particular, test cases can be constructed already in the very beginning, as soon as the first loose specifications are written. Our approach ensures that test cases which are designed at an early stage can be used without modification for the test of a later development stage. Ergo, test suites can be developed in parallel with the SUT, which reduces the overall development time and helps to avoid ambiguities and specification errors.

Testing and horizontal development We have proved that our notion of enhancement (or horizontal development) for CSP-CASL allows the preservation of expected result of test cases. Therefore, this notions allow to reuse test cases throughout a product line. In particular, test cases preserves their colour.

We have illustrated the overall framework of testing and CSP-CASL development with the example of a remote control units for home appliances.

PART IV

Industrial applications

(CHAPTER ...11)

The electronic payment system EP2

Contents

In this chapter we demonstrate that the so far presented theoretical results are applicable in an industrial setting. Namely, we apply our technique to an electronic payment system called EP2. In the next section, we will introduce EP2 and discuss the specification structure and style of EP2. We demonstrate how we capture in a faithful way such specification in CSP-CASL. We prove the formal refinement of the different levels of abstraction. We analyze deadlock and livelock freedom, and finally we show a testing framework for an EP2 payment system.

A first modelling approach in CSP-CASL of the EP2 specification has been presented in [GRS05]; while in [KR09] a refinement verification as well as deadlock and livelock analysis of EP2 has been carried out.

11.1 Introducing the EP2 payment system

The EP2 system is an electronic payment system and it stands for 'EFT/POS 2000', short for 'Electronic Fund Transfer/Point Of Service 2000', it is a joint project established by a number of (mainly Swiss) financial institutes and companies in order to define EFT/POS infrastructure for credit, debit, and electronic purse terminals in Switzerland[1]. EP2 builds on a number of other standards, most notably EMV 2000 (the Europay/Mastercard/Visa Integrated Circuit Card standard[2]) and various ISO standards. The EP2 project began in

[1] www.eftpos2000.ch
[2] www.emvco.com

October 2000, and was officially completed on May 31 2003. The latest version (*ep2_spec V5.0.0*) of the EP2 standard was released on October 2008.

The EP2 system, as illustrated in Figure 11.1, consists of seven autonomous entities:

Acquirer: A system for processing electronic payment transactions.

Card: Payment utility opened by a specific cardholder.

Point Of Service (POS): A system where a cardholder may purchase goods and/or services.

POS Management System (PMS): System that allows the merchant to administrate his terminal population.

Service Center: A system component used for configuration and maintenance of a terminal.

Terminal: A system used for processing transactions.

These components are centered around an EP2 *Terminal*. The different entities communicate with the *Terminal* and, to a certain extent, with one another via XML-messages in a fixed format over TCP/IP. The messages contain information about authorisation, financial transactions, as well as initialisation and status data. The state of each component heavily depends on the content of the exchanged data. Each component is a reactive system defined by a number of use cases. Thus, there are both reactive parts and data parts which need to be modelled. Both these parts are heavily intertwined.

11.1.1 EP2 document structure and specification style

The EP2 specification consists of twelve documents, each of which describe the different components or some aspect common to the components. Figure 11.2 illustrates a general overview of the EP2 document structure. Different books are concerned with a particular aspect of the system.

- EP2 *system book*: Contains general information on EP2 as well as chapters which are common interest in all EP2 specification documents.
- EP2 *component book*: Contains information about the functional and non-functional requirements of each component, i.e., *Acquirer*, *Service Center*, etc.
- EP2 *interface book*: Contains the specification of the communication interfaces of the EP2 system, i.e., blue lines in Figure 11.1.
- EP2 *data dictionary book*: Contains a detailed description of the various XML message formats exchanged between the different EP2 components.
- EP2 *security book*: Contains information regarding the the EP2 security mechanism. It contains the format and management of the various public key, i.e., as well as

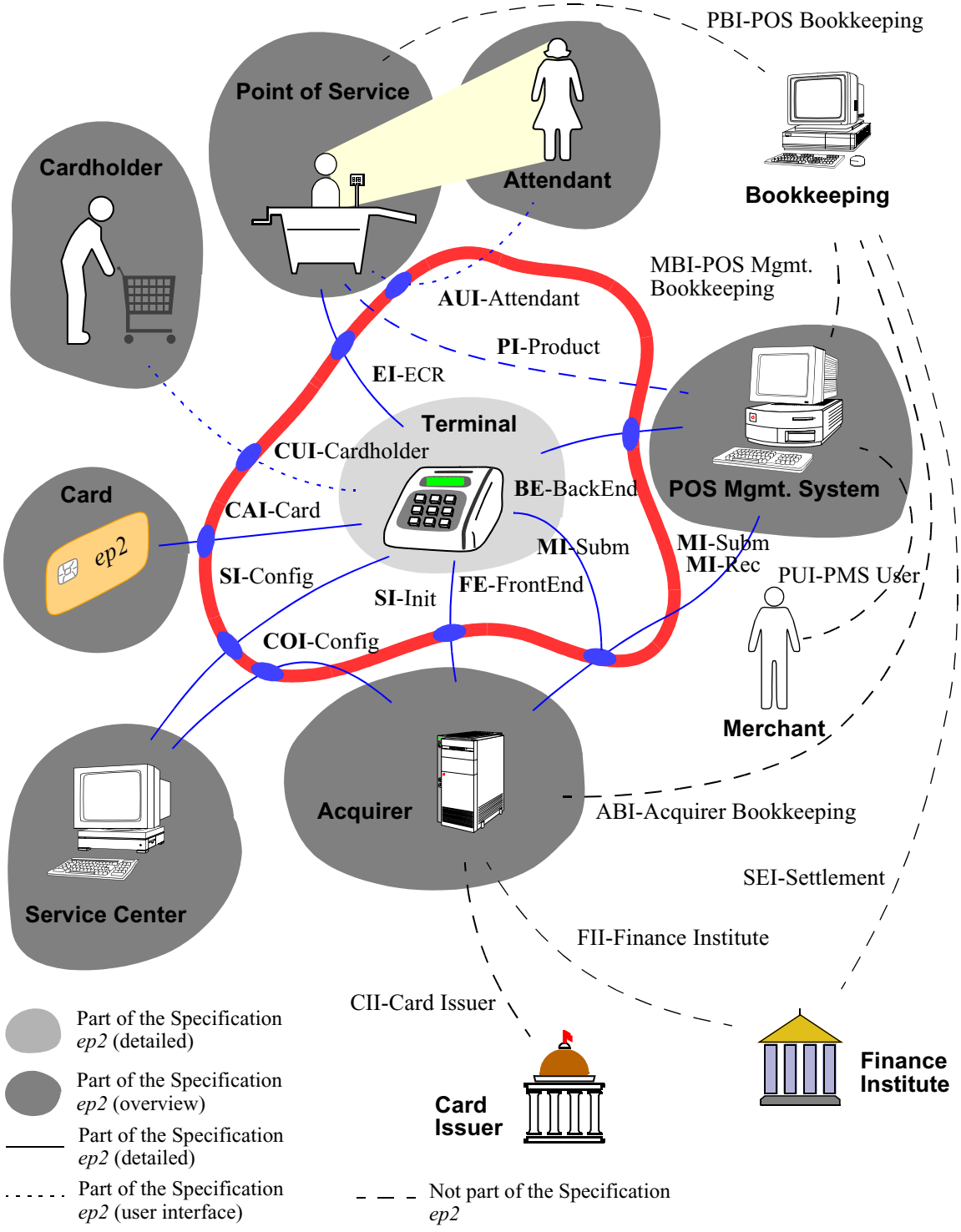

Figure 11.1: The EP2 system [Con08].

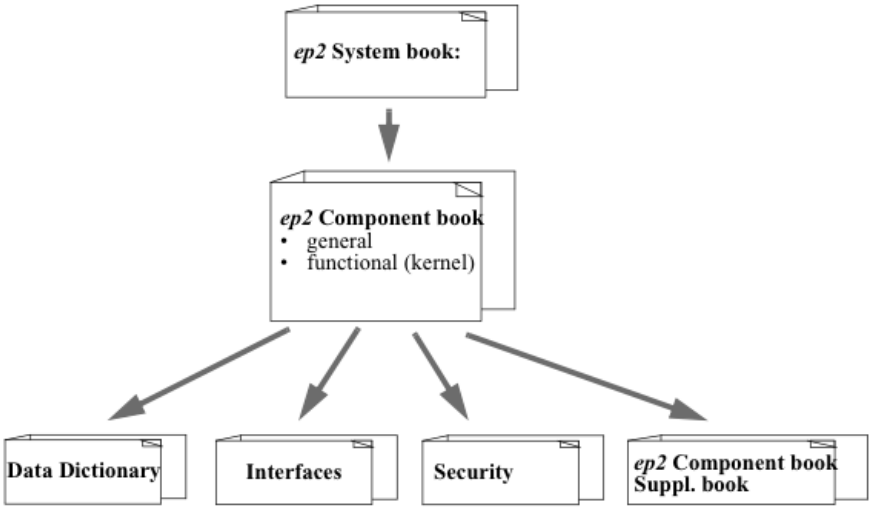

Figure 11.2: Overview of EP2 document structure. [Con08].

specification of the cryptographic algorithms used for authentication and financial transaction.

- EP2 *component book (supplementary book)*: Contains additional information on the various EP2 component.

One of the characteristics of such a document structure is that, when considering a particular dialogue between two (or more) EP2 components, the information required to understand that aspect is contained in several different books; each of which describe the dialogue from different points of view. In order to understand one particular dialogue between two EP2 components, one has to look at the individual component book, the interface book and the data dictionary book.

Each document is comprised of a number of different specification notations: plain English; UML-like graphics (use cases, activity diagrams, message sequence charts, class models, etc.); pictures; tables; lists; file descriptions; encoding rules, etc.

The top level EP2 documents provide only an overview of the data involved, while the presentation of further details for a specific type is delayed to separate low-level documents.

CSP-CASL is able to match such a document structure by a library of specifications, where the informal design steps of the EP2 specification are mirrored in terms of a formal refinement relation. Structuring the CSP-CASL specifications in the same way as the original EP2 documents allows to exhibit some ambiguities, omissions and contradictions in the documents.

CSP-CASL's loose specification of data types plays an important role. Usually, the top level EP2 documents provide an overview of the data involved, while the presentation of

further details for a specific type is delayed to separate low-level documents. In the next section we illustrate how we match the informal specification of EP2 in CSP-CASL.

11.2 Modelling EP2 in CSP-CASL

In this section we describe the modelling of the various levels of the EP2 specification in CSP-CASL. We have modeled three levels of abstraction of the EP2 specification. Those are: the *architectural* level, the *abstract component* level and the *concrete component* level. Figure 11.3 illustrate the different levels of specification.

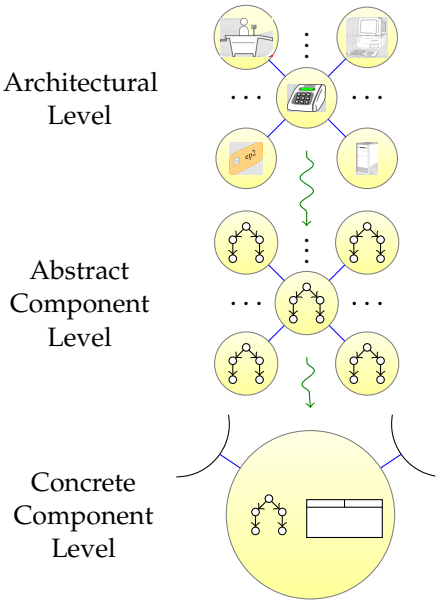

Figure 11.3: EP2 specification at different level.

On the *architectural* level we capture the overall system as depicted in Figure 11.1. On the *abstract component* level, we model the interaction between two EP2 components. Here, we model the use cases of the different functionality of the EP2 components, e.g., configuration of the terminal, initialization different services, processing payments, etc. In some sense, at this level, we narrow down the modelling exercise to an abstract view of the interaction between the different components. On the *concrete component* level we refine the abstract view of each component by modelling which specific values the different components are going to send and receive. Here, we model the behavior of each component to be *stateful*. That is, each component will behave according to what kind of message it is receiving or sending.

In the next subsections, we illustrate the EP2 specifications in CSP-CASL on the different levels. We first give an explanation of how the informal EP2 specification describes the various components and the different methodologies used to specify them. Subsequently, we will show how we capture these aspects in CSP-CASL.

11.2.1 Architectural specification

The architectural specification of EP2 portraits the general overview of the system. An overview of the EP2 architectural level is illustrated in Figure 11.1. It specifies, at a high level, each of the nine interfaces represented by solid blue lines in Figure 11.1 (*CAI_Card, SI_Config, COI_Config, SI_Init, FE_FrontEnd, MI_Subm, MI_Rec, BE_BackEnd, EI_ECR*).

The data specification (ARCH_EP2_DATA) defines sorts describing the data communicated on each of the interfaces listed above; this is loose specification, where in fact all we are doing is defining a name for each sort.

spec ARCH_EP2_DATA =
  sorts *CAI_Card*; *SI_Config*; *COI_Config*; *SI_Init*; *FE_FrontEnd*;
    MI_Subm; *MI_Rec*; *BE_BackEnd*; *EI_ECR*
end

The CSP-CASL specification of th architectural level (ARCH_EP2), begins by declaring channels representing each of the interfaces; each channel's sort comes from the data part. The value of each data is communicated over channels; data of sort *CAI_Card* is interchanged over on a channel *C_CAI_Card* linking the *Card* and the *Terminal*. For each line of communication (or interface) we introduce a new channel. For instance, the *Acquirer* communicates with the *Terminal* and the *PMS* on the *MI_Subm* interface. Here, we model by introducing two different channels over the same interface; namely *C_PMSAcq_MI_Subm* typed over *MI_Subm* and *C_TerAcq_MI_Subm* typed over *MI_Subm*.

Each process is declared with the appropriate alphabet, consisting of the channels over which it may communicate. The *Card* process may communicate only on the *C_CAI_Card* channel.

This is followed by process equations defining the behaviour of each component's process as a CSP-CASL process term. Here, each process is modeled as the *RUN* process, i.e., they are always prepared to communicate any event from their alphabet. For example, the *ServiceCenter* process can communicate all values of channel *C_SI_Config* and *C_COI_Config*. Finally, we declare and define the process *ArchEP2*, which represent the entire system at the architectural level. Its communication alphabet consists of all the channels we have defined.

The *Terminal*, is at the center of the system, and communicates with the 'rest of the system' over the different channels; except *C_COI_Config* and *C_MI_Rec*. The 'rest of the system' is then modelled as three processes interleaved: the *Card*, the *POS*, and a process in which the communications between the *ServiceCenter*, *Acquirer* and *PMS* are restricted using alphabetized parallel; for instance here the *ServiceCenter* cannot communicate with the *PMS*.

ccspec ARCH_EP2 =
  data ARCH_EP2_DATA
  channels *C_CAI_Card* : *CAI_Card*; *C_SI_Config* : *SI_Config*;

```
        C_COI_Config : COI_Config; C_SI_Init : SI_Init;
        C_FE_FrontEnd : FE_FrontEnd; C_TerAcq_MI_Subm : MI_Subm;
        C_PMSAcq_MI_Subm : MI_Subm; C_MI_Rec : MI_Rec;
        C_BE_BackEnd : BE_BackEnd; C_EI_ECR : EI_ECR
process Card : C_CAI_Card ;
        ServiceCenter : C_SI_Config, C_COI_Config ;
        Acquirer : C_COI_Config, C_SI_Init, C_FE_FrontEnd,
                   C_PMSAcq_MI_Subm, C_TerAcq_MI_Subm, C_MI_Rec ;
        PMS : C_BE_BackEnd, C_MI_Rec, C_PMSAcq_MI_Subm;
        POS : C_EI_ECR ;
        Terminal : C_CAI_Card, C_SI_Config, C_SI_Init, C_FE_FrontEnd,
                   C_TerAcq_MI_Subm, C_BE_BackEnd, C_EI_ECR ;
        ArchEP2 : C_CAI_Card, C_SI_Config, C_SI_Init, C_FE_FrontEnd,
                   C_BE_BackEnd, C_EI_ECR, C_COI_Config, C_PMSAcq_MI_Subm,
                   C_TerAcq_MI_Subm, C_MI_Rec ;
        Card = RUN(C_CAI_Card )
        ServiceCenter = RUN(C_SI_Config, C_COI_Config )
        Acquirer = RUN(C_COI_Config, C_SI_Init, C_FE_FrontEnd,
                       C_PMSAcq_MI_Subm,C_TerAcq_MI_Subm, C_MI_Rec )
        PMS = RUN(C_BE_BackEnd, C_PMSAcq_MI_Subm, C_MI_Rec )
        POS = RUN(C_EI_ECR )
        Terminal = RUN(C_CAI_Card, C_SI_Config, C_SI_Init, C_FE_FrontEnd,
                       C_TerAcq_MI_Subm, C_BE_BackEnd, C_EI_ECR )
        ArchEP2 =
        Terminal |[ C_CAI_Card, C_SI_Config, C_SI_Init, C_FE_FrontEnd,
                   C_TerAcq_MI_Subm, C_BE_BackEnd, C_EI_ECR]|
        (Card
        ||| ( ServiceCenter |[ C_SI_Config, C_COI_Config
                             || C_COI_Config, C_SI_Init,C_FE_FrontEnd,
                             C_TerAcq_MI_Subm, C_PMSAcq_MI_Subm, C_MI_Rec ]|
                             Acquirer |[ C_SI_Config, C_COI_Config, C_SI_Init,
                                         C_FE_FrontEnd, C_TerAcq_MI_Subm,
                                         C_PMSAcq_MI_Subm, C_MI_Rec
                                         || C_PMSAcq_MI_Subm, C_MI_Rec,
                                           C_BE_BackEnd ]| PMS )
        ||| POS )
end
```

This very first specification in CSP-CASL mirrors the informal architectural specification of EP2, portrayed in Figure 11.1. It is, a very abstract view of the system — but certainly not a trivial one. Even from such an abstract specification, we would like to verify interesting properties (see Section 11.3); and design meaningful test cases (see Section 11.4).

11.2.2 Abstract component level

At this level we model the activity diagram of the different functionality of the EP2 components. For the sake of good understanding, in this section we will present the details of the modelling of the interaction between two components, namely the *ServiceCenter* and the *Terminal*. In this interaction, both components exchange messages in order to configure the *Terminal* capabilities.

The *get configuration* use case describes how the *ServiceCenter* informs and maintains the *Terminal* configuration data. This communication is carried out over the *SI_Config* channel. This interface is a bidirectional message flow. The *Terminal* acts as communication master and the *ServiceCenter* as communication slave. The interface is used for the download of terminal specific configuration parameters; for instance information about the *Acquirer* initialisation server data. The *ServiceCenter* may optionally request information about the terminal configuration and initialisation data.

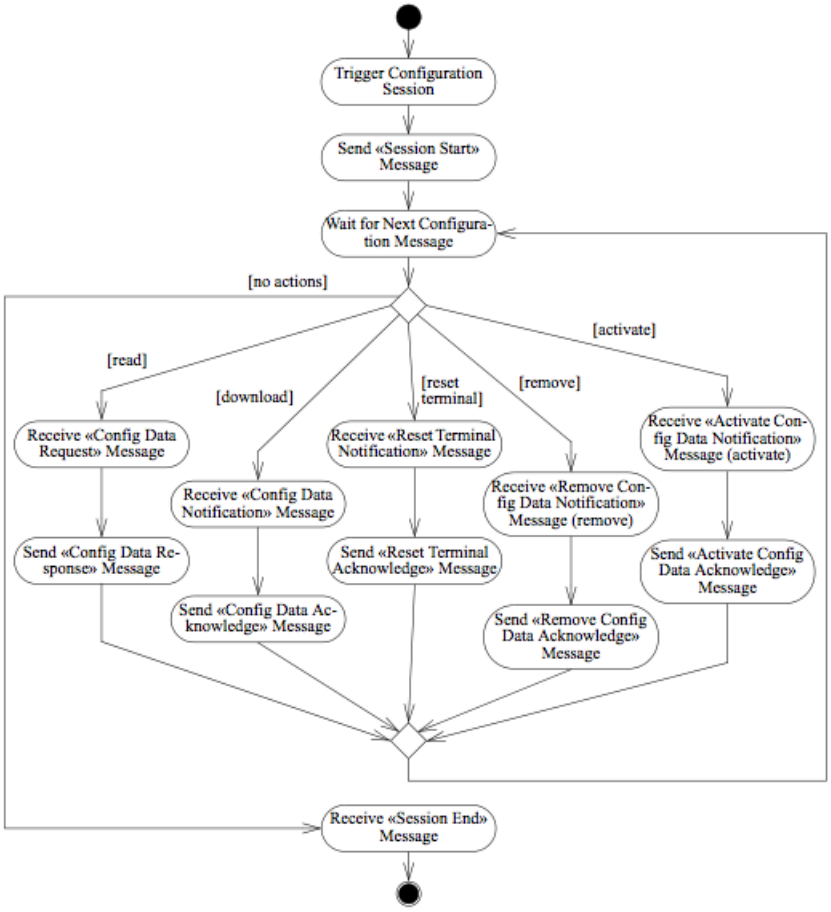

Figure 11.4: EP2 Get configuration activity diagram – *Terminal* part [Con08].

For both the *Terminal* and the *ServiceCenter*, activity diagrams are given describing the

flow of control on the receipt of messages. Figure 11.4 and 11.5 shows the diagrams of the *Terminal* and *ServiceCenter* component in the context of exchanging the configuration data.

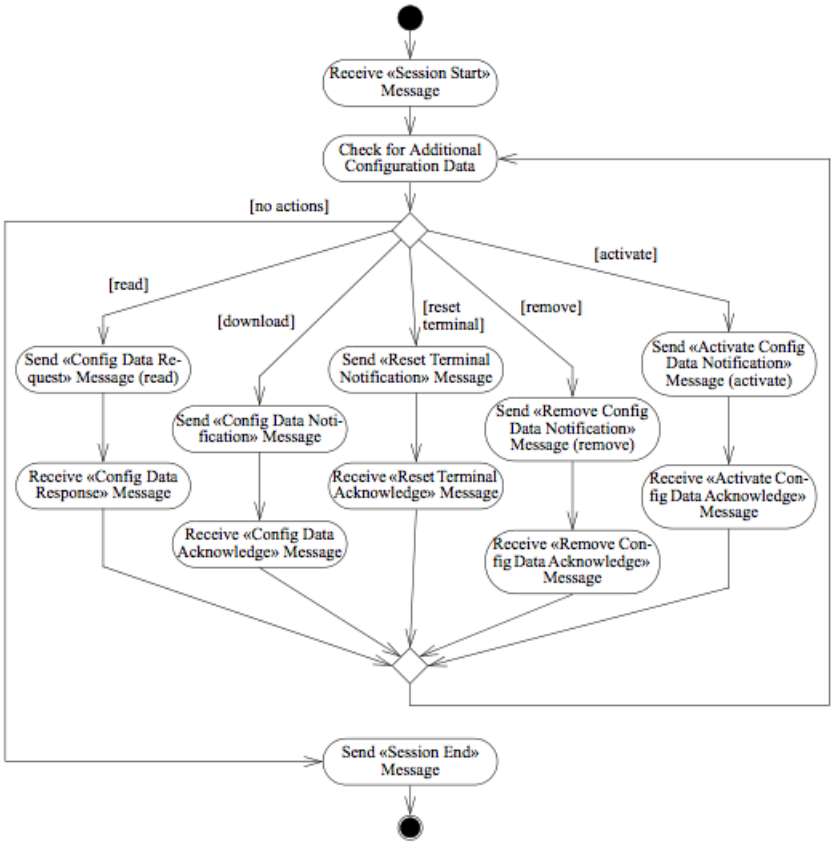

Figure 11.5: EP2 Get configuration activity diagram – *ServiceCenter* part [Con08].

At this level, the data specifications are refined by introducing a type system on messages. In CASL, this is realised by introducing subsorts of the various data sorts introduced in ARCH_EP2, e.g., $CAI\_Card, \ldots, EI\_ECR$.

On the data part (D_ACL_GETCONFIG), we introduce suitable subsort which corresponds to the various messages which are sent between the *ServiceCenter* and the *Terminal*. In the case of the get configuration dialogue, those are the messages which appear in the activity diagram (see Figure 11.4): *Session start*, *Config data request*, *Config data response*, etc. Here, we have to make sure that the various messages are different by adding some suitable axioms.

spec D_ACL_GETCONFIG =
sorts *SessionStart*, *SessionEnd*, *ConfigDataRequest*, *ConfigDataResponse*, *ConfigNotif*, *ConfigAck*, *TerminalClearNotif*, *TerminalClearAck*, *RemoveConfigNotif*, *RemoveConfigAck*, *ActivateConfigNotif*, *ActivateConfigAck* $<$ *D_SI_Config*
$\forall\, x : SessionEnd; y : ConfigDataRequest \bullet \neg\, x = y$

$\forall\, x : SessionEnd; y : ConfigNotif \bullet \neg\, x = y$
$\forall\, x : SessionEnd; y : TerminalClearNotif \bullet \neg\, x = y$
$\forall\, x : SessionEnd; y : RemoveConfigNotif \bullet \neg\, x = y$
$\ldots\ \ldots\ \ldots$
$\forall\, x : RemoveConfigNotif; y : ActivateConfigNotif \bullet \neg\, x = y$

end

On the process part (ACL_GETCONFIGURATION), the process *RUN* from the architectural specification is refined without changing the overall communication structure. In the case of the *get configuration* dialogue, we specify how the *Terminal* and the *ServiceCenter* reacts to the sending and receiving of the various messages.

ccspec ACL_GETCONFIGURATION =
data D_ACL_GETCONFIG
channel *C_SI_Config* : *D_SI_Config*
process
TerminalConfiguration = *Ter_Config* ⟦ *C_SI_Config* ⟧ *SC_Config*
Ter_Config = *C_SI_Config* ! *sesStart* :: *SessionStart* →*Ter_Mgm*
Ter_Mgm = *C_SI_Config* ? *configMess* :: *D_SI_Config* →
if *configMess* ∈ *D_SI_Config_SessionEnd* **then** *SKIP*
else if *configMess* ∈ *ConfigDataRequest*
then *C_SI_Config* ! *resp* :: *ConfigDataResponse* →*Ter_Mgm*
else if *configMess* ∈ *ConfigNotif*
then *C_SI_Config* ! *ack* :: *ConfigAck*→ *Ter_Mgm*
else if *configMess* ∈ *TerminalClearNotif*
then *C_SI_Config* ! *ackT* :: *TerminalClearAck* → *Ter_Mgm*
else if *configMess* ∈ *RemoveConfigNotif*
then *C_SI_Config* ! *ackR* :: *RemoveConfigAck* → *Ter_Mgm*
else if *configMess* ∈ *ActivateConfigNotif*
then *C_SI_Config* ! *ackA* :: *ActivateConfigAck* →*Ter_Mgm*
else *STOP*
SC_Config = *C_SI_Config* ? *sesStart* :: *SessionStart* → *SC_Mgm*
SC_Mgm = *C_SI_Config* ! *seM* :: *SessionEnd* → *SC_Config*
⊓ *C_SI_Config* ! *cdrM* :: *ConfigDataRequest* →
C_SI_Config ? *response* :: *ConfigDataResponse* → *SC_Mgm*
⊓ *C_SI_Config* ! *cdnM* :: *ConfigNotif* →
C_SI_Config ? *confAck* :: *ConfigAck* → *SC_Mgm*
⊓ *C_SI_Config* ! *tclearM* :: *TerminalClearNotif* →
C_SI_Config ? *tclearAck* :: *TerminalClearAck* → *SC_Mgm*
⊓ *C_SI_Config* ! *rcdnM* :: *RemoveConfigNotif* →
C_SI_Config ? *rmConfAck* :: *RemoveConfigAck* → *SC_Mgm*
⊓ *C_SI_Config* ! *acdnM* :: *ActivateConfigNotif* →
C_SI_Config ? *actAck* :: *ActivateConfigAck* → *SC_Mgm*

end

The process *TerminalConfiguration* models the interaction between the terminal (*Ter_Config*) and the service center (*SC_Config*), which runs in parallel via the channel *C_SI_Config*. The process *Ter_Config* initiates the dialogue by sending a message of type *D_SI_Config_SessionStart*; on the other side the process *SC_Config* receives this message. The process *SC_Config* takes the internal decision either to end the dialogue by sending the message of type *D_SI_Config_SessionEnd* or to send another type of message, for example a message of type *D_SI_Config_ConfigRequest*. This is modelled using the internal choice operator ($\sqcap$) of CSP. On the other side the process *Ter_Config*, depending on what kind of message the process *SC_Config* has requested, engages in a data exchange. For example, if it receives a message of type *ConfigDataRequest* it will send a message of type *ConfigDataResponse*.

This model captures in a faithful way the activity diagram of the *Terminal* and the *Service Center* represented in Figure 11.5 and 11.4.

11.2.3 Concrete component level

In the abstract component level we have captured the activity diagram of the various components as depicted in the original EP2 specification. Here, for each state of the activity diagram, a verbal description is given of which message parameters are admissible in this state, and what appropriate response messages are composed of. For example, in the activity diagram of the *ServiceCenter* (Figure 11.5), the state "*Send $\ll$Config data Request$\gg$ Message*" is accompanied by the following verbal description:

> *The service center shall send the message $\ll$ Config data Request $\gg$ Message to the terminal. The service center shall set $<$ Config Data Object $>$ to the configuration data object, which the service center is interested in. For CPTD, TACD and CAD the service center shall specify with an AID respectively ...*

The parameter values of the various configuration data objects, such as CPTD, TACD etc, are informally described in another table. Figure 11.6 illustrates an excerpt from such table.

For the concrete encoding of the various messages, we have to look in to two other books: the EP2 *interface book* and the EP2 *data dictionary book*. For instance in the *interface book*, we find the sequence diagram for the various activity. Figure 11.7 illustrate the sequence diagram for requesting the configuration data.

Moreover, in such documents, we find details of the data elements. For instance, for the message *$\ll$Config data Request$\gg$* we find the data elements reported in Figure 11.2.3. Here, the table presents the various XML tags and some conditions. The condition for *Acquirer Identifier* asserts that this data element should only be present in the case the requested data object is of type *ACD* and *AISD* and is optional in the case of a data object of type *LAID*.

<Config Data Object> value	Object Name	Additional Data Element	Returned by Terminal
ACD	Acquirer Config Data	<Acquirer Identifier>	One ACD object of the requested acquirer
AISD	Acquirer Init Srv Data	<Acquirer Identifier>	One AISD object of the requested acquirer
CAD	Certification Auth Data	<Registered Application Provider Identifier (RID)>	One CAD object of the requested RID.
CPTD	Card Profile Table Data	<Application Identifier (AID)>	One CPTD object of the requested AID.
LAID	List of AID's	-	A list of all AID's supported by this terminal
LAID	List of AID's	<Acquirer Identifier>	A list of all AID's supported by this acquirer
TACD	Terminal Application Config Data	<Application Identifier (AID)>	One TACD object of the requested AID.
TCD	Terminal Config Data	-	The TCD object

Figure 11.6: Message parameters for terminal configuration data [Con08].

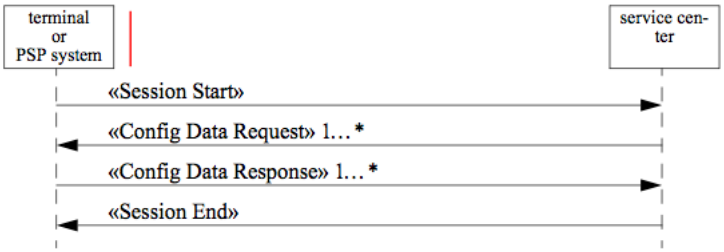

Figure 11.7: Sequence diagram 'Request configuration data' [Con08].

Name	XML-Tag	Condition
<Acquirer Identifier>	ep2:AcqID	only for ACD and AISD, optional for LAID
<Application Identifier (AID)>	ep2:AID	only for TACD and CPTD
<Config Data Object>	ep2:ConfDataObj	
<Registered Application Provider Identifier (RID)>	ep2:RID	only for CAD not applied for e-commerce (PSP)
<Service Center Identifier>	ep2:SCID	
<Terminal Identification>	ep2:TrmID	

Figure 11.8: Data elements for ≪*Config data Request*≫ [Con08].

On the concrete component level, we model which specific values the different EP2 components are going to send and receive. It is at this level, the processes becomes *stateful*. Here, the state is represented by a pair:

$$p : Pair[State][Trigger]$$

State represents the EP2 *Terminal*'s memory, while *Trigger* represents what kind of messages initiate the communication (e.g., configuration management in the case of the get configuration data).

The data part (D_CCL_GETCONFIG) becomes more elaborated and detailed. In order to model the state of the processes, we import from the CASL standard library the specification of PAIR and MAYBE. The latter, is necessary to model the fact that in the data models certain elements are optional (See Figure 11.2.3). The following illustrate the MAYBE specification:

spec MAYBE[**sort** S] =
  sort *Maybe*[S]
  ops *nothing* : *Maybe*[S];
    just : $S \rightarrow$ *Maybe*[S];
    getJust : *Maybe*[S] $\rightarrow ?$ S
  pred *defined* : *Maybe*[S]
    • $\neg$ *def getJust*(*nothing*)
    • $\forall\, x : S \bullet$ *getJust*(*just*(x)) = x
    • $\forall\, x$: *Maybe*[S] • *defined*(x) $\Leftrightarrow$ *def getJust*(x)
end

Finally, we import the data specification from the abstract component level of the get configuration (D_ACL_GETCONFIG). We then extend the data with new sorts, i.e., *AcquirerID*, *AID*[3] etc.

spec D_CCL_GETCONFIG =
  PAIR [**sort** *State* **fit sort** $S \mapsto$ *State*] [**sort** *Trigger* **fit sort** $T \mapsto$ *Trigger*]
  and MAYBE[**sort** *ACD*]
  and MAYBE[**sort** *AISD*]
  and MAYBE[**sort** *CAD*]
  and MAYBE[**sort** *CPTD*]
  and MAYBE[**sort** *CAD*]
  and MAYBE[**sort** *TACD*]
  and MAYBE[**sort** *TCD*]
  and MAYBE[**sort** *AcqID*]
  and MAYBE[**sort** *AID*]
  and MAYBE[**sort** *RID*]
  and D_ACL_GETCONFIG
then sorts *AcquirerID*, *AID*, *RID*, *TerminalRangeID*, *TerminalUnitID*, *ServiceCenterID*, ...

The concrete value of each message triggers a specific behavior in the process part. Thus, it is necessary to specify the data types up to representation. Specifically, at this level we would like to capture the data elements such as those in Figure 11.6. These messages can be modelled by a CASL free type, and we can make concrete what data is involved in each message. In the following for instance, we specify what type of elements the configuration data object (*Config Data Obj*) contains. Those are exactly the elements specified in

[3]AID: stands for Application Identifier. RID: stands for Registered Application Identifier. TerminalRangeID is the a unique number assigned to a terminal by the EP2 registration authority. TerminalUnitID is a unique identifier assigned to a terminal by the merchant. Both the TerminalRangeID and the TerminalUnitID constitutes a unique identfier for a particular terminal.

Figure 11.2.3.

free type *ConfigDataObj* ::= *ACD* | *AISD* | *CPTD* | *CAD* | *TACD* | *TCD*

The specific message for requesting data configuration (*ConfigDataRequest*) is then specified as follows:

free type *ConfigDataRequest* ::=
mk_ConfigDataRequest(*get_AcqID* : *Maybe*[*AcqID*];
get_AID : *Maybe*[*AID*];
get_req : *ConfigDataObj*;
get_RID : *Maybe*[*RID*];
get_SCID : *ServiceCenterID*;
get_TrmID : *TerminalID*)

We now have to add some suitable axioms in order to capture the conditions specified in the data elements of Figure 11.2.3. For example, in the following we specify the condition that if the requested message is of type *ACD* the *Acquirer ID* is defined while the application identifier (*AID*) and the registered application provider identifier (*RID*) are not defined.

$\forall$ *cdr* : *ConfigDataRequest*
• $get\_req(cdr) = ACD \Rightarrow defined(get\_AcqID(cdr)) \wedge \neg\, defined(get\_AID(cdr)) \wedge \neg\, defined(get\_RID(cdr))$
• $get\_req(cdr) = AISD \Rightarrow defined(get\_AcqID(cdr)) \wedge \neg\, defined(get\_AID(cdr)) \wedge \neg\, defined(get\_RID(cdr))$
• $get\_req(cdr) = CPTD \Rightarrow \neg\, defined(get\_AcqID(cdr)) \wedge defined(get\_AID(cdr)) \wedge \neg\, defined(get\_RID(cdr))$
• $get\_req(cdr) = CAD \Rightarrow \neg\, defined(get\_AcqID(cdr)) \wedge \neg\, defined(get\_AID(cdr)) \wedge defined(get\_RID(cdr))$
• $get\_req(cdr) = TACD \Rightarrow \neg\, defined(get\_AcqID(cdr)) \wedge defined(get\_AID(cdr)) \wedge \neg\, defined(get\_RID(cdr))$
• $get\_req(cdr) = TCD \Rightarrow \neg\, defined(get\_AcqID(cdr)) \wedge \neg\, defined(get\_AID(cdr)) \wedge \neg\, defined(get\_RID(cdr))$

Once the *Terminal* receives a request for a configuration data from the *ServiceCenter*, the response of the terminal is dependent of what kind of message the *ServiceCenter* has requested. The data response from the *Terminal* has the following format:

type *ConfigDataResponse* ::=
mk_ConfigDataResponse(*get_SCID* : *ServiceCenterID*;
get_TrmID : *TerminalID*;
get_ACD : *Maybe*[*ACD*];
get_AISD : *Maybe*[*AISD*];
get_CAD : *Maybe*[*CAD*];
get_CPTD : *Maybe*[*CPTD*];
get_TACD : *Maybe*[*TACD*];
get_TCD : *Maybe*[*TCD*])

In order to compute the correct data response for the configuration data request, we declare a function which takes the *ConfigDataRequest* and the *State* of the terminal and computes the *ConfigDataResponse*.

op $msg\_DataResp : ConfigDataRequest \times State \rightarrow ConfigDataResponse$

Which specific value should the data response contain is stated by adding some axioms. For example, in the following we specify that if the data request contains the message *ACD*, then the terminal should retrieve the data elements of *ACD*.

$\forall\ cdr : ConfigDataRequest; s : State$
$\bullet\ get\_req(cdr) = ACD \Rightarrow defined(get\_ACD(msg\_DataResp(cdr, s)))$
$\wedge\ \neg\ defined(get\_AISD(msg\_DataResp(cdr, s)))$
$\wedge\ \neg\ defined(get\_CAD(msg\_configDataResponse(cdr, s)))$
$\wedge\ \neg\ defined(get\_CPTD(msg\_DataResp(cdr, s)))$
$\wedge\ \neg\ defined(get\_TACD(msg\_DataResp(cdr, s)))$
$\wedge\ \neg\ defined(get\_TCD(msg\_DataResp(cdr, s)))$

On the process part, we have that at the concrete component level, the activity of the *ServiceCenter* remains unchanged. Basically it is the same as specified in the abstract component level (see ACL_GETCONFIGURATION). However, the *Terminal*'s reactive behavior changes completely. Here, we would like to capture the fact that the *Terminal* is stateful, i.e., it depends on the pair $State \times Trigger$.

ccspec CCL_GETCONFIGURATION =
data D_CCL_GETCONFIG
channel *C_SI_Config* : *D_SI_Config*
process
TerminalConfiguration(*Pair* [*State*,*Trigger*]) : *C_SI_Config* ;
Ter_Config(*Pair*[*State*,*Trigger*]) : *C_SI_Config* ;
Ter_Mgm(*Pair* [*State*, *Trigger*]) : *C_SI_Config* ;

TerminalConfiguration(*p*) = *Ter_Config* (*p*) ⟦ *C_SI_Config* ⟧ *SC_Config*

Ter_Config(*p*) = *C_SI_Config* ! *msg_sessionStartConf* (*second*(*p*)) → *Ter_Mgm*(*p*)

Ter_Mgm(*p*) = *C_SI_Config* ? *configMess* :: *D_SI_Config* →
if *configMess* ∈ *D_SI_Config_SessionEnd* **then** *SKIP*
else if *configMess* ∈ *ConfigDataRequest*
then *C_SI_Config* ! *msg_DataResp* (*configMess as ConfigDataRequest*, *first* (*p*))
→*Ter_Mgm*(*p*)
else if *configMess* ∈ *ConfigNotif*
then *C_SI_Config* ! *msg_configAck* (*configMess as ConfigNotif*, *first*(*p*))
→*Ter_Mgm* (*pair* ((*st_configAck* (*configMess as ConfigNotif*, *first* (*p*))),
second(*p*)))
else if *configMess* ∈ *TerminalClearNotif*
then *C_SI_Config* ! *msg_clearlearNotif* (*configMess as TerminalClearNotif*, *first*(*p*))

→*Ter_Mgm* (*pair*((*st_terClearNotif* (*configMess as TerminalClearNotif*, *first*(*p*))),
second(*p*)))
else if *configMess* ∈ *RemoveConfigNotif*
then *C_SI_Config* ! *msg_removeAck* (*configMess as RemoveConfigNotif*, *first*(*p*))
→ *Ter_Mgm* (*pair*((*st_removeAck*(*configMess as RemoveConfigNotif*, *first* (*p*))),
second(*p*)))
else if *configMess* ∈ *ActivateConfigNotif*
then *C_SI_Config* ! *msg_actConfDataAck*(*configMess as ActivateConfigNotif*, *first*(*p*))
→ *Ter_Mgm* (*pair* ((*st_actConfAck* (*configMess as ActivateConfigNotif*, *first* (*p*))),
second(*p*)))
else *STOP*
(... Service Center process specification ...)
end

In CCL_GETCONFIGURATION the process *TerminalConfiguration* is dependent on the parameter *Pair*[*State*, *Trigger*], i.e., *TerminalConfiguration*(*Pair*[*State*, *Trigger*]). On the terminal side (*Ter_Config*), a message *configMess* is received from the *ServiceCenter* over the channel *C_SI_Config*. Depending on the type of *configMess*, different answers are sent back to the *ServiceCenter* and the internal state of the *Terminal* changes. For example, in the case *configMess* is of type *ConfigDataRequest*, the terminal replies with the current configuration message. The function *msg_DataResp*(*configMess as ConfigDataRequest*, *first*(*p*)) computes the right data elements requested; here, in *configMess as ConfigDataRequest* we need to downcast the type, since *configMess* is of type *D_SI_Config* and *ConfigDataRequest* is a subsort of *D_SI_Config* (see [Gim08]). The activity of requesting configuration data from the *Terminal* doesn't change the internal state of the *Terminal*; thus we don't model the change of the state.

In the case *ServiceCenter* informs the *Terminal* of some changes, that is when *configMess* is of type *ConfigNotif*, the internal state of the *Terminal* changes. This is computed by the function *st_configAck*(*configMess as ConfigNotif*, *first*(*p*)).

The complete specification of the *get configuration* dialogue at the three levels of abstraction can be found in the Appendix C.5.

11.3 Property verification of EP2

In this section we prove some interesting properties of EP2. Namely we show that refinement steps from the different layers of specification hold. Furthermore, for selected dialogue of EP2 components we prove the absence of deadlock and livelock. Here we take a single dialogue, namely the *get configuration* dialogue between the service center and the terminal, to illustrate how the verification is done in CSP-CASL.

11.3.1 Refinement

Our notion of CSP-CASL refinement presented in Chapter 6 is capable of capturing the vertical development steps presented in the previous section. Summarizing the vertical development of EP2: the first system design sets up the interface between the components (architectural level), then these components are developed further to capture the dialogue between the components (abstract component level), in the next level the system becomes stateful and heavily dependent on the specific messages exchanged between the parties. Figure 11.9 shows the overall idea how the refinement verification is carried out.

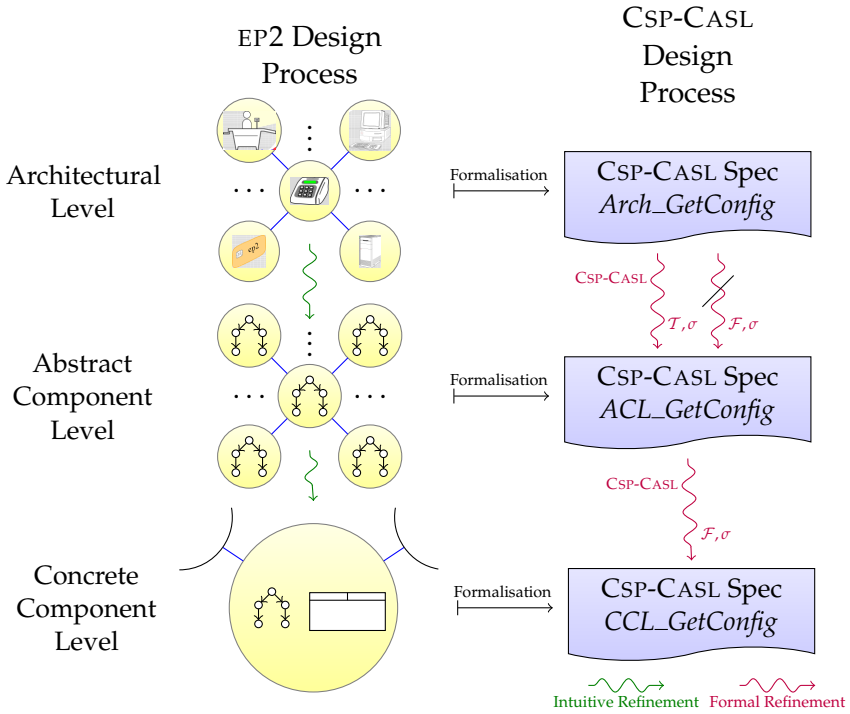

Figure 11.9: EP2 refinement verification in CSP-CASL.

In order to prove the refinement from the architectural level to the abstract component level for the *get configuration* dialogue, we introduce some intermediate specifications and refinement proofs.

In the first step, we restrict the EP2 architectural specification (ARCH_EP2) to include only the communication over the *SI_Config* channel between the *ServiceCenter* and the *Terminal*. This is captured in the following specification:

ccspec ARCH_GETCONFIG =
data D_SI_Config
channel $C\_SI\_Config : D\_SI\_Config$
process
  let $ServiceCenter = \text{EP2RUN} \quad Terminal = \text{EP2RUN}$
  in $Terminal \,||[\, C\_SI\_Config \,]||\, ServiceCenter$
end

The EP2RUN process is specified as:

$$\text{EP2RUN} = (C\_SI\_Config\ ?x : D\_SI\_Config \rightarrow Config\_RUN) \,\Box\, SKIP$$

Such process is always prepared to communicate an event from *D_SI_Config* or to terminate successfully. We now prove the following lemma:

LEMMA 11.3.1 $\text{EP2RUN} =_{\mathcal{T}} \text{EP2RUN} \,||[\, C\_SI\_Config \,]||\, \text{EP2RUN}$.

PROOF. The proof is done in CSP-CASL-PROVER. ∎

Then, we introduce a new specification SEQ_GETCONFIG. Such specification is a sequential version of the *get configuration* dialogue:

ccspec SEQ_GETCONFIG =
data D_ACL_GETCONFIG
channel $C\_SI\_Config : D\_SI\_Config$
process
$Seq\_Start = C\_SI\_Config\ !\ sesStart :: SessionStart \rightarrow SC\_Mgm$
$Seq\_Mgm = C\_SI\_Config\ !\ seM :: SessionEnd \rightarrow SC\_Config$
$\quad \sqcap\ C\_SI\_Config\ !\ cdrM :: ConfigDataRequest \rightarrow$
$\quad\quad C\_SI\_Config\ !\ response :: ConfigDataResponse \rightarrow Seq\_Mgm$
$\quad \sqcap\ C\_SI\_Config\ !\ cdnM :: ConfigNotif \rightarrow$
$\quad\quad C\_SI\_Config\ !\ confAck :: ConfigAck \rightarrow Seq\_Mgm$
$\quad \sqcap\ C\_SI\_Config\ !\ tclearM :: TerminalClearNotif \rightarrow$
$\quad\quad C\_SI\_Config\ !\ tclearAck :: TerminalClearAck \rightarrow Seq\_Mgm$
$\quad \sqcap\ C\_SI\_Config\ !\ rcdnM :: RemoveConfigNotif \rightarrow$
$\quad\quad C\_SI\_Config\ !\ rmConfAck :: RemoveConfigAck \rightarrow Seq\_Mgm$
$\quad \sqcap\ C\_SI\_Config\ !\ acdnM :: ActivateConfigNotif \rightarrow$
$\quad\quad C\_SI\_Config\ !\ actAck :: ActivateConfigAck \rightarrow Seq\_Mgm$
end

We prove that SEQ_GETCONFIG is equivalent over the stable failure model to ACL_GETCONFIG[4].

LEMMA 11.3.2 ACL_GETCONFIG $=_{\mathcal{F}}$ SEQ_GETCONFIG.

[4] A similar proof methodologies have been applied in [OIR09]

PROOF. The proof is done in CSP-CASL-PROVER. Here, both processes uses the same data specification. The following is a snippet of the Isabelle proof script.

```
theorem GetConfig_Seq: "Acl_GetConfig =F Seq_GetConfig"
  apply(simp add: Acl_GetConfig_def  Seq_GetConfig_def)
  apply(rule cspF_fp_induct_right[of _ _ "Seq_To_Config"])
  apply(simp_all)
  apply(simp)
  apply(induct_tac procName)
  apply(cspF_auto | cspF_hsf | rule cspF_decompo)+
done
```

We first unfold the definitions of `Acl_GetConfig` and `Seq_GetConfig`. Next, we apply (metric) fixed point induction on the rhs and make a case distinction over the process names, here encoded as `induct_tac procName`. In the last step we apply powerful CSP-PROVER tactics which combines three basic tactics. The result of which rewrites the processes to head normal form (`cspf_hsf`), and applies simplification (`cspF_auto` and `cspF_decompo`) in terms of CSP step laws in order to equate processes. CSP-PROVER tactics are explained in details in [IR]. ■

Finally we prove the main refinement proof:

THEOREM 11.3.3 ARCH_GETCONFIG $\leadsto^{\sigma}_{\mathcal{T}}$ ACL_GETCONFIG.

PROOF. Having established some equivalence in Lemma 11.3.1 and 11.3.2, we now use CSP-CASL-PROVER to establish this refinement.

We want to prove that EP2RUN $\leadsto^{\sigma}_{\mathcal{T}}$ SEQ_GETCONFIG. To this end we apply the decomposition theorem presented in Section 7.1. Using HETS, we automatically prove the data refinement D_ARCH_GETCONFIG $\overset{\text{data}}{\leadsto}_{\sigma}$ D_ACL_GETCONFIG.

Now, we have formed the specification $(\text{D\_ACL\_GETCONFIG}, P_{\text{SEQ\_GETCONFIG}})$.
Where $P_{\text{SEQ\_GETCONFIG}}$ denotes the process part of SEQ_GETCONFIG. Next we show in CSP-CASL-PROVER that, over the traces model $\mathcal{T}$, the refinement RUN_GETCONFIG $\leadsto^{\sigma}_{\mathcal{T}}$ SEQ_GETCONFIG holds. Here, we show a snippet of the Isabelle proof script for the refinement proof.

```
theorem Arch_ACL_GetConfig : "EP2RUN <=T Seq_Start"
  apply(unfold EP2RUN_def Seq_Start_def)
  apply(rule cspT_fp_induct_right[of _  _"SeqToRun"])
  apply(induct_tac procName)
  ...
  apply(rule cspT_rw_left | rule cspT_decompo)+
  ...
  apply(cspT_auto | auto)
  apply(simp add: cspT_semantics)
  apply(rule)
  apply(simp add: in_traces)
  apply(auto simp add: trace_nil_or_Tick_or_Ev)
  apply(auto simp add: SessionStart_def SessionEnd_def ... )
done
```

We first unfold the definitions of `Config_Run` and `Seq_Start`. Next, we apply (metric) fixed point induction on the rhs and make a case distinction over the process names, here encoded as `induct_tac procName`. After rewriting and decomposing both of the processes we compute the trace semantics (`cspT_semantics`) and check that there is indeed an inclusion of traces (`in_traces`); here, we need to add the definition of the various sorts (`SessionStart_def` etc). ■

As illustrated in Figure 11.9 the refinement from the architectural level to the abstract component level doesn't hold over the stable failure model $\mathcal{F}$. We recall that refinement in $\mathcal{F}$ holds if and only if there is an inclusion of the trace and the failure set. In our case, however the process EP2RUN has less refusal set than the process part of SEQ_GETCONFIG, i.e., *failures*(SEQ_GETCONFIG) $\not\subseteq$ *failures*(EP2RUN).

We now prove the refinement step from the abstract component level to the concrete component level, in the context of the *get configuration* dialogue.

THEOREM 11.3.4 ACL_GETCONFIG $\rightsquigarrow^{\sigma}_{\mathcal{F}}$ CCL_GETCONFIG.

PROOF. Again we use the decomposition theorem to first establish the data refinement D_ACL_GETCONFIG $\overset{\text{data}}{\rightsquigarrow_{\sigma}}$ D_CCL_GETCONFIG. Such proof is discharged automatically in HETS. Now, we have formed the specification $(\text{D\_CCL\_GETCONFIG}, P_{\text{CCL\_GETCONFIG}})$. Where $P_{\text{CCL\_GETCONFIG}}$ denotes the process part of CCL_GETCONFIG. Next we show in CSP-CASL-PROVER that, over the stable failure model $\mathcal{F}$, the refinement

$$\text{ACL\_GETCONFIG} \rightsquigarrow^{\sigma}_{\mathcal{F}} \text{CCL\_GETCONFIG}$$

holds. The proof in CSP-CASL-PROVER is relatively substantially longer than the proof of Theorem 11.3.3. Here, we illustrate the main snippet of the Isabelle proof script.

```
theorem ACL_TO_CCL: "!!p. ACL_GetConfig <=F CCL_GetConfig p"
  apply(simp add: ACL_Configuration_def CCL_Configuration_def)
  apply(rule cspF_decompo)
  apply(simp)
  ...
     (* Refinement on the Terminal side *)
  ...
  apply(rule cspF_fp_induct_left[of _ "ACL_TO_CCL"])
  apply(simp_all)
  apply(rule cspF_Rep_int_choice_left)
  apply(simp)
  apply(rule_tac x="p" in exI)
  apply(simp)
  apply(simp add: config_general)
  ...
      (* Refinement on the ServiceCenter side*)
  ...
  apply(rule cspF_fp_induct_left[of _ "ACL_TO_CCL"])
  apply(simp_all)
  apply(simp)
  apply(simp add: config_general)
done
```

We first unfold the definitions of `ACL_GetConfig` and `CCL_GetConfig`. The latter process is parameterised by P, which is pair of *State* and *Trigger* (See Section 11.2). The proof of refinement is done first on the *Terminal* side and then on the *ServiceCenter* side. On both sides, we apply (metric) fixed point induction on the rhs and apply simplification tactics. ■

11.3.2 Deadlock analysis

Most of EP2 components interact with each other over some channels (see Figure 11.1). As illustrated in Section 11.2, this is modeled as a parallel composition, in which both components communicate over a channel, i.e., in CSP this is realized using the generalized parallel operator $P \,||[\, C \,]||\, Q$.

In such interaction it is possible that the deadlock phenomenon could occur. Furthermore, processes like *Ter_Mgm* of the specification ACL_GETCONFIG includes the CSP process *STOP* within one branch of its conditional. Should this branch of *Ter_Mgm* be reached, the whole system will be in deadlock. This is of course an undesirable situation, especially in the case of payment transaction, i.e., communication between the *Terminal* and the *Acquirer* over the *FE_FrontEnd* in order to authorize a payment (see Figure 11.10). Hence, an early analysis of such undesirable behavior is very beneficial for the overall verification of the EP2 system.

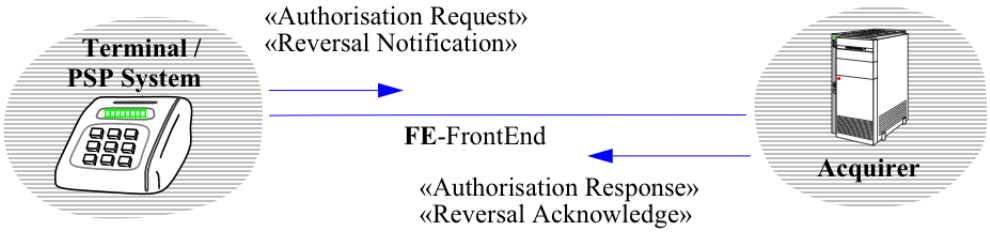

Figure 11.10: EP2 process transaction[Con08].

In this subsection we illustrate how the deadlock analysis is carried out in the *get configuration* dialogue. Such analysis is done at the abstract component level.

In Lemma 11.3.2 we have proven that the *Get configuration* dialogue is equivalent on the stable failure model to its sequential version (SEQ_GETCONFIG). By syntactic characterization such process is deadlock free. Here, we prove that indeed SEQ_GETCONFIG is deadlock free. To this end we establish in CSP-CASL-PROVER that DF $\rightsquigarrow^{\sigma}_{\mathcal{F}}$ SEQ_GETCONFIG, where DF is the least refined deadlock free process described in Section 8.1.

LEMMA 11.3.5 DF $\rightsquigarrow^{\sigma}_{\mathcal{F}}$ SEQ_GETCONFIGUARTION.

PROOF. The proof is done in CSP-CASL-PROVER, here we give the main snippet of the Isabelle proof script:

```
theorem GetConfiguration_Is_DF: "DF <=F Seq_GetConfig"
  apply(unfold Seq_GetConfig_def DF_def)
  apply(rule cspF_fp_induct_right[of _ _ "Seq_to_DF"])
  apply(simp_all)
  apply(simp)
  apply(induct_tac procName, auto)
  apply(cspF_auto)+
  apply(rule cspF_Int_choice_left1)
  apply(rule cspF_decompo_ref)
  apply(cspF_auto | auto )+
  apply(rule cspF_Int_choice_left2)
  apply(cspF_auto | auto)+
  ...
  apply(rule cspF_Int_choice_left1 | rule cspF_decompo_ref
                                   | cspF_auto | auto)+
done
```

First the main goal is unfolded, then we apply the (metric) fixed point induction is applied; here we have defined a mapping from the process *Seq_GetConfig* to the process *DF*. Next, the involved recursive processes are proven to be guarded using the Isabelle simplification tactics. Then, we make a case distinction over the process names (`induct_tac procName`). At this point, we have seven subgoals of the following form:

$$x \in \mathit{SessionEnd} \Longrightarrow \mathit{DF} \sqsubseteq_{\mathcal{F}} \mathit{C\_SI\_Config}\ x \rightarrow \mathit{SKIP}$$

Each of these subgoals corresponds to the internal choice branching of the SEQ_GETCONFIG. In order to discharge these subgoals we use CSP-PROVER tactics, which make use of several CSP step laws, i.e., the tactic `cspF_Int_choice_left1` allows to make the following step $P_1 \sqsubseteq_{\mathcal{F}} Q \Rightarrow P_1 \sqcap P_2 \sqsubseteq_{\mathcal{F}} Q$. ■

We have now established that ACL_GETCONFIG is deadlock free. In Theorem 11.3.4 we have proven that the refinement ACL_GETCONFIG $\rightsquigarrow^{\sigma}_{\mathcal{F}}$ CCL_GETCONFIG holds. The stable failure model preserves deadlock freeness of processes (see Section 8.1). Therefore, we can conclude that CCL_GETCONFIG is deadlock free. Figure 11.11 illustrates the overall idea of deadlock analysis of the *get configuration* dialogue.

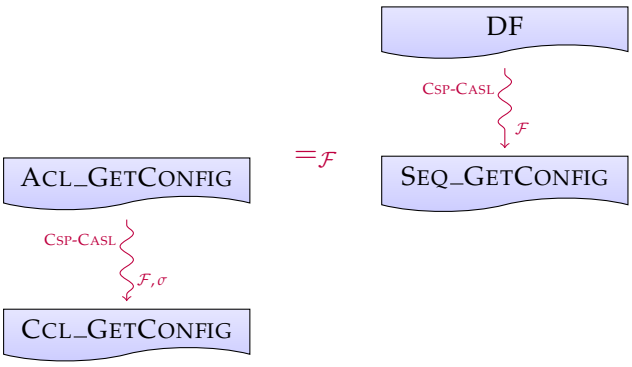

Figure 11.11: EP2 deadlock analysis in CSP-CASL.

We have carried out deadlock analysis of several other dialogues between EP2 components. Basically, these analysis follow the same kind of approach as presented for the *get configuration* dialogue.

11.3.3 Livelock Analysis

As described in Section 8.2 livelock freeness is best analysed in the failures/divergences model $\mathcal{N}$. The model $\mathcal{N}$ has not been implemented in CSP-CASL-PROVER. However, in the following, we prove using basic step and distributivity laws of CSP that the dialogue between the *Terminal* and the *ServiceCenter* is livelock free.

We first show that the sequential version of the *get configuration* dialogue is livelock free. To this end, we use Theorem 8.2.4 to prove that the refinement DIVF $\rightsquigarrow^{\sigma}_{\mathcal{N}}$ SEQ_GETCONFIG holds. Here, DIVF is the least refined livelock free process:

ccspec DIVF =
  data D_ACL_GETCONFIG
  process
    $DivF = (STOP \sqcap SKIP) \sqcap (\bigsqcap_{s:S} !x : s \rightarrow DivF)$

end

The sort S contains all the sorts declared in D_ACL_GETCONFIG. For simplicity, in the SEQ_GETCONFIG specification we consider only a process nucleus. That is, only one branch of the internal choice operator, i.e.,

$Seq\_Start = C\_SI\_Config\ !\ sesStart :: SessionStart \rightarrow SC\_Mgm$
$Seq\_Mgm = C\_SI\_Config\ !\ seM :: SessionEnd \rightarrow SC\_Config$
$\quad \sqcap\ C\_SI\_Config\ !\ cdrM :: ConfigDataRequest \rightarrow$
$\quad\quad C\_SI\_Config\ !\ response :: ConfigDataResponse \rightarrow Seq\_Mgm$

THEOREM 11.3.6 DIVF $\rightsquigarrow^{\sigma}_{\mathcal{N}}$ SEQ_GETCONFIG.

PROOF. This is basically a process refinement over the failures/divergence model, i.e., DIVF $\sqsubseteq_{\mathcal{N}}$ SEQ_GETCONFIG. This can be transformed into

$$\text{DIVF} =_{\mathcal{N}} \text{DIVF} \sqcap \text{SEQ\_GETCONFIG}$$

We now apply CSP step laws to prove the equivalence:

$$\begin{array}{ll} & DivF =_{\mathcal{N}} DivF \sqcap Seq\_GetConfig \\ (by\ symmetry) & | \\ & DivF \sqcap Seq\_GetConfig =_{\mathcal{N}} DivF \\ (unfolding) & | \end{array}$$

$$
\begin{array}{ll}
 & ((STOP \sqcap SKIP) \sqcap (\bigsqcap_{s:S} !x :: s \rightarrow DivF)) \\
 & \sqcap \\
 & Seq\_Start = C\_SI\_Config \,!\, sesStart :: SessionStart \rightarrow Seq\_Mgm \\
 & Seq\_Mgm = C\_SI\_Config \,!\, seM :: SessionEnd \rightarrow SKIP \\
 & \qquad \sqcap C\_SI\_Config \,!\, cdrM :: ConfigDataRequest \\
 & \qquad\quad \rightarrow C\_SI\_Config \,!\, response :: ConfigDataResponse \\
 & \qquad\quad \rightarrow Seq\_Mgm \\
(distributivity\ of\ \sqcap) & | \\
 & ((\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap SKIP) \sqcap ((\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap STOP)) \\
 & \sqcap \\
 & Seq\_Start = C\_SI\_Config \,!\, sesStart :: SessionStart \rightarrow Seq\_Mgm \\
 & Seq\_Mgm = C\_SI\_Config \,!\, seM :: SessionEnd \rightarrow SKIP \\
 & \qquad \sqcap C\_SI\_Config \,!\, cdrM :: ConfigDataRequest \\
 & \qquad\quad \rightarrow C\_SI\_Config \,!\, response :: ConfigDataResponse \\
 & \qquad\quad \rightarrow Seq\_Mgm \\
(\sqcap\text{-}step\ law) & | \\
 & ((\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap SKIP) \sqcap ((\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap STOP)) \\
 & \sqcap \\
 & Seq\_Start = C\_SI\_Config \,!\, sesStart :: SessionStart \rightarrow Seq\_Mgm \\
 & Seq\_Mgm = C\_SI\_Config \,!\, seM :: SessionEnd \rightarrow SKIP \\
(\sqcap\text{-}step\ law) & | \\
 & ((\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap STOP) \\
 & \sqcap \\
 & Seq\_Start = C\_SI\_Config \,!\, sesStart :: SessionStart \rightarrow Seq\_Mgm \\
 & Seq\_Mgm = C\_SI\_Config \,!\, seM :: SessionEnd \rightarrow SKIP \\
(\bigsqcap\text{-}rewriting) & | \\
 & ((\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap STOP) \\
 & \sqcap \\
 & \bigsqcap_{s:\{SessionStart,SessionEnd\}} !x :: s \rightarrow Seq\_Start \sqcap SKIP \\
(distributivity\ of\ \sqcap) & | \\
 & (\bigsqcap_{s:S} !x :: s \rightarrow DivF) \sqcap (STOP \sqcap SKIP) = DivF
\end{array}
$$

This proves that $DivF =_{\mathcal{N}} DivF \sqcap Seq\_GetConfig$. ■

Now that we have proved SEQ_GETCONFIG is livelock free, we proceed to verify that the actual dialogue between the *Terminal* and the *ServiceCenter* is livelock free. To this end, we use the property that the failures/divergences model preserves livelock freeness. We show that the sequential version is equivalent over the failures/divergences model to the actual dialogue. For simplicity, in the ACL_GETCONFIG specification we consider only a nucleus process nucleus. That is, one branch of the activity diagram presented in Figure 11.5, i.e.,

$$
\begin{array}{l}
TerminalConfiguration = Ter\_Config \;[\![\, C\_SI\_Config \,]\!]\; SC\_Config \\
Ter\_Config = C\_SI\_Config \,!\, sesStart :: SessionStart \rightarrow Ter\_Mgm \\
Ter\_Mgm = C\_SI\_Config \,?\, configMess :: D\_SI\_Config \rightarrow \\
\qquad \textbf{if}\ configMess \in D\_SI\_Config\_SessionEnd\ \textbf{then}\ SKIP
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad \textbf{else if } \mathit{configMess} \in \mathit{ConfigDataRequest} \\
&\qquad\qquad\qquad \textbf{then } \mathit{C\_SI\_Config} \mathbin{!} \mathit{resp} :: \mathit{ConfigDataResponse} \rightarrow \mathit{Ter\_Mgm} \\
&\qquad\qquad \textbf{else if } \mathit{configMess} \in \mathit{ConfigNotif} \\
&\qquad\qquad\qquad \textbf{then } \mathit{C\_SI\_Config} \mathbin{!} \mathit{ack} :: \mathit{ConfigAck} \rightarrow \mathit{Ter\_Mgm} \\
&\mathit{SC\_Config} = \mathit{C\_SI\_Config} \mathbin{?} \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \mathit{SC\_Mgm} \\
&\mathit{SC\_Mgm} = \mathit{C\_SI\_Config} \mathbin{!} \mathit{seM} :: \mathit{SessionEnd} \rightarrow \mathit{SC\_Config} \\
&\qquad\qquad \sqcap\ \mathit{C\_SI\_Config} \mathbin{!} \mathit{cdrM} :: \mathit{ConfigDataRequest} \rightarrow \\
&\qquad\qquad\quad \mathit{C\_SI\_Config} \mathbin{?} \mathit{response} :: \mathit{ConfigDataResponse} \rightarrow \mathit{SC\_Mgm}
\end{aligned}
$$

THEOREM 11.3.7 ACL_GETCONFIG $=_{\mathcal{N}}$ SEQ_GETCONFIG.

PROOF. We apply CSP step laws to prove the equivalence. We start from the lhs:

$$
\begin{aligned}
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \mathit{Ter\_Mgm} \\
& \qquad\qquad\qquad |[\, \mathit{C\_SI\_Config} \,]| \\
& \qquad \mathit{C\_SI\_Config} \mathbin{?} \mathit{sesStart} : \ \mathit{SessionStart} \rightarrow \mathit{SC\_Mgm} \\
(\| -\mathit{step}) \quad & | \\
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \\
& \qquad\quad (\mathit{Ter\_Mgm} \,|[\, \mathit{C\_SI\_Config} \,]|\, \mathit{SC\_Mgm}) \\
(\mathit{unfold}) \quad & | \\
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \\
& \qquad (\mathit{C\_SI\_Config} \mathbin{?} \mathit{configMess} \ \rightarrow \\
& \qquad\quad (\textbf{if } (\mathit{configMess\ in\ DataRequest}) \\
& \qquad\qquad \textbf{then } \mathit{C\_SI\_Config} \mathbin{!} \mathit{resp} : \mathit{ConfigDataResponse} \rightarrow \ \mathit{Ter\_Mgm} \\
& \qquad\quad \textbf{else if } (\mathit{configMess\ in\ SessionEnd}) \textbf{ then } \mathit{SKIP} \textbf{ else } \mathit{STOP}) \\
& \qquad\qquad\qquad\qquad |[\, \mathit{C\_SI\_Config} \,]| \\
& \qquad\quad \mathit{C\_SI\_Config} \mathbin{!} \mathit{seM} :: \mathit{SessionEnd} \rightarrow \mathit{SKIP} \\
& \qquad \sqcap\ \mathit{C\_SI\_Config} \mathbin{!} \mathit{cdrM} :: \mathit{ConfigDataRequest} \\
& \qquad\quad \rightarrow \mathit{C\_SI\_Config} \mathbin{!} \mathit{response} :: \mathit{ConfigDataResponse} \rightarrow \mathit{SC\_Mgm} \\
(\| -\mathit{distrib}) \quad & | \\
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \\
& \qquad\quad (\mathit{C\_SI\_Config} \mathbin{!} \mathit{seM} :: \mathit{SessionEnd} \rightarrow \mathit{SKIP} \\
& \qquad\qquad\qquad |[\, \mathit{C\_SI\_Config} \,]| \\
& \qquad (\mathit{C\_SI\_Config} \mathbin{?} \mathit{configMess} \ \rightarrow (\textbf{if } \ldots)) \\
& \qquad\qquad\qquad\qquad \sqcap \\
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{cdrM} :: \mathit{ConfigDataRequest} \\
& \qquad\quad \rightarrow \mathit{C\_SI\_Config} \mathbin{!} \mathit{response} :: \mathit{ConfigDataResponse} \rightarrow \mathit{SC\_Mgm} \\
& \qquad\qquad\qquad\qquad |[\, \mathit{C\_SI\_Config} \,]| \\
& \qquad (\mathit{C\_SI\_Config} \mathbin{?} \mathit{configMess} \ \rightarrow (\textbf{if } \ldots))) \\
(\mathit{if} - \mathit{step}) \quad & | \\
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \\
& \qquad\quad (\mathit{C\_SI\_Config} \mathbin{!} \mathit{seM} :: \mathit{SessionEnd} \rightarrow \mathit{SKIP}) \\
& \qquad\qquad\qquad\qquad \sqcap \\
& \qquad \mathit{C\_SI\_Config} \mathbin{!} \mathit{cdrM} :: \mathit{ConfigDataRequest} \\
& \qquad\quad \rightarrow \mathit{C\_SI\_Config} \mathbin{!} \mathit{response} :: \mathit{ConfigDataResponse} \rightarrow \mathit{SC\_Mgm}
\end{aligned}
$$

The last $\mathit{if} - \mathit{step}$ is applied twice, i.e., one per $\sqcap$ branch. By doing a process renaming and re-structuring, we obtain SEQ_CONFIG, i.e,

$$
\begin{aligned}
&\mathit{Seq\_Start} = \mathit{C\_SI\_Config} \,!\, \mathit{sesStart} :: \mathit{SessionStart} \rightarrow \mathit{SC\_Mgm} \\
&\mathit{Seq\_Mgm} = \mathit{C\_SI\_Config} \,!\, \mathit{seM} :: \mathit{SessionEnd} \rightarrow \mathit{SC\_Config} \\
&\qquad\qquad \sqcap\ \mathit{C\_SI\_Config} \,!\, \mathit{cdrM} :: \mathit{ConfigDataRequest} \rightarrow \\
&\qquad\qquad\quad \mathit{C\_SI\_Config} \,!\, \mathit{response} :: \mathit{ConfigDataResponse} \rightarrow \mathit{Seq\_Mgm}
\end{aligned}
$$

■

Similarly to the deadlock analysis, we have carried out livelock analysis of several other dialogues between EP2 components. This analysis follow the same kind of approach as presented for the *get configuration* dialogue.

11.4 Testing framework for EP2

In this section we describe how the testing process of EP2 has been carried out. Here, we show how we select the test cases to be executed. We evaluate the test cases using CSP-CASL-PROVER; for this, we use the syntactic encoding of colouring test cases presented in Section 9.3.

Moreover, we introduce the EP2 Testing Evaluator tool (TEV). This is an on-the-fly testing framework for an EP2 terminal. Such a tool will allow us to run test cases in a hardware-in-the-loop testing fashion. Here, we describe its architecture and its basic features.

11.4.1 Test case selection and evaluation

Selection of test cases is done using some general guidelines based on the informal EP2 specification and the CSP-CASL specifications. In selecting the test cases we adopt three general test purposes:

Main functionality of the EP2 components: Here, the purpose is to test the various functionality of the EP2 components prescribed in the various EP2 specification books. For instance, functionality like: configuration of the terminal by the service center; payment authorization performed by the acquirer; book keeping and logging of transaction performed by the POS management system, etc.

For such purpose, we select most of the test cases from the activity diagram of the various components. For instance, for the *get configuration* activity diagram (see Figure 11.4), we design at least one test case for each branch of the activity state.

Security features: EP2 uses the latest cryptographic techniques, which are very complex. Those techniques are built upon the combination of various cryptographic concepts such as cryptographic algorithms, mode, padding, hashing, *Message Authentication Code* (MAC) and Key exchange. The EP2 security specification prescribes three levels of security for exchanging messages between the components:

- Level 0: Has the lowest security with no cryptographic property. Messages at this level are sent in plain.
- Level 1: It enhances the security of the communication by introducing a MAC into the header of the message. This provides a message integrity check.
- Level 2: This is the highest security level. At this level, messages not only have a MAC for integrity check, but also the whole message is encrypted.

Depending on the message type the EP2 components are exchanging, one of the three level of security is applied. For instance, when an authorization for a payment transaction is sent from the terminal to the acquirer, level 2 is used. The acquirer replies to the terminal with a message which uses only level 1. In case of an error message, level 0 is used.

For such purpose, we select test cases that experiments the right level of security is used during the interactions.

Re-use of test case: Here, the purpose is to select test cases in order to illustrate our approach of re-using test cases in a vertical development as described in Chapter 10.

In the following we describe a sample of test cases that illustrate the overall approach of testing from CSP-CASL specifications. The first set of test cases are designed to experiment the EP2 system at the architectural level. As described in Section 11.2, the architectural specification of EP2, portraits the general overview of the system. Here, we design test cases which mainly will be used for setting up the test environment and the EP2 terminal system. In the following, we describe three CSP-CASL test cases:

$$T_0 = C\_SI\_Init!x :: D\_SI\_Init \rightarrow C\_SI\_Init!y :: D\_SI\_Init \rightarrow STOP$$
$$T_1 = x :: D\_BE\_BackEnd \rightarrow STOP$$

Here, x and y are variable over the indicated sorts.

T_0 : Experiments a secure communication between the acquirer and the terminal. Even with such simple test case, we can experiment various features described in the specification. On the functionality level, T_0 experiments the *bidirectional* communication between the terminal and the acquirer. Using such test, we can ensure a successful communication between the test environment and the SUT.

On the security level, here we can test for the authenticity of a particular acquirer. In fact the EP2 security specification prescribes that the service center is responsible of informing the terminal of the acquirer data. This information includes the ip and port address and the public key of the acquirer allowed to communicate with the terminal. The public key is then used to calculate the MAC data of a message. In the case of a fake acquirer the control for message integrity would fail. Therefore, this test case experiment a required behavior of the EP2 system; thus is colored *green*.

T_1 : Experiments a communication between the terminal and the PMS. Here, we don't specify the channel of the communication. With this test we want to test that com-

munication happens in designated channels. Therefore, this test case experiments a forbidden behavior of the system; hence is colored *red*.

We now illustrate a set of test cases designed to experiment the EP2 system at the abstract and concrete component level. The following are the CSP-CASL test cases:

$$
\begin{aligned}
T_2 &= C\_FE\_FrontEnd\ !\ authReq :: AuthRequest \\
&\quad \rightarrow C\_FE\_FrontEnd\ !\ authRes :: AuthResponse \rightarrow STOP \\
T_3 &= C\_SI\_Init\ !\ sesStart :: SessionStart \rightarrow C\_SI\_Init\ !\ notif :: DataNotif \\
&\quad \rightarrow C\_SI\_Init\ !\ sesEnd :: Session \rightarrow STOP \\
T_4 &= C\_SI\_Init\ !\ sesStart :: SessionStart \rightarrow C\_SI\_Init\ !\ notif :: RemoveNotif \\
&\quad \rightarrow C\_SI\_Init\ !\ notif :: RemoveAck \rightarrow C\_SI\_Init\ !\ actNotif :: ActivationNotif \\
&\quad \rightarrow C\_SI\_Init\ !\ actAck :: ActicationAck \rightarrow C\_SI\_Init\ !\ sesEnd :: Session \rightarrow STOP \\
T_5 &= C\_SI\_Config\ !\ sesStart :: SessionStart \rightarrow C\_SI\_Config\ !\ req :: ConfigDataRequest \\
&\quad \rightarrow C\_SI\_Config\ !\ res :: ConfigDataResponse \rightarrow C\_SI\_Config!sesEnd :: SessionEnd \\
&\quad \rightarrow STOP
\end{aligned}
$$

Here we give a brief explanation of:

T_2 Experiments a communication between the acquirer and the terminal in the context of *payment transaction*. Here, the terminal sends a message to the acquirer to authorize a payment for a purchased goods. The acquirer authorize the transaction by sending a message of type *AuthResponse*. This is a required behavior of the system, as specified in the EP2 terminal (and acquirer) book [Con08].

T_3 Experiments a communication between the acquirer and the terminal in the context of the *initialization* of the terminal data. Here, the acquirer informs the terminal what type of credit card are acceptable at this point. This is done by sending a *notification* message to the terminal. However the communication ends without an acknowledgment from the acquirer. This is a forbidden behavior, as all message exchanges needs to be acknowledged.

T_4 Experiments a communication between a acquirer and the terminal in the contest of *initialization* of the terminal data. The acquirer informs the terminal that a payment transaction using a particular type of cards (e.g., *Maestro* cards) are no longer acceptable. This is done by sending a message of type *remove config data* to the terminal. Then, by sending an activation message in order to make the initialization message active on the terminal. The data specified at the abstract component level, does not prescribe what kind of data the message *remove config data* has. At this point of the vertical development this decision is still left open.

T_5 Experiments a communication between a service center and the terminal in the contest of *get configuration data* use case. Here, we experiment the retrieval of some configuration data from the terminal. The service center sends a message of type *config data request* and is interested to retrieve the configuration data of a non existent acquirer. Again, at the level of the abstract component level, the details of the message *config data request* is left open. Only in the concrete component level will make this message concrete.

The color of the test cases $T_3, \ldots, T_6$ is as follows:

Spec. Level	T_2	T_3	T_4	T_5
ACL_EP2	GREEN	RED	YELLOW	YELLOW
CCL_EP2	GREEN	RED	GREEN	RED

11.4.2 Testing framework for EP2

Here, we describe the architecture of the testing framework for EP2– Testing EValuator (TEV). TEV is a hardware-in-a-loop on-the-fly testing framework, designed to test an EP2 terminal.

Hardware-in-a-loop testing (HIL) is a well established approach to validate complex systems, where the correct integration of software with its underlying hardware is essential. HIL has been deployed in defense and aerospace industry as early as the 1950s [NBAR04], nowadays it is an established testing technique. HIL is heavily used in verifying critical system in projects such as the power and thermal control unit of the X-ray satellite "ABRIXAS" [SMH99] and for cabin management controllers for Airbus families [Pel02] and more.

The general architecture of TEV was originally designed in [Chu05]. The designed architecture has been only tested with a simulator of EP2 messages, called CEPTEST EP2 developers test tool. The latter has been developed by CELSI AG[5], which is a member of the EP2 consortium. CEPTEST allows developers of any EP2 components to be able to create and send messages through any message oriented EP2 interface.

The author of this thesis have made significant changes to the architecture in order to be able to interface with the 'physical terminal'- *cCredit Terminal Software* provided by *Six Card Solutions*[6]. Here, we list the changes and new features implemented:

- Interface with the actual software of *cCredit Terminal Software*.
- Ability to test multiple EP2 components in a single run of a test case.
- Implementation of new security features as described in the EP2 standard *version* 4.0.
- Implementation of the test evaluation algorithm described in Section 9.4.

Figure 11.12 illustrate the hardware-in-the-loop testing framework for EP2.

As described in Section 11.1, the EP2 standard involves various technologies, such as cryptography, XML, and TCP/IP. The testing framework has to mirror such a complex system that involves different technologies. Hence, a well-structured testing architecture is required. Figure 11.13 illustrate TEV's architecture.

[5] http://www.celsi.ch/
[6] www.six-card-solutions.com

Figure 11.12: Hardware in the loop testing for EP2.

TEV is written in Java and uses several APIs and libraries: For cryptographic operation: *JCA* (Java Cryptography Architecture)[7] and *JCE* (Java Cryptography Extension)[8], for manipulation of XML documents: *SAX* (Simple API for XML)[9], *JDOM* (Java Document Object Model)[10], *XMLUnit*[11], *FOP*(Format Objects Processor)[12]; and other logging utility libraries such as *log4j*[13].

TEV is composed of a number of layers. Each layer provides a set of functions for the transmission and manipulation of the EP2 data. Here, we explain each layer in Figure 11.13:

- *Test Applications*: This layer is the basic interface of TEV. The tester interacts with the functionality provided in this layer in order to create, execute and get the verdict of test suites. It is composed by three stand-alone applications:
  - TEVCREATOR : Automatically generates a test case protocol in an XML format.
  - TEVMANAGER : It runs automatically the chosen test cases and evaluates on the fly the verdict of a test case.
  - TEVREPORTER : Automatically generates test verdicts in a readable format (ps, pdf, etc.)
- EP2 *Components*: This layer implements the various EP2 components which will be used by the test manager.

[7] http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html
[8] http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html
[9] http://www.saxproject.org/
[10] http://www.jdom.org/
[11] http://xmlunit.sourceforge.net/
[12] http://xmlgraphics.apache.org/fop/
[13] http://logging.apache.org/log4j

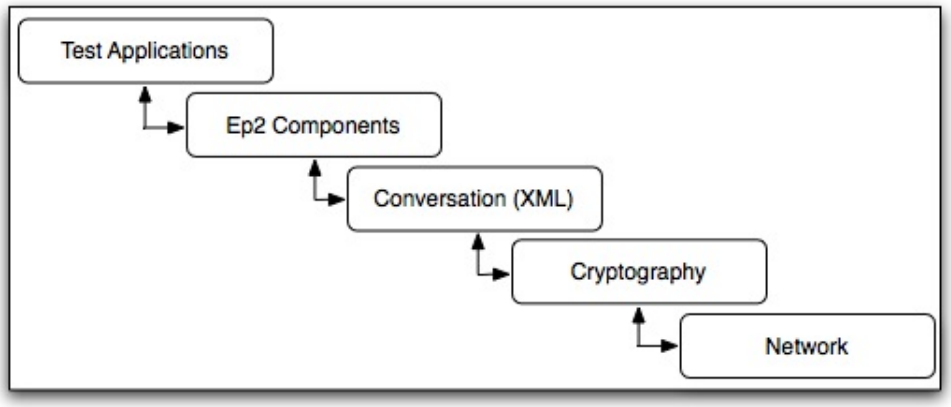

Figure 11.13: TEV – architecture [Chu05].

- *Conversation*: This layer deals with the manipulation of the EP2 XML messages. At this level most of the test verdict algorithm is implemented.
- *Cryptography*: This layer implements the necessary cryptographic operations for the encryption and decryption of EP2 data.
- *Network*: This layer deals with the network communication protocol between EP2 components.

Figure 11.4.2 illustrate a typical test case protocol of EP2. After the user inputs the necessary information, such file is automatically generated by the tool TEVCREATOR. The structure of the test case protocol consists of six parts:

- *meta* (line 3-7): contains basic information of the test case.
- *ccTest* (line 8-14): illustrates the CSP-CASL test process.
- *pcoInfo* (line 15-20): contains information about the PCO, the level of abstraction of the specification and the name of the CSP-CASL specification.
- *testcaseEvalInfo* (line 21-25): contains information about the test case evaluation, i.e., color of test case, location of the colouring proof and timeout information.
- *componentList* (line 26-50): contains information about the EP2 components which are involved in this specific test run. For instance in this test case, the *Aquirer* and the *Terminal* are involved. Here, for each component we specify: in which interface (channel) they communicate (e.g., *FEFrontEnd*), the level of security in which the interaction happen (e.g., Level 2), in which communication mode the component is communicating (e.g., *server*). Here, we can add as many EP2 components as we like to test.
- *testSequence* (line 51-72): contains the actual test sequence run. Here, we specify how the test environment interact with the SUT. Here, the *event* section of the test se-

quence represent one test run. Within each *event* we specify the *conversation* between the parties, i.e., *recieve* and *send*. Moreover, at this level we specify the expected message for each message sent from the SUT.

TEVMANAGER takes as input a test case protocol, execute the test case and compute the test verdict. The latter is then written in an XML file. TEVMANAGER is also responsible to simulate the EP2 component(s) which is interacting with the *cCredit Terminal Software*.

In the next section we describe the scenario of a typical test case execution of EP2.

11.4.3 Test case execution

The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test process. Here, we need to first establish a PCO. Here we consider a PCO, which connects test cases derived from the CSP-CASL abstract component level specification. For instance, let us consider the CSP-CASL test case T_2 described above. T_2 experiments the authorization of a payment transaction between the terminal and the acquirer. The following is the concrete XML message that the terminal send for the authorization:

XML message for authorization request

```
<?xml version="1.0" encoding="UTF-8"?>
<ep2:message xmlns:ep2="http://www.eftpos2000.ch" specversion="0400">
  <ep2:authreq msgnum="7222">
    <ep2:AcqID>00000000004</ep2:AcqID>
    <ep2:TrmID>TERM1234</ep2:TrmID>
    <ep2:TrxDate>20100223</ep2:TrxDate>
    <ep2:TrxTime>130842</ep2:TrxTime>
    <ep2:TrxSeqCnt>24551</ep2:TrxSeqCnt>
    <ep2:AmtAuth>50</ep2:AmtAuth>
    <ep2:TrxCurrC>756</ep2:TrxCurrC>
    <ep2:Track2Dat>OZYbmsV8ODZ3EC3vY4z9yA==</ep2:Track2Dat>
    <ep2:TVR>AAAAgAA=</ep2:TVR>
    <ep2:CVMRes>HgAA</ep2:CVMRes>
    <ep2:POSEntry>90</ep2:POSEntry>
    <ep2:TrxTypeExt>3</ep2:TrxTypeExt>
    <ep2:AID>oAAAAVcAIA==</ep2:AID>
  </ep2:authreq>
</ep2:message>
```

In order to establish the PCO we develop an equivalence relation which allows us to abstract some aspects of the primitive events, such as the message number (*msgnum*). In general, for each interface of communication between EP2 components (blue lines in Figure 11.1) we establish an equivalence relation of the XML messages exchanged, i.e., $\sim_{SI\_Config}, \sim_{SI\_Init}, \cdots, \sim_{FE\_FrontEnd}$.

As an example, we show a PCO in order to execute test case T_2:

Alphabet The alphabet of primitive events are all the possible XML messages that are communicated over the interface *FE_FrontEnd*, such as the XML message reported above. We denote the set of these XML messages as *XML_FE_FrontEnd*, e.g., $[ep2$:

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
  <meta>
    <result>resources/TestVerdict/PaymentTestResult.xml</result>
    <description>Test Case 7 - Authorization for payment transaction</description>
    <name>Process Transaction Test</name>
  </meta>
  <ccTest>
    <testCase number="T7">
     T7 =    C_FE_FrontEnd ! authreq::D_FE_FrontEnd_AuthReq
          -> C_FE_FrontEnd ! authresp::D_FE_FrontEnd_AuthRes
          -> STOP
    </testCase>
   </ccTest>
   <pcoInfo>
    <pcoFile>resources/PCO/PCO.xml</pcoFile>
    <SpecificationLevel>Abstract Component Level</SpecificationLevel>
    <SpecificationFile>ACL_ProcessTransaction</SpecificationFile>
    <Timeout>10ms</Timeout>
   </pcoInfo>
   <testcaseEvalInfo>
    <ep2Dialogue>ProcessTransaction</ep2Dialogue>
    <color>GREEN</color>
    <colorProofDir>resources/ColorProof/GREEN/T8</colorProofDir>
  </testcaseEvalInfo>
  <componentList>
    <component class="Acquirer">
      <namespace>http://www.eftpos2000.ch</namespace>
      <templatePath>resources/template/FEFrontEnd/</templatePath>
      <serverIp>192.168.1.1</serverIp>
      <encoding>1</encoding>
      <AcqID>00000000004</AcqID>
      <prefix>ep2</prefix>
      <interfaceName>FEFrontEnd</interfaceName>
      <serverPort>6625</serverPort>
      <name>Acquirer</name>
      <communicationMode>server</communicationMode>
    </component>
    <component class="Terminal">
      <namespace>http://www.eftpos2000.ch</namespace>
      <templatePath>resources/template/FEFrontEnd/</templatePath>
      <encoding>2</encoding>
      <TrmID>TERM1234</TrmID>
      <prefix>ep2</prefix>
      <interfaceName>FEFrontEnd</interfaceName>
      <port>6625</port>
      <ip>192.168.1.2</ip>
      <name>cCredit Terminal</name>
    </component>
   </componentList>
   <testSequence>
    <event source="Acquirer" target="cCredit Terminal">
     <conversation>
        <receive>
          <type>authreq</type>
          <AcqID>00000000004</AcqID>
          <TrmID>TERM1234</TrmID>
        </receive>
        <expected>
           <type>authreq</type>
           <filename>authreq.xml</filename>
        </expected>
       <send>
          <AcqID>00000000004</AcqID>
          <type>authrsp</type>
          <TrmID>TERM1234</TrmID>
          <filename>authrsp.xml</filename>
        </send>
        </conversation>
      </event>
   </testSequence>
</test>
```

authreq] $\in$ *XML_FE_FronEnd* where [*ep*2 : *authreq*] denote the XML message presented above.

Mapping In T_2 we have two events, those are mapped as follows:

$$\|[ep2 : authreq]\| = C\_FE\_FrontEnd.authReq$$
$$\|[ep : authres]\| = C\_FE\_FrontEnd.authRes$$

Direction For each linear test case we determine the direction of the particular test case. In the case of T_2 we have that, *C_FE_FrontEnd.authReq* is of type *sut2ts*. The acquirer is simulated by TEV, thus the direction of *C_FE_FrontEnd.authRes* is of type *ts2sut*.

The test execution is conducted in a network environment, where two different machines are connected with a crossed Ethernet cable. Figure 11.15 illustrate this setting. Here, the EP2 terminal software is running on the black laptop (Windows OS), where a *Pinpad* is attached on the serial port. TEV is running on the white laptop (Mac OS).

In order to perform experiments regarding the payment transaction (test cases like T_2), we use magnetic stripe test cards of different financial institutes: *Visa*, *MasterCard*, *Maestro*[14] etc.

Figure 11.15: EP2 testing framework in action.

For the execution of the test cases, we define 10ms as the period of time in which we expect a response from the EP2 payment terminal. Executing the test cases $T_0, \ldots, T_6$ results in the following test verdict:

[14]Courtesy of *Six Card Solutions*.

Spec. Level	T_0	T_1	T_2	T_3	T_4	T_5
ARCH_EP2	PASS	PASS	-	-	-	-
ACL_EP2	-	-	PASS	PASS	Not executed	Not executed
CCL_EP2	-	-	PASS	PASS	PASS	PASS

The automatically generated test verdict for test case T_3 can be found in the Appendix C.6.

11.5 Summary and evaluation of the project

In this chapter we have described the modelling, verification and testing of the electronic payment system EP2. A first modeling approach of the different levels of EP2 in CSP-CASL has been described in [GRS05]. Here, we have extended the modeling in more detail by carrying out the specification of the various EP2 components at different levels of abstraction. We have systematically proven the refinement steps of the various level of specification using CSP-CASL-PROVER [OIR09]. Moreover, we have proven that the interaction of the EP2 components is deadlock free. Again this is done systematically using CSP-CASL-PROVER. For some selected interaction of EP2 components we have proven to be livelock free.

For the testing part, we have evaluated test cases using CSP-CASL-PROVER. Moreover, we have presented a testing framework for a EP2 payment terminal. Such testing framework, tests the EP2 payment terminal in a hardware-in-the-loop testing fashion.

The overall experience of modeling, verification and testing EP2 in CSP-CASL was positive. In the following we evaluate how the three activity have been conducted:

Modeling of EP2 Formalizing EP2 in CSP-CASL leads to the partial or complete resolution of some of the ambiguity or inconsistency problems, described in [GRS05]. For instance, in the EP2 specification inconsistency arises from the fact that certain aspects of the system are described in multiple books. For example, in order to understand how the payment transaction works, one has to look at the terminal and the acquirer books for the functionality and then the interface and data dictionary books in order to get the specific messages exchanged at this point. CSP-CASL enables us to specify the data involved only one and – via CSP-CASL's library mechanism – use it then in different contexts.

Identifying the three level of abstraction was rather straightforward; this was already done in [GRS05]. The modeling of the architectural and the abstract component level was relatively easy. Although, at the abstract component level, we had to take decisions of which information to formalized at this point or leave it open for the next level of abstraction. For this two levels we have pretty much followed the semi-formal description present in the EP2 specifications. Those are, the activity diagram, UML-like diagrams and use cases.

At the concrete component level, one has to deal with more unresolved and unclear

descriptions (mostly presented as text), and decide which information must be formalized and what details should be ignored at this point.

As for CSP-CASL's expressive power in formalizing EP2, both the data types and the reactive behavior present in EP2 was adequately formalized. In modeling the data types, CASL's subsorting feature and the CASL standard library were quite helpful. In modelling the process part the different operators of CSP allowed us to capture precisely certain aspects of the system. For example, at the architectural level, the interleaving and the generalized parallel operator allowed us to capture the fact that some EP2 components interact with each other or they simply run independently.

Verification of EP2 The benefit of specifying EP2 in CSP-CASL is that it makes it possible to establish properties by formal proofs. Here, we have shown formal proofs of refinement, deadlock and livelock analysis.

The refinement proof from the architectural level to the abstract component level was quite straightforward. This is thanks to the introduction of intermediate specification, i.e., the sequential version of the dialogues. The overall proof script for discharging the refinement proof is fairly long, this is due to the fact that at some point of the proof we needed to calculate the semantics of the processes. Conversely, as expected, the refinement verification from the abstract component level to the concrete component level was more complicated and the proof script was rather long.

The verification of deadlock freedom has been done by considering each individual dialogue between the EP2 components, i.e., the communication between terminal and acquirer, etc. Although this give us a good insight in the verification of deadlock freeness, still deadlock could occur when we consider the whole system. This could be analaysed using new theories for compositional reasoning of reactive systems.

Testing of EP2 In testing EP2, we could clearly see the impact of our testing theory of re-using test cases in a vertical development. Test cases developed from the architectural level allowed us to set up the interface with the testing framework. Later, at each stage of the vertical development we were able to re-use the green and red test cases.

The presented testing framework was capable of running test cases in an efficient and convincing way. However, there is still several aspects that haven't been undertaken. Firstly, although in principle our testing framework presents several benefits, a 'formal' comparison between our approach and the current testing practice at *SIX Card Solutions* is needed. Secondly, since payment systems like EP2 rely heavily on cryptographic based security features; here there is a need of a novel strategy to test such features. Finally, only a sample of test cases have been executed on the SUT. In order to increase our confidence that the SUT is conform with the specification we need to run more meaningful test cases. The latter could be achieved by studying new concepts of test converges from CSP-CASL specifications.

Overall this industrial application, demonstrates the feasibility of modeling, verification and testing in CSP-CASL. This includes many aspects:

- *Scalability*: It is possible to completely model a non trivial system like EP2.
- *Expressiveness*: CSP-CASL is expressive enough to capture the different aspect of such system. The reactive behavior and the complex data types.
- *Clarity*: CSP-CASL is able to mirror the informal specification and to capture the different level of abstraction. The refinement between the different levels are then proven using CSP-CASL-PROVER.
- *Verification and Testing*: Having a formal specification made it possible to prove interesting properties of the system.

(CHAPTER ...12)

ROLLS-ROYCE BR725 starting system

Contents

IN this chapter we report on a successful applications of the theoretical framework of testing based on CSP-CASL presented in Chapter 9. In particular we apply this theory to the starting system of a ROLLS-ROYCE BR725 [1] software control. The BR725 is a newly designed jet engine for ultra-long-range and high-speed business jets. We model the starting system in CSP; specifically we use CSP-M – the machine readable version of CSP. Here, we have chosen CSP-M instead of CSP-CASL due to the nature of the system specification we formalize. These specifications do not require loosely specified data or complex data structures. CSP-M is able to capture the data type that we require. Moreover, using CSP-M we can use the FDR2– model checker for CSP and PROBE– simulator of CSP specifications.

We validate our model using the CSP simulator PROBE. We then evaluate the test suites against the formal model. Such evaluation is done using the model checker FDR2. Part of the test suites is inspired by existing test cases of the BR700 family jet engines. We execute our test suite in an in-the-loop setting on the so-called "rig". This puts the engine control system through test scenarios identical to those carried out in engine test stand testing, however with considerably lower cost, reduced risk, and less burden on human and mechanical resources.

We first describe a control system of a jet engine in general, continuing to concentrate on the starting system of the BR725 jet engine. In Section 12.2 we show how we model the BR725 starting system in CSP. In Section 12.3 we illustrate the verification of interesting

[1] In the rest of the chapter we will refer to this engine type simply as BR725.

properties of the modeled system. Subsequently, in Section 12.4 we show how we evaluate and execute test cases. In Section 12.5 we conclude this chapter by evaluating the advantages and disadvantages of this experience.

The results presented in this chapter have been published in [KHRS09].

12.1 Introducing the ROLLS-ROYCE BR725 starting system

Formal methods offer many advantages for the development and testing of control software for jet engines. They can help to achieve high reliability, and they can be used to provide evidence of reliability claims which can then be subjected to external scrutiny.

Jet engines belong to the safety critical systems of an airplane. Their control software can be classified as a reactive system: it accepts commands from the pilot, receives status messages from the airframe and the engine sensors, and issues commands to the engine. ROLLS-ROYCE is one of the leading companies in the productions of jet engines of different sizes and for different purposes. Here we study the ROLLS-ROYCE BR725 jet engine. The BR725 is a new jet engine for ultra-long-range and high-speed business jets. It is part of the BR700 family. Figure 12.1 shows the BR725 jet engine and its electronic engine controller.

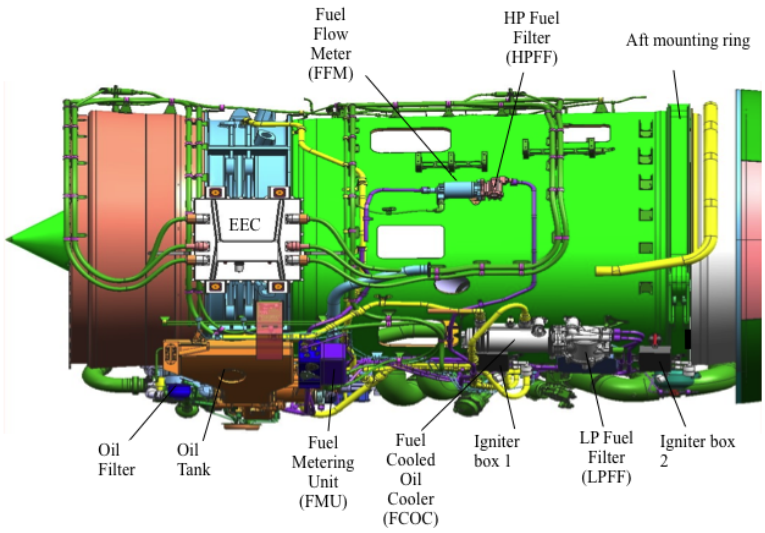

Figure 12.1: ROLLS-ROYCE BR725 jet engine – Courtesy of ROLLS-ROYCE.

The main component of the control system of a jet engine is the *Electronic Engine Controller* (EEC). A simplified view of the BR725 EEC architecture and its interfaces with the engine is shown in Figure 12.1. The EEC encapsulates all signaling aspects of the engine; it controls, protects, and monitors the engine. In order to provide fault-tolerance, the EEC is realised as a dual-channel system. Its control loop involves: reading data from sensors and other computer systems in the aircraft, receiving commands from the pilot, calculating new positions of the engine actuators, and issuing commands to the engine actuators.

In its monitoring function it transmits data about the engine condition and information on any failures diagnosed on the electronics back to the aircraft. In this paper we concentrate on the *Starting System*, one of the many functionalities the EEC provides.

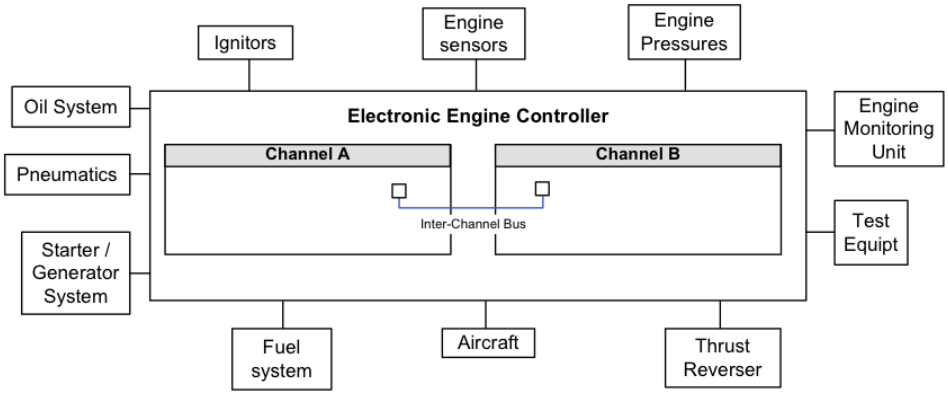

Figure 12.2: Electronic Engine Controller Architecture – Courtesy of ROLLS-ROYCE.

12.1.1 Control system at ROLLS-ROYCE

The control system team at ROLLS-ROYCE is responsible for designing, developing, verifying, testing and certifying the hardware and software components of the EEC. A typical team for each type of jet engine is composed by the following teams:

- *System design*: responsible for the design, development and certification of the hardware part.
- *Software development*: responsible for the design, development and certification of the software part.
- *System verification*: responsible for overall system (hardware + software) testing, i.e., black-box testing.

The author was part of the system verification team for the BR725 controls systems. The team is responsible for carrying out system level testing. Here, a black-box view is taken of the whole system.

The software developed by ROLLS-ROYCE executes on an *Electronic Engine Controller* (EEC) which is part of an engine control system comprising many electronic and mechanical components; this system is a part of an engine which is, in turn, a component of an aircraft.

A *Real-Time Engine Model* (RTEM) simulates the behaviour of the engine and is used to test prototype software builds in a PC-based environment. Later in the development lifecycle the implemented code is executed on the EEC in a Hardware-In-The-Loop (HIL) test environment. The process continues through a chain of "test vehicles", with each vehicle

providing an environment which is ever closer to reality. Examples of vehicles are hydromechanical rigs which incorporate sensors and actuators with real fluids (oil, air and fuel), real engine tests (on-ground and in altitude test facilities) and "fying test beds".

12.2 Modelling BR725 starting system in CSP

The jet engine BR725 can be started in both, on-ground and in-flight situations. Furthermore, the pilot can select between *Automatic* or *Manual* starting mode.

The detected situation (on-ground or in-flight) together with the selected starting mode (automatic or manual) results in four different control flows in which the EEC controls the engine. Cranking adds two further flows of the EEC, namely *dry* and *wet* cranking (i.e., without and with fuel on respectively). Dry cranking is usually used to remove any residual fuel in the combustor or jet pipe that remains from a previous failed start. Wet cranking is used to push the inhibiting fluid through the fuel system until sufficient fuel can be metered to the combustor for ignition. The functionalities provided for the starting system are the following:

Normal (automatic) Ground Start: provides an automatic start mode available on ground. The start sequencing is fully controlled by the EEC after being initiated by the pilot.

Manual Ground Start: provides the pilot to start the engine manually on ground. We will study this function in detail in the rest of the section.

Normal (automatic) Flight Start: this function is provided in order to reduce the pilot workload in-flight.

Manual Flight Start: this function enables the pilot to start the engine manually in-flight.

Cranking: this function provides a way to test the rotation of the windmill when the jet engine is on ground and is not ignited.

For automatic on-ground starts, a start sequence may include a maximum of two start attempts. A start sequence in-flight has as many attempts as necessary to successfully start the engine. There are three essential steps during a normal (i.e., anomaly-free) on-ground start sequence, which commences when both the fuel and start switch are in the "On" position:

- The starting system commands the starter motor on. The motor, mechanically coupled to an engine shaft, starts to rotate the engine.
- When the shaft has reached a sufficient rotational speed, and FOC (*Fuel-On Conditions*) are met, the starting system commands the fuel SOV (*Shut-Off Valve*) open, allowing fuel to flow to the combustion chamber. At this point the starting system also commands ignition on and, if there are no anomalies, the engine lights up.
- When the rotational speed of the engine reaches a threshold the starting system detects that the start is complete and commands both the starter motor and ignition

off.

Figure 12.2 shows an abstraction of the basic system architecture of a ROLLS-ROYCE jet engine starting system. The main signals transmitted between the components are as follows. In the Cockpit the pilot has a start switch in order to initiate the starting sequence; the pilot also has a fuel switch. The Airframe informs the EEC if the plane is in-flight or on-ground. The EEC can switch the Starter On and Off, can Open and Close the fuel SOV (*Shut-Off Valve*), and can turn the Ignition On and Off. The engine reports back to the EEC information about the shaft speed and the TGT (*Turbine Gas Temperature*). For BR725 the

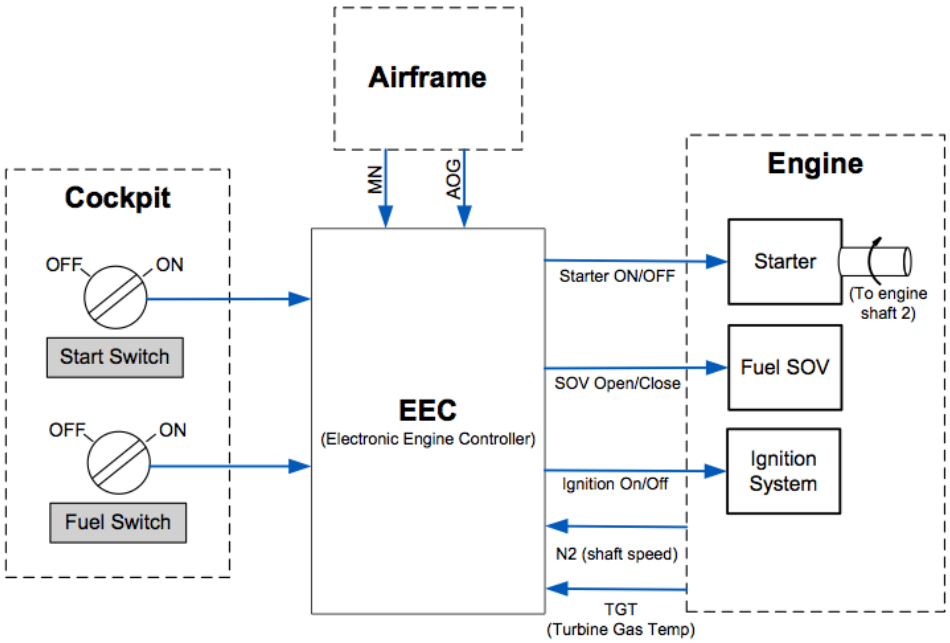

Figure 12.3: Starting system component architecture.

ROLLS-ROYCE *Starting Subsystem Definition Document* (SSDD) makes the Starting System within the EEC specific to this engine. This document describes all aspects of the Starting System: it gives an overview of the Starting System in general, it presents so-called activity diagrams and explains them in plain English. In the following we give an example of such an activity diagram and its accompanying text.

Figure 12.4 shows the internal logic of the manual ground start in the form of an activity diagram. These activity diagrams are formulated in an informal, graphical specification language. This language was specifically developed by ROLLS-ROYCE in order to describe engine controllers. An example of an accompanying text would be the following:

I3005/1 *When NH reaches the required speed, the pilot switches the Fuel Control to run.*

Note that every specification line is identified with a unique number, in our example with *I*3005/1. ROLLS-ROYCE makes use of this during the testing process as a coverage criterium: it is required that there is at least one test case for every line in the specification.

The graphical specification language of the activity diagram shown in Figure 12.4 uses symbols in the following way:

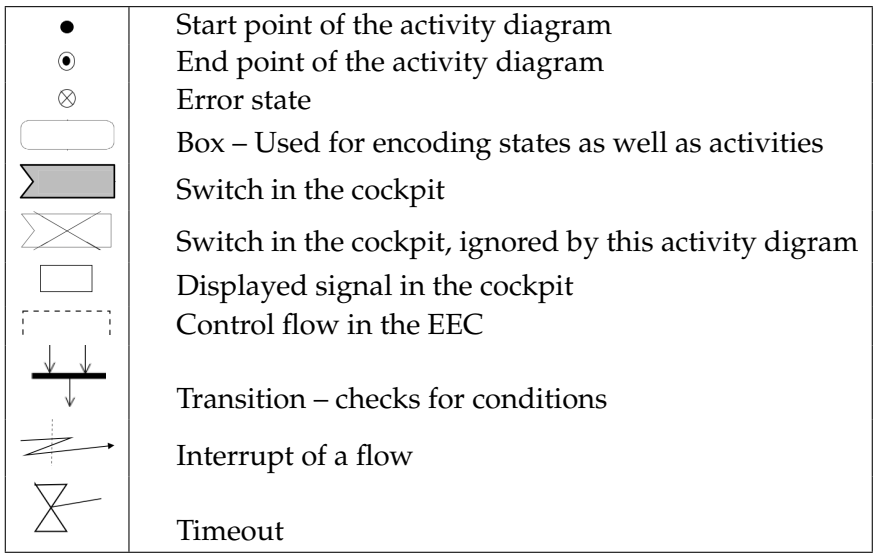

Figure 12.4 shows the Manual Ground Start (MGS) functionality which allows the pilot to start the engine manually. The flow from the start point to the second transition shows that the MGS can only be initiated when the aircraft is on the ground and the engine is not running, starting or cranking. In this situation the pilot can initiate the MGS by selecting the following control switches: *Master Crank* to `On`, *Continuous Ignitions* to `On`, and *Engine Start* to `On`.

Upon switching the *Engine Start* to `On`, the EEC will command the Starter Air Valve (*SAV*) to be opened and the starter motor is activated. If the pilot now switches the *Fuel Control Switch* to `Run` the EEC commands the fuel to flow. If at this point the *Continuous Ignitions* is still `On` the EEC ignites the motor (not shown in the Figure12.4) and begins to monitor the shaft speed of the engine. Should this speed reach a certain threshold the starting procedure is complete. While the starting procedure is active within the EEC, the pilot can abort it by switching the *Master Crank* or the *Fuel Control* to `Off`. If the pilot switches the *Continuous Ignitions* to `Off` the starting procedure ends in an error state.

12.2.1 Modeling of the starting system in CSP-M

Our main objective is to capture in a faithful way the original specification of the Starting System. To this end, we model the system in a way that a natural mapping can be drawn between the original specification (SSDD activity diagrams) and the CSP-M model. In the following we describe some aspects of the modeling in CSP-M of the manual on-ground starting procedure.

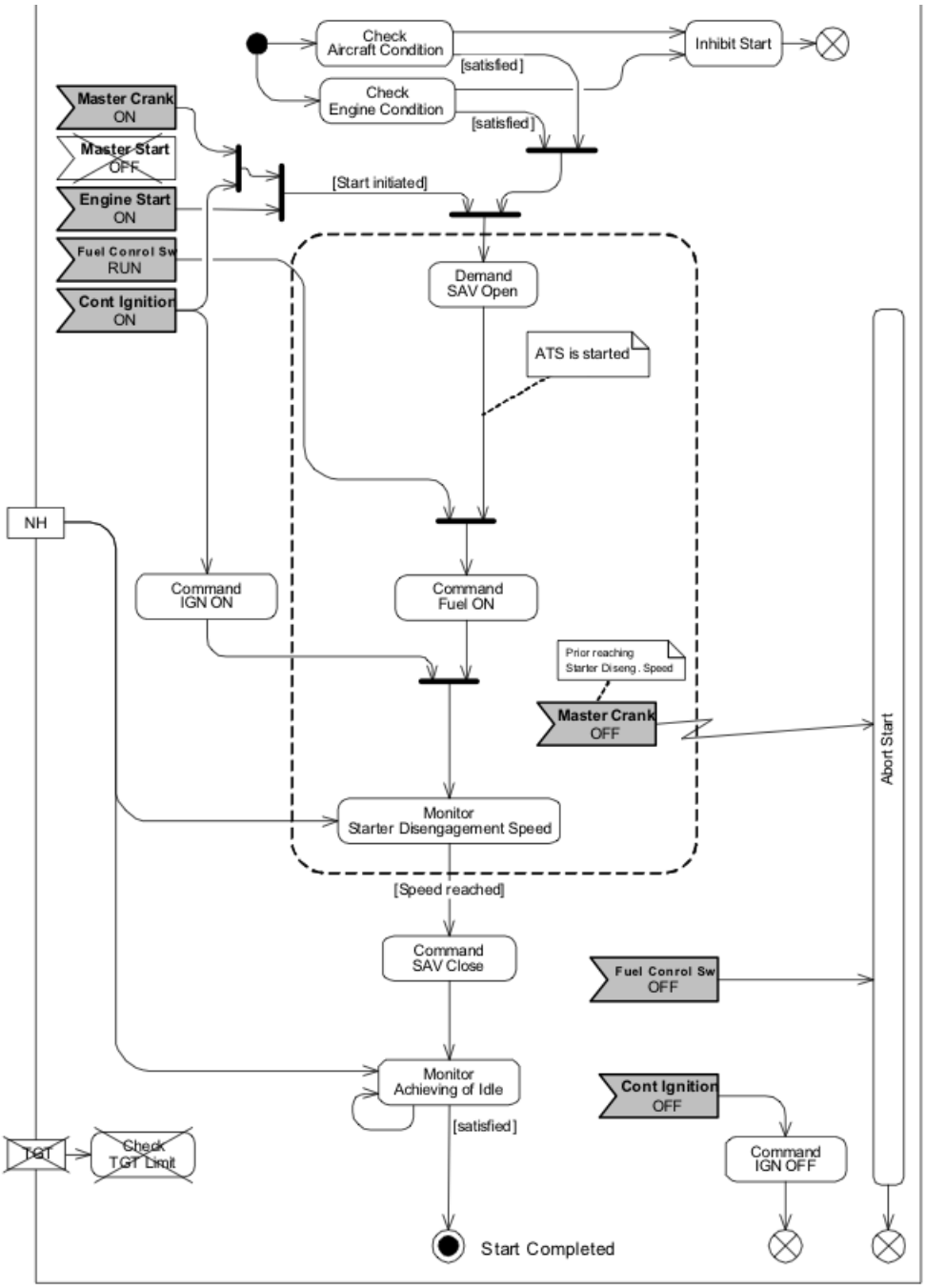

Figure 12.4: Activity diagram for manual ground start.

We have modeled the system in a two step approach: first we formalize the 'normal' execution pattern of the system. Only in a second step we add the handling of error cases such as interrupting the start by switching *Continuous Ignitions* to `Off`. CSP-M supports such a compositional approach of modelling via its interrupt operator.

In the following we describe some of the patterns which we have identified in the modeling process. We first describe some patterns and discuss then how their combination results in the overall control-flow.

Switch Buttons have two states: `ON` and `OFF`. Pressing a button in state `OFF` will turn it `ON`, releasing a button in state `ON` will turn it `OFF`.

```
channel press, release
ButtonOFF = press -> ButtonON
ButtonON = release -> ButtonOFF
```

We instantiate `ButtonON` and `ButtonOFF` to form the different switch buttons available in the Cockpit for the Starting System. This is done by simply using the CSP renaming operator, e.g., for `MasterCrank`:

```
channel mc_press, mc_release
MasterCrank = ButtonOFF[press <- mc_press
                        release <- mc_release]
```

All button processes run in an interleaved way. This corresponds to arbitrary press / release operations in the Cockpit. Note how this specification covers in an obvious way part of the activity diagram in Figure 12.4.

```
Buttons =  MasterCrank ||| MasterStart
        ||| EngineStartON ||| FuelControl ||| ContIgnition
```

Here, we have leave out the code of the `EngineStartON` which as a push button has no state.

Active waiting The starting sequence can only proceed when the following events happens: (1) the checks for *Aircraft* and *Engine* condition has been successful, (2) the pilot has issued the necessary starting commands. This is captured in the CSP-M model in the following way:

```
InteractEEC=(CheckConditions [|{synchStart}|] CrankAndIgnite)
              \ {synchStart}
            ; FuelAndSAV
            ; MasterIdle
```

where `CheckConditions` is the process that checks for the *Aircraft* and *Engine* condition.

```
channel aircraftCondition:Bool
channel engineCondition:Bool
channel inhibitStart, startOK
CheckConditions = aircraftCondition ? ac
                  -> engineCondition ? ec -> Checking(ac,ec)
                [] engineCondition ? ec
                  -> aircraftCondition ? ac -> Checking(ac,ec)
Checking(ac,ec) = if (ac and ec)
                       then startOK -> SKIP
                  else InhibitStart

InhibitStart = inhibitStart -> Idle
```

`CrankAndIgnite` is the process that handles the input of the pilot; namely the cranking and the ignition commands. `CheckConditions` and `CrankAndIgnite` synchronize on the `synchStart` event, only when the synchronization is successful the process `FuelandSAV` takes over. The `FuelandSAV` and `MasterIdle` processes capture the rest of the starting sequence.

```
datatype SAVMode = open | close
channel sav:SAVMode
channel commandFuelON, commandIgnON, started
SPEED1 = 15      SPEED2 = 65
channel readNH:{0..100}

FuelandSAV =  sav.open -> fc_press -> Fuel
          [] fc_press -> sav.open -> Fuel

Fuel =  commandFuelON -> commandIgnON  -> MasterSpeed
    []  commandIgnON  -> commandFuelON -> MasterSpeed

MasterSpeed = readNH ? x -> (if (x>SPEED1)
                                then SpeedReached
                             else MasterSpeed)

SpeedReached = sav.close -> SKIP

MasterIdle = readNH ? x -> (if (x>SPEED2)
                                then StartCompleted
                            else MasterIdle)

StartCompleted = started -> Idle
```

*SPEED*1 and *SPEED*2 are the percentage threshold of the overall speed to be reached.

Interleaving of decisions At different points of the activity diagram there are decisions that happen in an interleaved way. We model this scenario using the external choice operator in the following way:

```
CrankAndIgnite =    mc_press -> ci_press -> InitStartOK
                 [] ci_press -> mc_press -> InitStartOK
```

Here, in order to have state names available, we choose the semantically equivalent encoding of interleaving in terms of external choice and action prefix. The process `CrankAndIgnite` offers the to first switch `MasterCrank` or `ContIgnition`.

The whole manual on-ground start sequence is represented by the process `MGS`. The processes `InteractEEC` and `Buttons` runs in an alphabetized parallel. This corresponds to the interaction of the pilot (trough the Cockpit switches) and the EEC.

```
MGS_Core = InteractEEC || Buttons
```

In the second step of the modeling process we have identified and handled the error cases. Observing the activity diagrams, at different points of time during the starting sequence the pilot has the ability to abort the sequence by releasing the `FuelControl` or

the `ContIgnition`. We model this procedure using the CSP interrupt operator - $P \wedge Q$ – the progress of the process P can be interrupted on occurrence of the first event of Q.

```
channel commandIgnOFF, abort, error
InterruptFC = fc_release -> abort -> SKIP
InterruptCI = ci_release -> commandIgnOFF ->  Error

MGS_ErrorHandling = MGS_Core  /\  (InterruptFC [] InterruptCI)
```

The abortion due to releasing of the `MasterCrank` is only available after the *Command Fuel ON* event has occurred and prior reaching the *Starter Disengagement Speed*. In a pictorial way in Figure 12.4, this is identified by the dotted box.

In Appendix B.3 we report the full CSP-M specification for the *normal (automatic) ground start*.

12.2.2 Shortcomings

ROLLS-ROYCE uses activity diagrams such as shown in Figure 12.4 as memos. The engineers share common knowledge on jet engines, the activity diagrams merely trigger ideas how the control software works. Here, we list some of the shortcomings that we encountered during the modeling and reading process of the SSDD specification document.

- Although the *Engine Start* is a momentary button and *Master Crank* is a push button with two states both are shown with the same symbol in the activity diagram. That the *Engine Start* is a momentary button becomes clear from the textual description of the activity diagram. This explains also why there is no interrupt related to this button.
- We identically modeled both boxes which monitor speed (*Monitor Starter Disengagement Speed, Monitor Achieving of Idle*) relatively to the signal *NH*, while in the activity diagram the first box has no self-loop and the second box has a self-loop.
- Although the commands *IGN ON* and *IGN OFF* appear at first sight to be related, they are not: the command *IGN ON* is given by the pilot in the cockpit while the command *IGN OFF* is sent by the EEC to the engine. Therefore, we model these commands as two different events.
- As there is a command *FUEL ON* one would expect command *FUEL OFF* to appear in the activity diagram, e.g., when aborting the start. However, this is not the case.

12.2.3 CSP-M vs CSP-CASL

As described in Chapter 2, CSP-M is the machine readable version of CSP. Data in CSP-M are defined using a purely functional language with a strong type system , requiring

explicit type declarations for channels and data types. In CSP-M it is also possible to define higher order functions.

In CSP-CASL data are specified in CASL. Here, data are specified in an algebraic way. CASL has different type of semantics, where loose semantics is the standard one.

Confronting the two approaches: On one side we have data specified in CASL that allows us to specify data in a loose way, therefore suitable for step-wise development. On the other side we have the data specification in CSP-M where loose specification of data is not possible, however we are allowed to use all the features present in a typical functional language.

In this project we choose CSP-M instead of CSP-CASL due to the nature of the system specifications we formalize. These specifications do not require loosely specified data or complex data structures. Instead, they use simple data types only; mostly they speak about booleans and finite subranges of numbers. Data types used in modeling BR725, are available in the type system of CSP-M, and they can also be modeled within CASL (Figure 12.5). The following table illustrate, data specification in CSP-M and the corresponding version in CASL:

CSP-M	CASL
channel press	**free type** *press* ::= *press*
datatype savMode = *open* | *close*	**free type** *savMode* ::= *open* | *close*
channel sav : *savMode*	**free type** *sav* ::= *sav*(*s* : *savMode*)

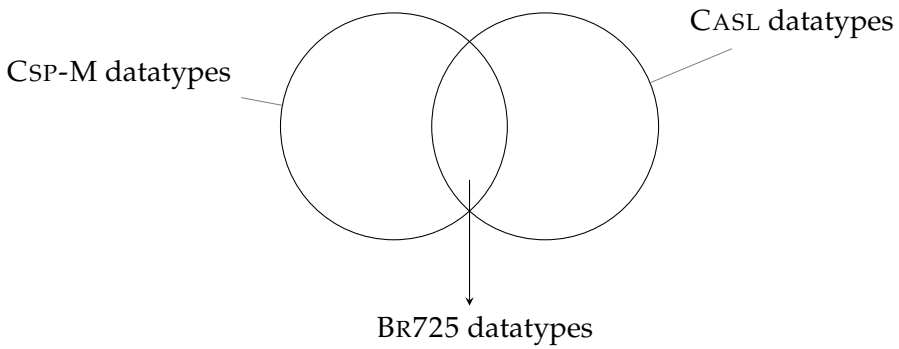

Figure 12.5: CSP-M data types for BR725.

12.3 Property verification of BR725 starting system

We have verified that our model is deadlock and livelock free. This is done using the model checker FDR2. In Figure 12.6 we illustrate a screenshot of FDR2 performing the verification of our model. Furthermore, we have verified that our model is deterministic. The screenshot presents FDR2 after successfully proved the three properties. This

is illustrated by the green tick (✓) symbol next to the main CSP-M process definition (*ManualGroundStart*). For the different property verification we choose the appropriate CSP model in which to perform the verification: for *deadlock* analysis the model $\mathcal{F}$, for livelock and determinism analysis the model $\mathcal{N}$.

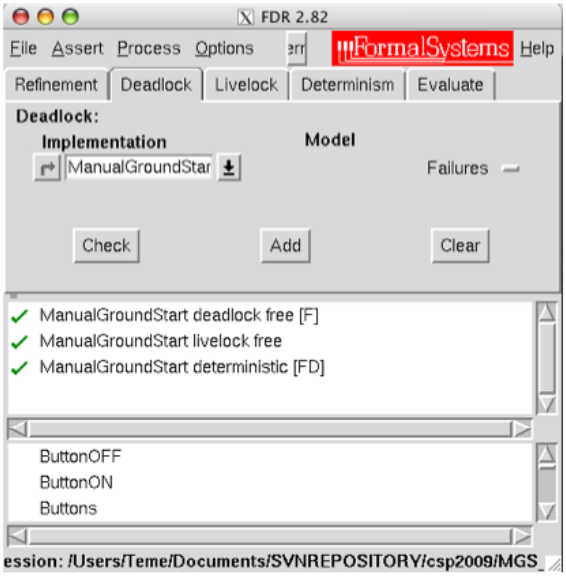

Figure 12.6: Screenshot of FDR2 for the verification of our model.

Simulations with the CSP animator PROBE, discussions with the ROLLS-ROYCE verification team, and – last but not least – the structural correspondence with the activity diagram validate our formal model. Figure 12.7 shows a screenshot of PROBE simulating the CSP specification of the manual on-ground starting functionality.

The screenshot shows a simulation of the manual on-ground starting CSP-M model. Here, the user acts as the environment and chooses the different events possible at certain instance of the process definition. In the figure, for instance as a first event we choose *aircraftCondition.true*, and then the event *engineCondition.true* followed by *ci_press*.

12.4 Testing BR725 starting system

In this section we describe the evaluation of test cases, how we establish the PCO and execute the test cases in the hardware-in-the-loop rig.

12.4.1 Test case evaluation

We use the syntactic encoding, presented in Section 9.3 to check the colour of a CSP-M test case T w.r.t. a CSP-M specification P. In the following we show how the test case

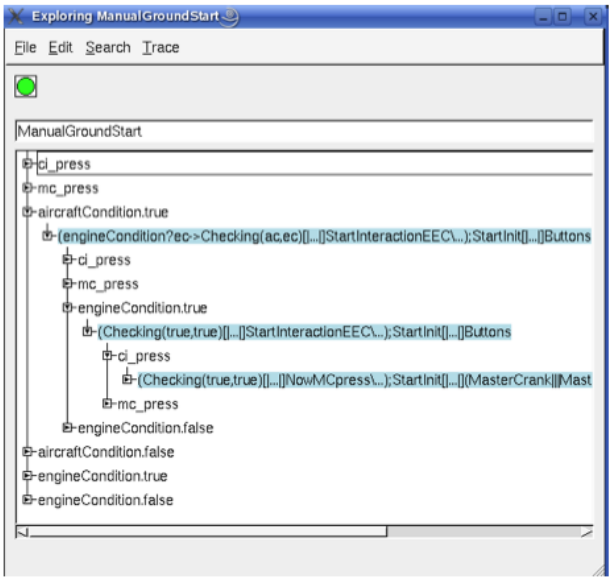

Figure 12.7: Screenshot of PROBE for the simulation of our model.

*TC*1 is coloured green. *TC*1 experiment a successful on-ground manual start. Figure 12.8 illustrates the input test script to check the trace condition. Here, `CheckT` encode the trace condition. Equality checking in FDR2 is done by refinement checking on both sides, i.e., $\sqsubseteq_{\mathcal{T}} \wedge \sqsupseteq_{\mathcal{T}} \Rightarrow =_{\mathcal{T}}$.

```
TC1 = aircraftCondition.true -> engineCondition.true -> MC_press -> CI_press
     -> engineStartON -> sav.open -> FC_press -> commandFuelON -> commandIgnON
      -> readNH.17 -> sav.close -> readNH.68 -> started -> STOP


channel OK, a
GREEN = OK -> STOP


CheckT = (  ( ( MGS [ {|aircraftCondition.true,...,started|}   |]  TC1   % Parallel
              ) [[aircraftCondition.true <- a,...,started <-a]]         % Renaming
            ) [| {a} |]  a -> ... -> a -> OK -> STOP                    % Parallel
        ) \ { obs }                                                     % Hiding


assert CheckT [T= GREEN                assert GREEN [T= CheckT     % Assert Check_T
```

Figure 12.8: FDR2 test script to check *Green* test case.

Since our model is deterministic and uses monomorphic data specifications, there is no need to check for the failures condition, w.r.t, green coloring of test cases.

For red test cases we prove that `CheckT` doesn't hold. We simply negate the assertion of the traces condition `CheckT` (see Figure 12.9). For instance, `TC2` experiment a manual on-ground start sequence, however the first `NH` value is less than the prescribed threshold (15). Therefore, `TC2` is colored red.

```
TC2 = aircraftCondition.true -> engineCondition.true -> MC_press -> CI_press
      -> engineStartON -> sav.open -> FC_press -> commandFuelON -> commandIgnON
      -> readNH.10 -> sav.close -> readNH.68 -> started -> STOP

assert not GREEN [T= CheckT                    assert not CheckT [T= GREEN
```

Figure 12.9: FDR2 test script to check *Red* test case.

12.4.2 Establishing a PCO

The test verdict is obtained during the execution of a test case in the HIL rig. ROLLS-ROYCE uses a propetary scripting language in order to write test scripts; a snippet of such script is shown below:

```
1  Set("MasterCrankCnd",1)
2  WaitTime(2)
3  Set("REngContinuousIGN",1)
4  WaitTime(2)
5  Set("REngStartCnd",1)
6  WaitTime(2)
7  Set("REngStartCnd",0)
8  WaitUntil("NHP>15")
9  Set("MasterLever",0)
10 WaitUntil("LIT==1")
11 Set("FlightStatus",1)
12 WaitUntil("NHP>65",60)
13 ...
```

Line 1,3,5 are commands to switch the *Master Crank*, *Continuos Ignition* and *Fuel Control* switch to `ON` respectively. The time delay, in between commands, is necessary in order to capture the signal, and store it in a log for an offline analysis.

We now establish a PCO $\mathcal{P} = (\mathcal{A}, \|...\|, \mathcal{D})$ in the following way:

- The alphabet of primitive events: $\mathcal{A} = \{$ `MasterCrankCnd`,...,`FlightStatus`$\}$.
- We use ASN.1 (Abstract Syntax Notation One) [Dub00] to map the primitive events to CSP events.

```
MasterCrankCnd ::= ENUMERATED{
            MC_press        (1)
            MC_release      (0)
}
```

```
MasterLever ::= ENUMERATED{
            FC_press        (1)
            FC_release      (0)
}
```

  For example, we describe `MasterCrankCnd` and `MasterLever` as the **ASN.1** type *ENUMERATED*; `MasterCrankCnd` can take only the values specified in the list, e.g., the value 0 stands for `MC_press`.

- The direction of primitive events are defined as follows: *ts*2*RIG* stands for signals which are sent from the testing software to the HIL rig. For instance, `set(...)` in the test script are of type *ts*2*RIG*. The other direction *RIG*2*ts*, which stands for signals which are sent from the HIL rig to the testing software. Those are captured by different logging system; for instance `Log_HST(...)`: logs primary variables of the EEC, `Log_Mod(...)`: logs simulation parameters and `Log_HST(...)`: logs

aircraft discretes. The following are examples of such logging system:

```
Log_HST("mstrstrtswtchsncckpt")    ;Master Start Switch - EEC variable
Log_Mod("P30")                     ;Engine pressure simulation parameters
Log_ARINC("IL270_DAU2_LA_2_B20")   ;SAV discretes
```

After running the test case in the HIL rig, the analysis of the test result is done "offline", by analyzing the different logs. Such "offline" analysis is carried out using tools provided within MatLab. Basically, in this process the systems engineer analysis the behaviour of the different signals (captured in the log files).

12.5 Summary and evaluation of the project

In this chapter we have described a work completed in cooperation with the ROLLS-ROYCE system verification team. We have applied the theory of formal testing based on CSP-CASL, to the starting system of ROLLS-ROYCE BR725 control software. We have modeled the system in CSP, evaluate test suites against the formal model using the model checker FDR2. We executed the test suites in an in-the-loop setting of the SUT. The SUT did not show any deviation from the intended behaviour, i.e., the testing process increased the trust in its correctness.

The modeling and testing of such systems worked successfully on the chosen of abstraction. Overall, modeling the system in CSP was quite. On the positive side, various CSP operator came very handy in the modeling process. The interleaving operator, the sequential composition, the hiding operator and the interrupt allowed us to capture many system aspects in an elegant way. On the negative side, the global state approach of CSP forced us to explicitly have one process name per transition (arrow in the activity diagrams). This allowed us to take care of or ignore state changes of the buttons while following the control flow of the activity diagram. Overall, however, CSP served well in modeling such a controller. On the tool side, FDR2 and PROBE coped quite easily in discharging the proof obligations, w.r.t, test coloring and deadlock/livelock/determinism check.

In general this case study demonstrates the applicability of our testing theory to industrial systems: it scales up to real world applications and it potentially fits into current verification practice at ROLLS-ROYCE. Due to time restriction, the decision procedure to determine the test verdict on the fly, by running the tests on the rig, has not been implemented. As described earlier, the test verdict is determined off-line (using tools like MatLab), by analyzing the different signals and log files.

On the other side, our approach gave new insights to the ROLLS-ROYCE BR725 system verification team. As a first point we mention that the ability of our approach to formally link the outcome of a test case with the specification by means of coloring technique, could partially help the engineers in the certification process. In particular, the engineers could use our coloring technique to trace which test case comes from which part of the specification.

In fact, engineers are required to explicitly illustrate from which part of the specification a certain test case has been designed – this is formally know as the *traceablity aim*.

As a second point we mention, that our approach gave a new insight in designing negative test cases. That is, test cases that experiment for properties that are not specified in the specification. More often the test engineers designs test cases which experiments the intended behavior prescribed in the specification.

Conclusion

(CHAPTER ...13)

Conclusions and further work

Contents

THE final chapter of this thesis present a short summary of the work and highlights its scientific contributions. In Section 13.2 we give an overview of possible future work in the area of systems development notion for CSP-CASL and specification based testing for CSP-CASL.

13.1 Summary

In this thesis we have described a theoretical and industrial application in the area of formal systems development, verification and formal testing using the specification language CSP-CASL. The latter is a comprehensive specification language which allows to describe systems in a combined algebraic/ process algebraic notation. To this end it integrates the process algebra CSP and the algebraic specification language CASL.

The thesis has proposed various formal development notions for CSP-CASL capable of capturing informal vertical and horizontal software development which we typically find in industrial applications. Here, we have presented two directions of system development: a refinement (or *vertical development*) notion for CSP-CASL; and an enhancement (or *horizontal development*) notion for CSP-CASL specifications.

For the refinement part, we have defined a new notion based on model class inclusion with arbitrary change of signature in the data part. We also presented a theory of enhancement for CSP-CASL. This theory allows us to capture the notion of *horizontal development*, in which new features (or functions) are added to existing systems.

We have provided proof techniques for the CSP-CASL development notions and verification methodologies to prove interesting properties of reactive systems. Here, we have

presented techniques to discharge proof obligations that could arise from the development notions of CSP-CASL.

On the refinement side of CSP-CASL specifications, we established an approach based on a decomposition theorem. Such decomposition theorem allows us to prove CSP-CASL refinement, first by reasoning about data refinement and then by process refinement. Based on this approach we are able to re-use existing tools to discharge proof obligations.

On the enhancement side of CSP-CASL specifications, we have proposed three enhancement patterns that allow us to capture the notions of adding new features to existing specifications.

We have proposed proofs techniques for the verification of properties of CSP-CASL specifications. Here, we have illustrated how to analyse deadlock and livelock freeness in the context of CSP-CASL. We have established establish a proof technique for deadlock and livelock freeness based on CSP-CASL refinement, which turns out to be complete.

We have also proposed a theoretical framework for formal testing from CSP-CASL specifications. Here, we have presented a conformance relation between a physical system and a CSP-CASL specification. In particular we have studied the relation between CSP-CASL development notion and the implemented system. The major innovations are the separation of the test oracle and the test evaluation problem by defining:

- the expected result (*green*, *red* and *yellow*) and,
- the verdict (*pass*, *fail* and *inconclusive*) of a test case.

The CSP-CASL specification determines the alphabet of the test suite, and the expected result of each test case. The expected result of a test case, in terms of the coloring scheme, is proved using CSP-CASL-PROVER.

The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test processes. Here, we have defined an algorithm which allows to determine the verdict of the test case on the fly. Moreover, we have presented a link between CSP-CASL development notion and the testing theory. Such link allow us to perform testing at all stages in a system's design, and to re-use test cases. For the latter, in the case of enhancement we showed that test cases which have been designed for basic features can be re-used whenever a more advanced product is conceived which includes these features.

The proposed theoretical notions of formal system development, property verification and formal testing for CSP-CASL, have been successfully applied to two industrial application: an electronic payment system called EP2 and the starting system of the BR725 ROLLS-ROYCE control software.

13.2 Further work

There are a number of questions which arise from this work that could be undertaken to follow on from this project. In the following subsections, we address some of these aspects.

13.2.1 New development notions for CSP-CASL

Regarding development notions for CSP-CASL, for both directions we identify some direction for future works:

Vertical development In terms of vertical development notions, more "sophisticated" refinement notions for CSP-CASL could be studied. In [BST08], Bidoit et al., presents a refinement notion based on observational interpretation of CASL specifications.

The study of the *behavioural refinement* [BST06, BH05], notions is motivated by the fact that, in general an implementation does not need to satisfy strictly the properties outlined in the abstract specification but it can be considered as correct if this implementation respects the observable consequences of the specification to be implemented. Often there are models that do not satisfy the axioms in a strictly way but in which all observations nevertheless deliver the required results.

Following the work of Bidoit et al., one can develop observational refinement for CSP-CASL. In the context of EP2 such refinement would be required in order to capture the relations between the more detailed levels. That is, in order to capture the XML level of the EP2 system we need a more 'sophisticated' refinement notion. The XML level in EP2 is the a further refinement of the concrete component level. On the data part, one would model in CASL the various constructs of the XML messages. Figure 13.2.1 illustrate this idea.

Horizontal development For the horizontal development in CSP-CASL, there are other notions of enhancement which are not covered by our definition. For example, in object-oriented systems, re-use is by inheritance of signatures and methods: The enhances version of a software product may inherit certain fields and classes, and redefine others. It would be intresting to study such notions also for CSP-CASL.

13.2.2 Testing theory for CSP-CASL

Regarding future work for formal testing from CSP-CASL we mention:

Automatic tool support for coloring test cases We have developed a convincing proof strategy in CSP-CASL-PROVER to discharge proof obligations for the coloring of test cases. However, one could develop new strategy in order to discharge such proofs in a (semi)automatic fashion, for instance by incorporating automated provers.

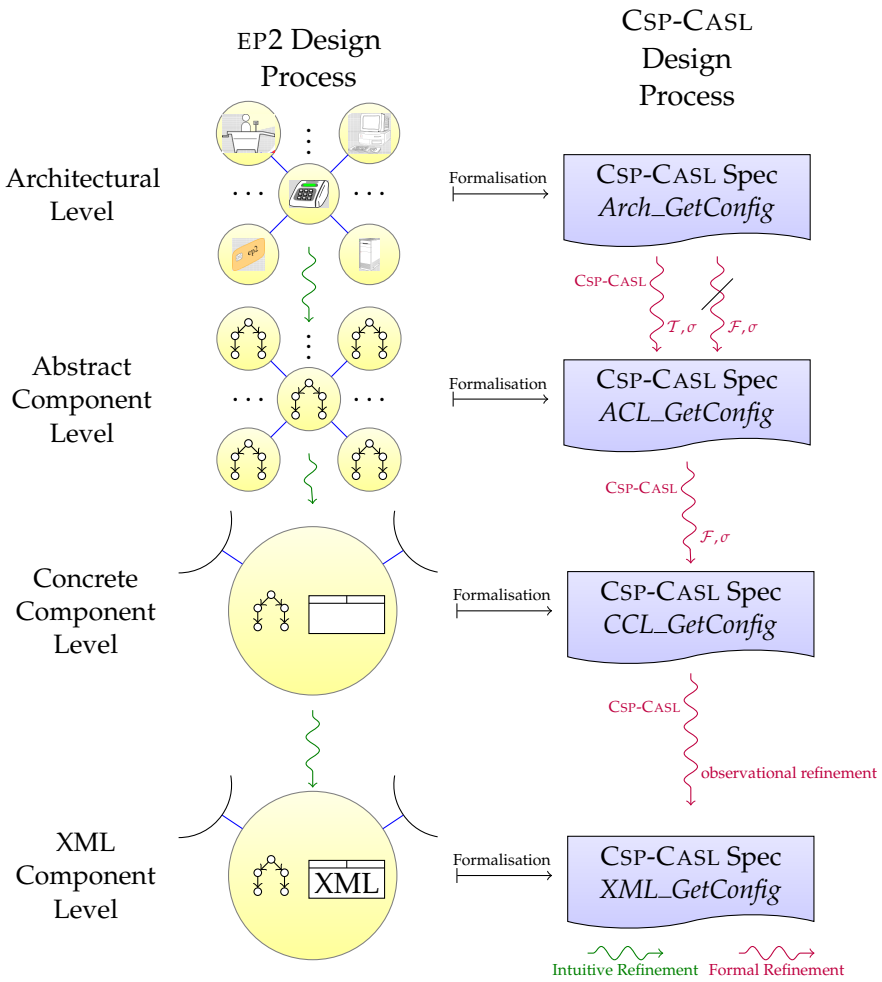

Figure 13.1: EP2 observational refinement in CSP-CASL.

Test coverage In this thesis, we haven't addressed the topic of test coverage criteria. In the literature one finds a sheer amount of test coverage metrics, e.g., [WRHM06]. It would be interesting to include a coverage criteria in our testing theory.

Automatic test case selection Many research activities have been directed at finding appropriate theories and algorithms to derive test cases from formal specifications such that certain correctness properties can be guranteed if the system under test passes all test cases of a test suite. Early attempts were contributes by Brinksma [Bri88] using the specification language LOTOS. Here, one could study new theories for the automatic test case selection from CSP-CASL specification based on a predefined coverage criteria.

Appendices

(APPENDIX ...A)

CSP-CASL development notion and testing

In this appendix we report proofs from Chapter 6, 7 and Section 10.3.

A.1 Proof of CSP-CASL reduct property

In the following we give a full proof of Theorem 6.1.8.

Reduct property over the CSP models Let P be an arbitrary CSP process of a CSP-CASL specification $Sp = (D, P)$. Moreover, let $\sigma : \Sigma \to \Sigma'$ be a CSP-CASL data logic signature morphism and M' a Σ'-model. Then,

$$\begin{aligned}
&\mathit{traces}(\llbracket P \rrbracket_{\nu:X \to M'|_\sigma}) = \hat{\alpha}_{\mathcal{T}}(\mathit{traces}(\llbracket \rho(P) \rrbracket_{\hat{\nu}:\sigma(X) \to M'})) \\
&\mathit{failures}(\llbracket P \rrbracket_{\nu:X \to M'|_\sigma}) = \hat{\alpha}_{\mathcal{F}}(\mathit{failures}(\llbracket \rho(P) \rrbracket_{\hat{\nu}:\sigma(X) \to M'})) \\
&\mathit{divergences}(\llbracket P \rrbracket_{\nu:X \to M'|_\sigma}) = \hat{\alpha}_{\mathcal{N}}(\mathit{divergences}(\llbracket \rho(P) \rrbracket_{\hat{\nu}:\sigma(X) \to M'}))
\end{aligned}$$

where X is the set of free variables in P, $\nu : X \to M'|_\sigma$ and $\hat{\nu} : \sigma(X) \to M'$ are variable evaluations with

$$\nu(x : s) = \hat{\nu}(x : \sigma(s)).$$

PROOF. The proof is by structural induction on the CSP process operator P. Here, we show for each semantical model, how the proof is carried out for each CSP process operator. The proofs for the primitive process *STOP* and action prefix $a \to P$ can be found in Section 6.1.

Traces model

- *SKIP* : We need to prove the following:

$$\mathit{traces}(\llbracket \mathit{SKIP} \rrbracket_\nu) = \hat{\alpha}_{\mathcal{T}}(\mathit{traces}(\llbracket \rho(\mathit{SKIP}) \rrbracket_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we calculate the trace set, which is $\{\langle\rangle, \langle\checkmark\rangle\}$. Applying the inverse translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ we obtain $\hat{\alpha}^{*\checkmark}(\langle\rangle, \langle\checkmark\rangle)$, i.e., $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(SKIP)]\!]_{\hat{\nu}}))$.

Thus, $traces([\![SKIP]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(SKIP)]\!]_{\hat{\nu}}))$.

- *DIV* : We need to prove the following:

$$traces([\![DIV]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(DIV)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we calculate the trace set, which is $\{\langle\rangle\}$. Applying the inverse translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ we obtain $\hat{\alpha}^*(\langle\rangle)$, i.e., $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(DIV)]\!]_{\hat{\nu}}))$. Thus, $traces([\![DIV]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(DIV)]\!]_{\hat{\nu}}))$.

- $?x :: s \rightarrow P$: We need to prove the following:

$$traces([\![?x :: s \rightarrow P]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(?x :: s \rightarrow P)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces(?x :: [\![s]\!]_{\nu} \rightarrow [\![P]\!]_{\lambda z.\nu}).$$

We then calculate the trace set:

$$\{\langle\rangle\} \cup \{\langle a\rangle \frown t \mid t \in traces([\![P]\!]_{\nu[a/x]}),\ a \in [\![s]\!]_{\nu}\}.$$

We now unfold the definition of the variable evaluation $[\![s]\!]_{\nu}$ (details of this definition can be found in [Rog06]):

$$[\![s]\!]_{\nu} = [s]_{\sim_{M'|_{\sigma}}}.$$

Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_{\sigma}}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$. We now apply the inverse alphabet translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_{\nu})$, we obtain:

$$\{\hat{\alpha}^*(\langle\rangle)\} \cup \{\langle a\rangle \frown t \mid t \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}[a/x]})),\ a \in \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})\}.$$

Pulling out the $\hat{\alpha}$ from the above trace set, we obtain $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(?x :: s \rightarrow P)]\!]_{\hat{\nu}}))$. Thus, $traces([\![?x :: s \rightarrow P]\!]_{\nu} = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(?x :: s \rightarrow P)]\!]_{\hat{\nu}}))$.

- $!x :: s \rightarrow P$: The same procedure as in the case of $?x :: s \rightarrow P$.

- $P \fatsemi Q$: We need to prove the following:

$$traces([\![P \fatsemi Q]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![P]\!]_{\nu} \fatsemi [\![Q]\!]_{\nu}).$$

We then calculate the trace set:

$$\begin{array}{rl} & traces([\![P]\!]_\nu) \cap \mathcal{A}lph(M'|_\sigma)^* \\ \cup & \{s \frown t \mid s \frown \langle\checkmark\rangle \in traces([\![P]\!]_\nu), t \in traces([\![Q]\!]_\nu)\}. \end{array}$$

Applying the inverse alphabet translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$ and $traces([\![Q]\!]_\nu)$, we obtain:

$$\begin{array}{rl} & \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}})) \cap \hat{\alpha}^*(\mathcal{A}lph(M')^*) \\ \cup & \{s \frown t \mid s \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}})), t \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(Q)]\!]_{\hat{\nu}}))\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above trace set, we obtain $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}}))$. Thus, $traces([\![P \fatsemi Q]\!]_\nu = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}}))$.

- $P \,\Box\, Q$: We need to prove the following:

$$traces([\![P \,\Box\, Q]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \,\Box\, Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![P]\!]_\nu \,\Box\, [\![Q]\!]_\nu).$$

We then calculate the trace set:

$$traces([\![P]\!]_\nu) \cup traces([\![Q]\!]_\nu).$$

Applying the inverse alphabet translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$ and $traces([\![Q]\!]_\nu)$, we obtain:

$$\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}})) \cup \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(Q)]\!]_{\hat{\nu}})).$$

Pulling out the $\hat{\alpha}$ from the above trace set, we obtain $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \,\Box\, Q)]\!]_{\hat{\nu}}))$. Thus, $traces([\![P \,\Box\, Q]\!]_\nu = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \,\Box\, Q)]\!]_{\hat{\nu}}))$.

- $P \sqcap Q$: The same procedure as in the case of $P \,\Box\, Q$.
- $P \,|[\, s \,]|\, Q$: We need to prove the following:

$$traces([\![P \,|[\, s \,]|\, Q]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![P]\!]_\nu \,|[\, [\![s]\!]_\nu \,]|\, [\![Q]\!]_\nu).$$

We then calculate the trace set:

$$\bigcup\{t1 \,|[\, [\![s]\!]_\nu \,]|\, t2 \mid t1 \in traces([\![P]\!]_\nu) \wedge t2 \in traces([\![Q]\!]_\nu)\}.$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_\nu = [s]_{\sim_{M'|_\sigma}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_\sigma}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$ and $traces([\![Q]\!]_\nu)$, we obtain:

$$\bigcup\{t1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, t2 \mid \begin{array}{l} t1 \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}})) \\ \wedge\ t2 \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(Q)]\!]_{\hat{\nu}}))\}. \end{array} \quad (p1)$$

In order to pull out the inverse translation $\hat{\alpha}$ from the above trace set, we need to show the following equality:

$$t1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, t2 = \hat{\alpha}(t1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, t2). \quad (p2)$$

This is done by induction on the trace length n and m of the trace $t1$ and $t2$. Let $[\![a]\!]_{\hat{\nu}} \in [\![\sigma^S(s)]\!]_{\hat{\nu}}$ and $[\![b]\!]_{\hat{\nu}} \notin [\![\sigma^S(s)]\!]_{\hat{\nu}}$. As a base case we consider the following cases:

$$\begin{array}{l} \langle\rangle \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle\rangle = \hat{\alpha}(\langle\rangle \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle\rangle) = \{\langle\rangle\} \\ \langle\rangle \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![a]\!]_{\hat{\nu}}\rangle = \hat{\alpha}(\langle\rangle \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![a]\!]_{\hat{\nu}}\rangle) = \{\} \\ \langle\rangle \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![b]\!]_{\hat{\nu}}\rangle = \hat{\alpha}(\langle\rangle \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![b]\!]_{\hat{\nu}}\rangle) = \{\langle [\![b]\!]_{\hat{\nu}}\rangle\}. \end{array}$$

Let the equality $p2$ holds for a trace length $n-1$ and $m-1$. We now consider the following cases (here, let $q1, q2 \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}}))$):

1. Let $t1 = \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1$ and $\langle [\![b]\!]_{\hat{\nu}}\rangle \frown q2$, then for the left hand side we have:

$$\begin{array}{ll} & \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![b]\!]_{\hat{\nu}}\rangle \frown q2 \\ = & \{\langle [\![b]\!]_{\hat{\nu}}\rangle \frown q \mid q \in \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, q2\}. \end{array}$$

   By induction hypothesis for the right hand side we obtain:

$$\begin{array}{ll} & \hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![b]\!]_{\hat{\nu}}\rangle \frown q2) \\ = & \{\langle [\![b]\!]_{\hat{\nu}}\rangle \frown q \mid q \in \hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, q2)\}. \end{array}$$

2. Let $t1 = \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1$ and $\langle [\![a]\!]_{\hat{\nu}}\rangle \frown q2$, then for the left hand side we have:

$$\begin{array}{ll} & \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q2 \\ = & \{\langle [\![a]\!]_{\hat{\nu}}\rangle \frown q \mid q \in q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, q2\}. \end{array}$$

   By induction hypothesis for the right hand side we obtain:

$$\begin{array}{ll} & \hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}}\rangle \frown q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![a]\!]_{\hat{\nu}}\rangle \frown q2) \\ = & \{\langle [\![a]\!]_{\hat{\nu}}\rangle \frown q \mid q \in \hat{\alpha}(q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, q2)\}. \end{array}$$

3. Let $t1 = \langle [\![a]\!]_{\hat{\nu}} \rangle \frown q1$ and $\langle [\![a']\!]_{\hat{\nu}} \rangle \frown q2$, where $[\![a']\!]_{\hat{\nu}} \in [\![\sigma^S(s)]\!]_{\hat{\nu}}$ and $[\![a]\!]_{\hat{\nu}} \neq [\![a']\!]_{\hat{\nu}}$. Then for the left hand side we have:

$$\langle [\![a]\!]_{\hat{\nu}} \rangle \frown q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![a']\!]_{\hat{\nu}} \rangle \frown q2 = \{\}.$$

By induction hypothesis for the right hand side we obtain:

$$\hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}} \rangle \frown q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![a']\!]_{\hat{\nu}} \rangle \frown q2) = \{\}.$$

4. Let $t1 = \langle [\![b]\!]_{\hat{\nu}} \rangle \frown q1$ and $\langle [\![b']\!]_{\hat{\nu}} \rangle \frown q2$, where $[\![b']\!]_{\hat{\nu}} \notin [\![\sigma^S(s)]\!]_{\hat{\nu}}$. Then for the left hand side we have:

$$\begin{array}{ll} & \langle [\![a]\!]_{\hat{\nu}} \rangle \frown q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![b]\!]_{\hat{\nu}} \rangle \frown q2 \\ = & \{\langle [\![b]\!]_{\hat{\nu}} \rangle \frown q \mid q \in q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, \langle [\![b']\!]_{\hat{\nu}} \rangle \frown q2\} \\ & \cup \{\langle [\![b']\!]_{\hat{\nu}} \rangle \frown q \mid q \in \langle [\![b]\!]_{\hat{\nu}} \rangle \frown q1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, q2\}. \end{array}$$

By induction hypothesis for the right hand side we obtain:

$$\begin{array}{ll} & \hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}} \rangle \frown q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![b]\!]_{\hat{\nu}} \rangle \frown q2 \\ = & \{\langle [\![b]\!]_{\hat{\nu}} \rangle \frown q \mid q \in \hat{\alpha}(q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, \langle [\![b']\!]_{\hat{\nu}} \rangle \frown q2\} \\ & \cup \{\langle [\![b']\!]_{\hat{\nu}} \rangle \frown q \mid q \in \hat{\alpha}(\langle [\![b]\!]_{\hat{\nu}} \rangle \frown q1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, q2)\}. \end{array}$$

Thus, we have that $t1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, t2 = \hat{\alpha}(t1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, t2)$. We now can pull out $\hat{\alpha}_{\mathcal{T}}$ from the trace set in $p1$, and we obtain $\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}}))$. Hence, $traces([\![P \,|[\, s \,]|\, Q]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}}))$.

- $P \;\; |[\, s1 \,|\, s2 \,]| \;\; Q$: Here, since $P \;\; |[\, s1 \,|\, s2 \,]| \;\; Q = P \,|[\, s1 \cap s2 \,]|\, Q$, the proof follows the same procedure as in the case of the generalized parallel.
- $P \;||\; Q$: Here, since $P \;||\; Q = P \,|[\, \mathcal{A}lph(M) \,]|\, Q$, the proof follows the same procedure as in the case of the generalized parallel.
- $P \;|||\; Q$: Here, since $P \;|||\; Q = P \,|[\, \{\} \,]|\, Q$, the proof follows the same procedure as in the case of the generalized parallel.
- $P \setminus s$: We need to prove the following:

$$traces([\![P \setminus s]\!]_{\nu}) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P \setminus s)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![P]\!]_{\nu} \setminus [\![s]\!]_{\nu}).$$

We then calculate the trace set:

$$\{t \setminus [\![s]\!]_{\nu} \mid t \in traces([\![P]\!]_{\nu})\}.$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_\nu = [s]_{\sim_{M'|_\sigma}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_\sigma}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the traces domain $\hat{\alpha}_\mathcal{T}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$, we obtain:

$$\{t \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \mid t \in \hat{\alpha}_\mathcal{T}(traces([\![\rho(P)]\!]_{\hat{\nu}}))\}. \qquad (h1)$$

In order to pull out the inverse translation $\hat{\alpha}$ from the above trace set, we need to show the following equality:

$$t \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) = \hat{\alpha}(t \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}}). \qquad (h2)$$

This is done by induction on the trace length n of t. For $n = 0$, we have that

$$\langle\rangle \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) = \hat{\alpha}(\langle\rangle \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}}) = \langle\rangle$$

Let the equality in $(h2)$ holds for a trace q of length $n-1$. Now let $t = \langle [\![a]\!]_{\hat{\nu}} \rangle \frown q$ where $q \in \hat{\alpha}_\mathcal{T}(traces([\![\rho(P)]\!]_{\hat{\nu}}))$. Here for the left hand side we have:

$$\hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}} \rangle \frown q) \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) := \begin{cases} \hat{\alpha}([\![a]\!]_{\hat{\nu}}) \frown (q \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})) & \text{if } [\![a]\!]_{\hat{\nu}} \notin [\![\sigma^S(s)]\!]_{\hat{\nu}} \\ q \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) & \text{otherwise.} \end{cases}$$

By induction hypothesis, for the right hand side we have that:

$$\hat{\alpha}(\langle [\![a]\!]_{\hat{\nu}} \rangle \frown q \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}}) := \begin{cases} \hat{\alpha}([\![a]\!]_{\hat{\nu}} \frown (q \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}})) & \text{if } [\![a]\!]_{\hat{\nu}} \notin [\![\sigma^S(s)]\!]_{\hat{\nu}} \\ \hat{\alpha}(q \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}}) & \text{otherwise.} \end{cases}$$

Thus, we have that $t \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) = \hat{\alpha}(t \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}})$. This allow us to pull out the $\hat{\alpha}$ from the trace set in $(h1)$, and we obtain $\hat{\alpha}_\mathcal{T}(traces([\![\rho(P \setminus s)]\!]_{\hat{\nu}}))$. Hence, $traces([\![P \setminus s]\!]_\nu) = \hat{\alpha}_\mathcal{T}(traces([\![\rho(P \setminus s)]\!]_{\hat{\nu}}))$.

- $P[[p]]$: We need to prove the following:

$$traces([\![P[[p]]]\!]_\nu) = \hat{\alpha}_\mathcal{T}(traces([\![\rho(P[[p]])]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces([\![P]\!]_\nu[[[\![p]\!]_\nu]]).$$

We then calculate the trace set:

$$\{t \mid \exists s \in traces([\![P]\!]_\nu) \,.\, s[\![p^*]\!]_\nu t\}.$$

We now unfold the definition of the variable evaluation $[\![p^*]\!]_\nu$ (details of this definition can be found in [Rog06]):

$$[\![p_{s1s2}]\!]_\nu = \{([s1,x]_{\sim_{M'|_\sigma}}, [s2,y]_{\sim_{M'|_\sigma}}) \mid (x,y) \in (p_{s1s2})_{M'|_\sigma}\}.$$

Applying the alphabet translation α we obtain:

$$\begin{array}{ll} & \{(\alpha([s1,x]_{\sim_{M'|_\sigma}}), \alpha([s2,y]_{\sim_{M'|_\sigma}})) \mid (x,y) \in \alpha((p_{s1s2})_{M'|_\sigma})\} \\ = & \{([\sigma^S(s1),x]_{\sim_{M'}}, [\sigma^S(s2),y]_{\sim_{M'}}) \mid (x,y) \in (p_{\sigma^S(s1)\sigma^S(s2)})_{M'}\} \\ = & \{([\![\sigma^S(s1),x]\!]_{\hat\nu}, [\![\sigma^S(s2),y]\!]_{\hat\nu}) \mid (x,y) \in (p_{\sigma^S(s1)\sigma^S(s2)})_{M'}\} \end{array}$$

We now apply the inverse alphabet translation of the traces domain $\hat\alpha_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$, we obtain:

$$\{t \mid \exists\, s \in \hat\alpha_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat\nu})\,.\, s\ \hat\alpha([\![p^*]\!]_{\hat\nu})\ t\}. \qquad (r1)$$

In order to pull out the inverse translation $\hat\alpha$ from the above trace set, we need to show the following equality:

$$s\ \hat\alpha([\![p^*]\!]_{\hat\nu})\ t = \hat\alpha(s\ [\![p^*]\!]_{\hat\nu}\ t). \qquad (r2)$$

This is done by induction on the trace length n of t. For $n = 0$: we have

$$\langle\rangle\ \hat\alpha([\![p^*]\!]_{\hat\nu})\ \langle\rangle = \hat\alpha(\langle\rangle [\![p^*]\!]_{\hat\nu}\ \langle\rangle) = \langle\rangle$$

Let the equality in $(r2)$ holds for a trace length of $n-1$. Now let $t = \langle [\![a]\!]_{\hat\nu}\rangle \frown q$ where $q \in \hat\alpha_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat\nu}))$. Here for the left hand side we have:

$$s\ \hat\alpha([\![p^*]\!]_{\hat\nu})\ (\langle [\![a]\!]_{\hat\nu}\rangle \frown q) ::= \begin{cases} \hat\alpha([\![a]\!]_{\hat\nu}) \frown (\hat\alpha(s)\hat\alpha([\![p^*]\!]_{\hat\nu})\ \hat\alpha(q)) & \text{if } [\![a]\!]_{\hat\nu} \notin [\![\sigma^S(s)]\!]_{\hat\nu} \\ \hat\alpha(s)\hat\alpha([\![\sigma^S(s)]\!]_{\hat\nu}))\hat\alpha(q) & \text{otherwise.} \end{cases}$$

By induction hypothesis, for the right hand side we have that:

$$\hat\alpha(s\ [\![p^*]\!]_{\hat\nu}\ \langle [\![a]\!]_{\hat\nu}\rangle \frown q) ::= \begin{cases} \hat\alpha([\![a]\!]_{\hat\nu} \frown (s[\![p^*]\!]_{\hat\nu}\ q)) & \text{if } [\![a]\!]_{\hat\nu} \notin [\![\sigma^S(s)]\!]_{\hat\nu} \\ \hat\alpha(s[\![\sigma^S(s)]\!]_{\hat\nu})q) & \text{otherwise.} \end{cases}$$

Thus, we have that $\hat\alpha(s)\ \hat\alpha([\![p^*]\!]_{\hat\nu})\ \hat\alpha(t) = \hat\alpha(s\ [\![p^*]\!]_{\hat\nu}\ t)$. This allow us to pull out the $\hat\alpha$ from the trace set in $(r1)$, and we obtain $\hat\alpha_{\mathcal{T}}(traces([\![\rho(P[[p]])]\!]_{\hat\nu}))$.
Hence, $traces([\![P[[p]]]\!]_\nu) = \hat\alpha_{\mathcal{T}}(traces([\![\rho(P[[p]])]\!]_{\hat\nu}))$.

- **if** φ **then** P **else** Q: We need to prove the following:

$$\begin{array}{ll} & traces([\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu) \\ = & \hat\alpha_{\mathcal{T}}(traces([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat\nu})). \end{array}$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$traces(\textbf{if } [\![\varphi]\!]_\nu \textbf{ then } [\![P]\!]_\nu \textbf{ else } [\![Q]\!]_\nu).$$

We then calculate the trace set:

$$\begin{cases} traces([\![P]\!]_\nu); & [\![\varphi]\!]_\nu \text{ if evaluates to } true \\ traces([\![Q]\!]_\nu); & [\![\varphi]\!]_\nu \text{ if evaluates to } false. \end{cases}$$

We now unfold the definition of the variable evaluation $[\![\varphi]\!]_\nu$ (details of this definition can be found in [Rog06]):

$$[\![\varphi]\!]_\nu := \begin{cases} true & \text{if } \nu \Vdash \varphi \\ false & \text{if not } \nu \Vdash \varphi \end{cases}$$

In [Rog06] (Theorem 6 – *Generalized satisfaction condition*) proves that:

$$\nu \Vdash \sigma(\varphi) \text{ iff } \hat{\nu} \Vdash \sigma.$$

Applying the alphabet translation α we obtain:

$$\alpha([\![\varphi]\!]_\nu) = [\![\sigma(\varphi)]\!]_{\hat{\nu}} := \begin{cases} true & \text{if } \hat{\nu} \Vdash \varphi \\ false & \text{if not } \hat{\nu} \Vdash \varphi \end{cases}$$

We now apply the inverse alphabet translation of the traces domain $\hat{\alpha}_{\mathcal{T}}$ and using the induction hypothesis on $traces([\![P]\!]_\nu)$ and $traces([\![Q]\!]_\nu)$, we obtain:

$$\begin{cases} \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}})); & \text{if } \hat{\nu} \Vdash \varphi \\ \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(Q)]\!]_{\hat{\nu}})); & \text{if not } \hat{\nu} \Vdash \varphi. \end{cases}$$

Pulling out the $\hat{\alpha}$ from the above trace set, we obtain:

$$\hat{\alpha}_{\mathcal{T}}(traces([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}})).$$

Thus, $traces([\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}}))$.

Stable failure model

For each process operator, the trace component is identical to the one presented for the traces model. Here, we illustrate how the proves goes for the failures component.

- *SKIP* : We need to prove the following:

$$failures([\![SKIP]\!]_\nu) = \hat{\alpha}(failures([\![\rho(SKIP)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we calculate the failure set:

$$\{(\langle\rangle, X) \mid X \subseteq \mathcal{A}lph(M'|_\sigma)\} \bigcup \{(\langle\surd\rangle, X) \mid X \subseteq \mathcal{A}lph(M'|_\sigma)^\surd\}.$$

Applying the inverse translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ we obtain:

$$\begin{array}{rl} & \{(\hat{\alpha}^*(\langle\rangle), \hat{\alpha}^\surd_{\mathbb{P}}(X)) \mid \hat{\alpha}^\surd_{\mathbb{P}}(X) \subseteq \hat{\alpha}(\mathcal{A}lph(M'))\} \\ \bigcup & \{(\hat{\alpha}^{*^\surd}(\langle\surd\rangle), \hat{\alpha}_{\mathbb{P}}(X)) \mid \hat{\alpha}_{\mathbb{P}}(X) \subseteq \hat{\alpha}^\surd \mathcal{A}lph(M')^\surd\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above failure set, we obtain $\hat{\alpha}(failures(\rho[\![SKIP]\!]))$. Thus, $failures([\![SKIP]\!]_\nu) = \hat{\alpha}(failures([\![\rho(SKIP)]\!]_{\hat{\nu}}))$.

- *DIV* : We need to prove the following:

$$\mathit{failures}([\![\mathit{DIV}]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(\mathit{DIV})]\!]_{\hat{\nu}})).$$

This trivially holds as the failure set for *DIV* is the empty set.

- $?x :: s \to P$: We need to prove the following:

$$\mathit{failures}([\![?x :: s \to P]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(?x :: s \to P)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}(?x :: [\![s]\!]_\nu \to [\![P]\!]_{\lambda\, z.\nu}).$$

We then calculate the failure set:

$$\begin{array}{ll} & \{(\langle\rangle, Y) \mid \mathcal{A}lph(M'|_\sigma) \cap Y = \emptyset,\ Y \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark)\} \\ \cup & \{(\langle a\rangle \frown t, Y) \mid (t, Y) \in \mathit{failures}([\![P]\!]_{\nu[a/x]}),\ a \in [\![s]\!]_\nu\}. \end{array}$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_\nu = [s]_{\sim_{M'|_\sigma}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_\sigma}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$, we obtain:

$$\begin{array}{ll} & \{(\hat{\alpha}^*(\langle\rangle), \hat{\alpha}^\checkmark_{\mathbb{P}}(Y)) \mid \mathcal{A}lph(M'|_\sigma) \cap \hat{\alpha}^\checkmark_{\mathbb{P}}(Y) = \emptyset,\ \hat{\alpha}^\checkmark_{\mathbb{P}}(Y) \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark)\} \\ \cup & \{(\hat{\alpha}^{*\checkmark}(\langle a\rangle \frown t), \hat{\alpha}^\checkmark_{\mathbb{P}}(Y)) \mid (t, Y) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}[a/x]})),\ a \in \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above failure set, we obtain $\hat{\alpha}(\mathit{failures}([\![\rho(?x :: s \to P)]\!]_{\hat{\nu}}))$. Thus, $\mathit{failures}([\![?x :: s \to P]\!]_\nu = \hat{\alpha}(\mathit{failures}([\![\rho(?x :: s \to P)]\!]_{\hat{\nu}}))$.

- $P \fatsemi Q$: We need to prove the following:

$$\mathit{failures}([\![P \fatsemi Q]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}([\![P]\!]_\nu \fatsemi [\![Q]\!]_\nu).$$

We then calculate the failure set:

$$\begin{array}{ll} & \{(t, X) \mid t \in \mathcal{A}lph(M'|_\sigma)^*,\ (t, X \cup \{\checkmark\}) \in \mathit{failures}([\![P]\!]_\nu)\} \\ \bigcup & \{(t \frown q, X) \mid t \frown \langle\checkmark\rangle \in \mathit{traces}([\![P]\!]_\nu), (q, X) \in \mathit{failures}([\![Q]\!]_\nu)\}. \end{array}$$

Applying the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$, $\mathit{traces}([\![P]\!]_\nu)$ and $\mathit{failures}([\![Q]\!]_\nu)$, we obtain:

$$\begin{array}{ll} & \{(t, X) \mid t \in \hat{\alpha}^*(\mathcal{A}lph(M')^*),\ (t, X \cup \{\checkmark\}) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}}))\} \\ \bigcup & \{(t \frown q, X) \mid t \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(\mathit{traces}([\![\rho(P)]\!]_{\hat{\nu}})), (q, X) \in \hat{\alpha}(\mathit{failures}([\![\rho(Q)]\!]_{\hat{\nu}}))\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above failure set, we obtain $\hat{\alpha}(\mathit{failures}([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}}))$. Thus, $\mathit{failures}([\![P \fatsemi Q]\!]_\nu = \hat{\alpha}(\mathit{failures}([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}}))$.

- $P \,\Box\, Q$: We need to prove the following:

$$\mathit{failures}([\![P \,\Box\, Q]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \,\Box\, Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{traces}([\![P]\!]_\nu \,\Box\, [\![Q]\!]_\nu).$$

We then calculate the failure set:

$$\begin{array}{ll} & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}([\![P]\!]_\nu) \cap \mathit{failures}([\![Q]\!]_\nu)\} \\ \cup & \{(t, X) \mid (t, X) \in \mathit{failures}([\![P]\!]_\nu) \cup \mathit{failures}([\![Q]\!]_\nu), t \neq \langle\rangle\} \\ \cup & \{(\langle\rangle, X) \mid X \subseteq \mathcal{A}lph(M'|_\sigma) \ \wedge\ \langle\checkmark\rangle \in \mathit{traces}([\![P]\!]_\nu) \cup \mathit{traces}([\![Q]\!]_\nu)\}. \end{array}$$

Applying the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$, $\mathit{failures}([\![Q]\!]_\nu)$, $\mathit{traces}([\![P]\!]_\nu)$ and $\mathit{traces}([\![Q]\!]_\nu)$ we obtain:

$$\begin{array}{ll} & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})) \cap \hat{\alpha}(\mathit{failures}([\![\rho(Q)]\!]_{\hat{\nu}}))\} \\ \cup & \{(t, X) \mid (t, X) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})) \cup \hat{\alpha}(\mathit{failures}([\![\rho(Q)]\!]_{\hat{\nu}})), t \neq \langle\rangle\} \\ \cup & \{(\langle\rangle, X) \mid \hat{\alpha}^{\checkmark}_{\mathbb{P}}(X) \subseteq \hat{\alpha}(\mathcal{A}lph(M'|_\sigma)) \ \wedge\ \langle\checkmark\rangle \in \hat{\alpha}(\mathit{traces}([\![\rho(P)]\!]_{\hat{\nu}})) \\ & \qquad\qquad \cup\, \hat{\alpha}(\mathit{traces}([\![\rho(Q)]\!]_{\hat{\nu}}))\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above failure set, we obtain: $\hat{\alpha}(\mathit{failures}([\![\rho(P \,\Box\, Q)]\!]_{\hat{\nu}}))$. Thus, $\mathit{failures}([\![P \,\Box\, Q]\!]_\nu = \hat{\alpha}(\mathit{failures}([\![\rho(P \,\Box\, Q)]\!]_{\hat{\nu}}))$.

- $P \sqcap Q$: We need to prove the following:

$$\mathit{failures}([\![P \sqcap Q]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \sqcap Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}([\![P]\!]_\nu \sqcap [\![Q]\!]_\nu).$$

We then calculate the failure set:

$$\mathit{failures}([\![P]\!]_\nu) \cup \mathit{failures}([\![Q]\!]_\nu).$$

Applying the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$ and $\mathit{failures}([\![Q]\!]_\nu)$, we obtain:

$$\hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})) \cup \hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(Q)]\!]_{\hat{\nu}})).$$

Pulling out the $\hat{\alpha}$ from the above failure set, we obtain $\hat{\alpha}(\mathit{failures}([\![\rho(P \sqcap Q)]\!]_{\hat{\nu}}))$. Thus, $\mathit{failures}([\![P \sqcap Q]\!]_\nu = \hat{\alpha}(\mathit{failures}([\![\rho(P \sqcap Q)]\!]_{\hat{\nu}}))$.

- $P \,|[\, s \,]|\, Q$: We need to prove the following:

$$\mathit{failures}([\![P \,|[\, s \,]|\, Q]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}([\![P]\!]_\nu \,|[\, [\![s]\!]_\nu \,]|\, [\![Q]\!]_\nu).$$

We then calculate the failure set:

$$\begin{array}{ll}\{(u, Y \cup Z) \mid & Y - ([\![s]\!]_\nu \cup \{\checkmark\}) = Z - ([\![s]\!]_\nu \cup \{\checkmark\}),\\ & \exists\, t, q.\ (t, Y) \in \mathit{failures}([\![P]\!]_\nu), (q, Z) \in \mathit{failures}([\![Q]\!]_\nu),\\ & u \in t \,|[\, [\![s]\!]_\nu \,]|\, q,\ Y \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark),\ Z \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark)\}.\end{array}$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_\nu = [s]_{\sim_{M'|_\sigma}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_\sigma}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$ and $\mathit{failures}([\![Q]\!]_\nu)$ we obtain:

$$\begin{array}{ll}\{(u, Y \cup Z) \mid & Y - (\hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \cup \{\checkmark\}) = Z - (\hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \cup \{\checkmark\}),\\ & \exists\, t, q.\ (t, Y) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})), (q, Z) \in \hat{\alpha}(\mathit{failures}([\![\rho(Q)]\!]_{\hat{\nu}})),\\ & u \in t \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, q,\ \hat{\alpha}^\checkmark_{\mathbb{P}}(Y) \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark),\\ & \qquad\qquad \hat{\alpha}^\checkmark_{\mathbb{P}}(Z) \in \mathbb{P}(\mathcal{A}lph(M'|_\sigma)^\checkmark)\}.\end{array}$$

Here, we can pull out $\hat{\alpha}$ from the above failure set. This is thanks to $t1 \,|[\, \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \,]|\, t2 = \hat{\alpha}(t1 \,|[\, [\![\sigma^S(s)]\!]_{\hat{\nu}} \,]|\, t2)$ (see proof in the traces model). Consequently, we obtain $\hat{\alpha}(\mathit{failures}([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}}))$. Thus,

$$\mathit{failures}([\![P \,|[\, s \,]|\, Q]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}})).$$

- $P \setminus s$: We need to prove the following:

$$\mathit{failures}([\![P \setminus s]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \setminus s)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}([\![P]\!]_\nu \setminus [\![s]\!]_\nu).$$

We then calculate the failures set:

$$\{(t \setminus [\![s]\!]_\nu, Y) \mid (t, [\![s]\!]_\nu \cup Y) \in \mathit{failures}([\![P]\!]_\nu)\}.$$

We now unfold the definition of the variable evaluation, i.e.,$[\![s]\!]_\nu = [s]_{\sim_{M'|_\sigma}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_\sigma}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$, we obtain:

$$\{(t \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}), Y) \mid (t, (\hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) \cup Y) \in \hat{\alpha}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}}))\}.$$

Here, we can pull out $\hat{\alpha}$ from the above failure set. This is thanks to $t \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) = \hat{\alpha}(t \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}})$ (see proof in the traces model).

Consequently we obtain $\hat{\alpha}(\mathit{failures}([\![\rho(P \setminus s)]\!]_{\hat{\nu}}))$. Thus,

$$\mathit{failures}([\![P \setminus s]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(P \setminus s)]\!]_{\hat{\nu}})).$$

- $P[[p]]$: We need to prove the following:

$$\mathit{failures}([\![P[[p]]]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(P[[p]])]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}([\![P]\!]_\nu[[[\![p]\!]_\nu]]).$$

We then calculate the failure set:

$$\{(t, X) \mid \exists\, t'.\, (t', t) \in [\![p^*]\!]_\nu,\ (t', p^{-1}(X)) \in \mathit{failures}([\![P]\!]_\nu).$$

We now unfold the definition of the variable evaluation $[\![p^*]\!]_\nu$:

$$[\![p_{s1s2}]\!]_\nu = \{([s1, x]_{\sim_{M'|_\sigma}}, [s2, y]_{\sim_{M'|_\sigma}}) \mid (x, y) \in (p_{s1s2})_{M'|_\sigma}\}.$$

Applying the alphabet translation α as illustrated in the traces model we obtain:

$$\{([\![\sigma^S(s1), x]\!]_{\hat{\nu}}, [\![\sigma^S(s2), y]\!]_{\hat{\nu}}) \mid (x, y) \in (p_{\sigma^S(s1)\sigma^S(s2)})_{M'}\}$$

We now apply the inverse alphabet translation of the stable failure domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{traces}([\![P]\!]_\nu)$, we obtain:

$$\{(t, X) \mid \exists\, t'.\, (t', t) \in \hat{\alpha}([\![p^*]\!]_{\hat{\nu}}),\ (t', p^{-1}(X)) \in \hat{\alpha}_{\mathcal{F}}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})).$$

Here, we can pull out $\hat{\alpha}$ from the above failure set. This is thanks to $t\ \hat{\alpha}([\![p^*]\!]_{\hat{\nu}})\ t' = \hat{\alpha}(t\ [\![p^*]\!]_{\hat{\nu}}\ t')$ (see proof in the traces model).

Consequently, we obtain $\hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(P[[p]])]\!]_{\hat{\nu}}))$. Thus,

$$\mathit{failures}([\![P[[p]]]\!]_\nu) = \hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(P[[p]])]\!]_{\hat{\nu}})).$$

- **if** φ **then** P **else** Q: We need to prove the following:

$$\begin{array}{rl} & \mathit{failures}([\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu) \\ = & \hat{\alpha}(\mathit{failures}([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}})). \end{array}$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$\mathit{failures}(\textbf{if } [\![\varphi]\!]_\nu \textbf{ then } [\![P]\!]_\nu \textbf{ else } [\![Q]\!]_\nu).$$

We then calculate the failures set:

$$\begin{cases} \mathit{failures}([\![P]\!]_\nu); & [\![\varphi]\!]_\nu \text{ if evaluates to } \mathit{true} \\ \mathit{failures}([\![Q]\!]_\nu); & [\![\varphi]\!]_\nu \text{ if evaluates to } \mathit{false}. \end{cases}$$

We now unfold the definition of the variable evaluation $[\![\varphi]\!]_\nu$:

$$[\![\varphi]\!]_\nu := \begin{cases} \mathit{true} & \text{if } \nu \Vdash \varphi \\ \mathit{false} & \text{if not } \nu \Vdash \varphi \end{cases}$$

In [Rog06] (Theorem 6 – *Generalized satisfaction condition*) proves that:

$$\nu \Vdash \sigma(\varphi) \text{ iff } \hat{\nu} \Vdash \sigma.$$

Applying the alphabet translation α we obtain:

$$\alpha([\![\varphi]\!]_\nu) = [\![\sigma(\varphi)]\!]_{\hat{\nu}} := \begin{cases} \mathit{true} & \text{if } \hat{\nu} \Vdash \varphi \\ \mathit{false} & \text{if not } \hat{\nu} \Vdash \varphi \end{cases}$$

We now apply the inverse alphabet translation of the failures domain $\hat{\alpha}_{\mathcal{F}}$ and using the induction hypothesis on $\mathit{failures}([\![P]\!]_\nu)$ and $\mathit{failures}([\![Q]\!]_\nu)$, we obtain:

$$\begin{cases} \hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(P)]\!]_{\hat{\nu}})); & \text{if } \hat{\nu} \Vdash \varphi \\ \hat{\alpha}_{\mathcal{T}}(\mathit{failures}([\![\rho(Q)]\!]_{\hat{\nu}})); & \text{if not } \hat{\nu} \Vdash \varphi. \end{cases}$$

Pulling out the $\hat{\alpha}$ from the above failures set, we obtain $\hat{\alpha}(\mathit{failures}([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}}))$. Thus,

$$\mathit{failures}([\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu) = \hat{\alpha}(\mathit{failures}([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}})).$$

Failures/Divergences model

For each process operator, the failures component is identical to the one presented for the stable failures model. Here, we illustrate how the proves goes for the divergences component.

- *SKIP* : We need to prove the following:

$$\mathit{divergences}([\![SKIP]\!]_\nu) = \hat{\alpha}_{\mathcal{N}}(\mathit{divergences}([\![\rho(SKIP)]\!]_{\hat{\nu}})).$$

This trivially holds as the divergence set of *SKIP* is the empty set.

- *DIV* : We need to prove the following:

$$divergences([\![DIV]\!]_\nu) = \hat{\alpha}_{\mathcal{N}}(divergences([\![\rho(DIV)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we calculate the divergence set, which is $\mathcal{A}lph(M'|_\sigma)^{*\checkmark}$. Applying the inverse translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$, we obtain $\hat{\alpha}^{*\checkmark}(\mathcal{A}lph(M')^{*\checkmark})$, i.e., $\hat{\alpha}_{\mathcal{N}}(divergences([\![\rho(DIV)]\!]_{\hat{\nu}}))$. Thus, $divergences([\![DIV]\!]_\nu) = \hat{\alpha}_{\mathcal{N}}(divergences([\![\rho(DIV)]\!]_{\hat{\nu}}))$.

- $?x :: s \to P$: We need to prove the following:

$$divergences([\![?x :: s \to P]\!]_\nu) = \hat{\alpha}^*(divergences([\![\rho(?x :: s \to P)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences(?x :: [\![s]\!]_\nu \to [\![P]\!]_{\lambda z.\nu}).$$

We then calculate the divergence set:

$$\{\langle\rangle\} \cup \{\langle a\rangle \frown t \mid t \in divergences([\![P]\!]_{\nu[a/x]}),\ a \in [\![s]\!]_\nu\}.$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_\nu = [s]_{\sim_{M'|_\sigma}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_\sigma}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $divergences([\![P]\!]_\nu)$, we obtain:

$$\{\hat{\alpha}^*(\langle\rangle)\} \cup \{\langle a\rangle \frown t \mid t \in \hat{\alpha}^*(divergences([\![\rho(P)]\!]_{\hat{\nu}[a/x]})),\ a \in \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})\}.$$

Pulling out the $\hat{\alpha}$ from the above divergence set, we obtain: $\hat{\alpha}^*(divergences([\![\rho(?x :: s \to P)]\!]_{\hat{\nu}}))$. Thus, $divergences([\![?x :: X \to P]\!]_\nu = \hat{\alpha}^*(divergences([\![\rho(?x :: s \to P)]\!]_{\hat{\nu}}))$.

- $!x :: s \to P$: The same procedure as in the case of $?x :: s \to P$.

- $P \fatsemi Q$: We need to prove the following:

$$divergences([\![P \fatsemi Q]\!]_\nu) = \hat{\alpha}^*(divergences([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences([\![P]\!]_\nu \fatsemi [\![Q]\!]_\nu).$$

We then calculate the divergence set:

$$\begin{array}{ll} & divergences([\![P]\!]_\nu) \\ \cup & \{s \frown t \mid s \frown \langle\checkmark\rangle \in traces^{\perp}([\![P]\!]_\nu), t \in divergences([\![Q]\!]_\nu)\}. \end{array}$$

Applying the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $traces^{\perp}([\![P]\!]_{\nu})$ and $divergences([\![Q]\!]_{\nu})$, we obtain:

$$\begin{array}{ll} & \hat{\alpha}_{\mathcal{N}}(divergences([\![\rho(P)]\!]_{\hat{\nu}})) \\ \cup & \{s \frown t \mid s \frown \langle\checkmark\rangle \in \hat{\alpha}_{\mathcal{T}}(traces([\![\rho(P)]\!]_{\hat{\nu}})), t \in \hat{\alpha}^*(divergences([\![\rho(Q)]\!]_{\hat{\nu}}))\}. \end{array}$$

Pulling out the $\hat{\alpha}$ from the above divergence set, we obtain:$\hat{\alpha}^*(divergences([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}}))$. Thus, $divergences([\![P \fatsemi Q]\!]_{\nu} = \hat{\alpha}^*(divergences([\![\rho(P \fatsemi Q)]\!]_{\hat{\nu}}))$.

- $P \Box Q$: We need to prove the following:

$$divergences([\![P \Box Q]\!]_{\nu}) = \hat{\alpha}^*(divergences([\![\rho(P \Box Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences([\![P]\!]_{\nu} \Box [\![Q]\!]_{\nu}).$$

We then calculate the divergence set:

$$divergences([\![P]\!]_{\nu}) \cup divergences([\![Q]\!]_{\nu}).$$

Applying the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $divergences([\![P]\!]_{\nu})$ and $divergences([\![Q]\!]_{\nu})$, we obtain:

$$\hat{\alpha}^*(divergences([\![\rho(P)]\!]_{\hat{\nu}})) \cup \hat{\alpha}^*(divergences([\![\rho(Q)]\!]_{\hat{\nu}})).$$

Pulling out the $\hat{\alpha}$ from the above trace set, we obtain $\hat{\alpha}^*(divergences([\![\rho(P \Box Q)]\!]_{\hat{\nu}}))$. Thus, $divergences([\![P \Box Q]\!]_{\nu} = \hat{\alpha}^*(divergences([\![\rho(P \Box Q)]\!]_{\hat{\nu}}))$.

- $P \sqcap Q$: The same procedure as in the case of $P \Box Q$.
- $P \,|[\, s \,]|\, Q$: We need to prove the following:

$$divergences([\![P \,|[\, s \,]|\, Q]\!]_{\nu}) = \hat{\alpha}^*(divergences([\![\rho(P \,|[\, s \,]|\, Q)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences([\![P]\!]_{\nu} \,|[\, [\![s]\!]_{\nu} \,]|\, [\![Q]\!]_{\nu}).$$

We then calculate the divergence set:

$$\begin{array}{ll} \{u \frown v \mid & \exists\, t \in traces^{\perp}([\![P]\!]_{\nu}), q \in traces^{\perp}([\![Q]\!]_{\nu}) \\ & u \in (t \,|[\, [\![s]\!]_{\nu} \,]|\, q) \cap \mathcal{A}lph(M'|_{\sigma})^* \\ & \wedge (t \in divergences([\![P]\!]_{\nu}) \vee q \in divergences([\![Q]\!]_{\nu}))\}. \end{array}$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_{\nu} = [s]_{\sim_{M'|_{\sigma}}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_{\sigma}}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $traces^{\perp}([\![P]\!]_{\nu})$, $divergences([\![P]\!]_{\nu})$ and $divergences([\![Q]\!]_{\nu}$, we obtain:

$$\begin{array}{ll}\{u \frown v \mid & \exists\, t \in \hat{\alpha}_{\mathcal{T}}(traces^{\perp}([\![\rho(P)]\!]_{\hat{\nu}})), q \in \hat{\alpha}_{\mathcal{T}}(traces^{\perp}([\![\rho(Q)]\!]_{\hat{\nu}})) \\ & u \in (t\,|[\,\hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})\,]|\,q) \cap \hat{\alpha}(\mathcal{A}lph(M')^*) \\ & \wedge (t \in \hat{\alpha}^*(divergences([\![\rho(P)]\!]_{\hat{\nu}})) \vee q \in \hat{\alpha}^*(divergences([\![\rho(Q)]\!]_{\hat{\nu}})))\}.\end{array}$$

Here, we can pull out $\hat{\alpha}$ from the above divergence set. This is thanks to $t1\,|[\,\hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})\,]|\,t2 = \hat{\alpha}(t1\,|[\,[\![\sigma^S(s)]\!]_{\hat{\nu}}\,]|\,t2)$ (see proof in the traces model).

Consequently, we obtain $\hat{\alpha}^*(divergences([\![\rho(P\,|[\,s\,]|\,Q)]\!]_{\hat{\nu}}))$. Thus,

$$divergences([\![P\,|[\,s\,]|\,Q]\!]_{\nu}) = \hat{\alpha}^*(divergences([\![\rho(P\,|[\,s\,]|\,Q)]\!]_{\hat{\nu}})).$$

- $P \setminus s$: We need to prove the following:

$$divergences([\![P \setminus s]\!]_{\nu}) = \hat{\alpha}^*(divergences([\![\rho(P \setminus s)]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences([\![P]\!]_{\nu} \setminus [\![s]\!]_{\nu}).$$

We then calculate the divergence set:

$$\begin{array}{ll} & \{(p \setminus [\![s]\!]_{\nu}) \frown t \mid p \in divergences([\![P]\!]_{\nu})\} \\ \bigcup & \{(u \setminus [\![s]\!]_{\nu}) \frown t \mid u \in \mathcal{A}lph(M'|_{\sigma})^w \wedge (u \setminus [\![s]\!]_{\nu}) \text{ finite} \\ & \qquad \wedge \forall\, p < u \,.\, p \in traces^{\perp}([\![P]\!]_{\nu})\}\end{array}$$

We now unfold the definition of the variable evaluation, i.e., $[\![s]\!]_{\nu} = [s]_{\sim_{M'|_{\sigma}}}$. Applying the alphabet translation α we obtain $\alpha([s]_{\sim_{M'|_{\sigma}}}) = [\sigma^S(s)]_{M'}$, which is equal to $[\![\sigma^S(s)]\!]_{\hat{\nu}}$.

We now apply the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $traces^{\perp}([\![P]\!]_{\nu})$ and $divergences([\![P]\!]_{\nu})$, we obtain:

$$\begin{array}{ll} & \{(p \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})) \frown t \mid p \in \hat{\alpha}^*(divergences([\![\rho(P)]\!]_{\hat{\nu}}))\} \\ \bigcup & \{(u \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})) \frown t \mid u \in \hat{\alpha}(\mathcal{A}lph(M')^w) \wedge (u \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}})) \text{ finite} \\ & \qquad \wedge \forall\, p < u \,.\, p \in \hat{\alpha}_{\mathcal{T}}(traces^{\perp}([\![\rho(P)]\!]_{\hat{\nu}}))\}\end{array}$$

Here, we can pull out $\hat{\alpha}$ from the above divergence set. This is thanks to $\hat{\alpha}(t) \setminus \hat{\alpha}([\![\sigma^S(s)]\!]_{\hat{\nu}}) = \hat{\alpha}(t \setminus [\![\sigma^S(s)]\!]_{\hat{\nu}})$.

Consequently we obtain $\hat{\alpha}^*(divergences([\![\rho(P \setminus s)]\!]_{\hat{\nu}}))$. Thus, $divergences([\![P \setminus s]\!]_{\nu}) = \hat{\alpha}^*(divergences([\![\rho(P \setminus s)]\!]_{\hat{\nu}}))$.

- $P[[p]]$ We need to prove the following:

$$divergences([\![P[[p]]]\!]_\nu) = \hat{\alpha}^*(divergences([\![\rho(P[[p]])]\!]_{\hat{\nu}})).$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences([\![P]\!]_\nu[[[\![p]\!]_\nu]]).$$

We then calculate the divergence set:

$$\{t \mid \exists\, s \in divergences([\![P]\!]_\nu)\,.\, s[\![p^*]\!]_\nu t\}.$$

We now unfold the definition of the variable evaluation $[\![p^*]\!]_\nu$:

$$[\![p_{s1s2}]\!]_\nu = \{([s1,x]_{\sim_{M'|_\sigma}}, [s2,x]_{\sim_{M'|_\sigma}}) \mid (x,y) \in (p_{s1s2})_{M'|_\sigma}\}.$$

Applying the alphabet translation α we obtain:

$$\{([\![\sigma^S(s1),x]\!]_{\hat{\nu}}, [\![\sigma^S(s2),y]\!]_{\hat{\nu}}) \mid (x,y) \in (p_{\sigma^S(s1)\sigma^S(s2)})_{M'}\}$$

We now apply the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $divergences([\![P]\!]_\nu)$, we obtain:

$$\{t \mid \exists\, s \in \hat{\alpha}^*(divergences([\![\rho(P)]\!]_{\hat{\nu}})\,.\, s\ \hat{\alpha}([\![p^*]\!]_{\hat{\nu}})\ t\}.$$

Here, we can pull out $\hat{\alpha}$ from the above divergence set. This is thanks to $\hat{\alpha}(s)\ \hat{\alpha}([\![p^*]\!]_{\hat{\nu}})\ \hat{\alpha}(t) = \hat{\alpha}(s\ [\![p^*]\!]_{\hat{\nu}}\ t)$.

Consequently we obtain $\hat{\alpha}_{\mathcal{N}}(divergences([\![\rho(P[[p]])]\!]_{\hat{\nu}}))$. Thus, $divergences([\![P[[p]]]\!]_\nu) = \hat{\alpha}^*(divergences([\![\rho(P[[p]])]\!]_{\hat{\nu}}))$.

- **if** φ **then** P **else** Q: We need to prove the following:

$$\begin{array}{rl} & divergences([\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu) \\ = & \hat{\alpha}^*(divergences([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}})). \end{array}$$

We unfold the left hand side of the equation. Here, we first apply the evaluation according to CASL, and we obtain:

$$divergences(\textbf{if } [\![\varphi]\!]_\nu \textbf{ then } [\![P]\!]_\nu \textbf{ else } [\![Q]\!]_\nu).$$

We then calculate the divergence set:

$$\begin{cases} divergences([\![P]\!]_\nu); & [\![\varphi]\!]_\nu \text{ if evaluates to } true \\ divergences([\![Q]\!]_\nu); & [\![\varphi]\!]_\nu \text{ if evaluates to } false. \end{cases}$$

We now unfold the definition of the variable evaluation $[\![\varphi]\!]_\nu$:

$$[\![\varphi]\!]_\nu := \begin{cases} true & \text{if } \nu \Vdash \varphi \\ false & \text{if not } \nu \Vdash \varphi \end{cases}$$

In [Rog06] (Theorem 6 – *Generalized satisfaction condition*) proves that:

$$\nu \Vdash \sigma(\varphi) \text{ iff } \hat{\nu} \Vdash \sigma.$$

Applying the alphabet translation α we obtain:

$$\alpha([\![\varphi]\!]_\nu) = [\![\sigma(\varphi)]\!]_{\hat{\nu}} := \begin{cases} \mathit{true} & \text{if } \hat{\nu} \Vdash \varphi \\ \mathit{false} & \text{if not } \hat{\nu} \Vdash \varphi \end{cases}$$

We now apply the inverse alphabet translation of the failures/divergences domain $\hat{\alpha}_{\mathcal{N}}$ and using the induction hypothesis on $\mathit{divergences}([\![P]\!]_\nu)$ and $\mathit{divergences}([\![Q]\!]_\nu)$, we obtain:

$$\begin{cases} \hat{\alpha}^*(\mathit{divergences}([\![\rho(P)]\!]_{\hat{\nu}})); & \text{if } \hat{\nu} \Vdash \varphi \\ \hat{\alpha}^*(\mathit{divergences}([\![\rho(Q)]\!]_{\hat{\nu}})); & \text{if not } \hat{\nu} \Vdash \varphi. \end{cases}$$

Pulling out the $\hat{\alpha}$ from the above divergence set, we obtain $\hat{\alpha}^*(\mathit{divergences}([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}}))$. Thus, $\mathit{divergences}([\![\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q]\!]_\nu) = \hat{\alpha}^*(\mathit{divergences}([\![\rho(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q)]\!]_{\hat{\nu}}))$.

■

A.2 Binary calculator refinement proof

The following is the Isabelle proof script for the BCALC refinement proof. Specifically we prove, BCALC0 $\rightsquigarrow_{\mathcal{F}}$ BCALC3.

```
theorem BCalcRefinement1: "BCalc0 <=F BCalc3"
        (*  unfolding *)
  apply(unfold BCalc0_def BCalc3_def)
        (* cms fixed point induction *)
  apply (rule cspF_fp_induct_left[of _"BCalc3_to_BCalc0"])
        (* simplification *)
  apply simp_all
  apply simp
        (* induction over process names *)
  apply(induct_tac p)
       (* unwinding process names *)
  apply(cspF_auto | auto)
       (* |~|-refinement law -- choose first left branch of the |~| choice *)
  apply(rule cspF_Int_choice_left1)
       (* process rewriting and decomposition *)
  apply(rule cspF_rw_right)
  apply(rule cspF_decompo)
  apply simp
  apply(cspF_auto | auto | cspF_hsf | rule cspF_decompo)+
       (* |~|-refinement law -- choose second left branch of the |~| choice *)
  apply(rule cspF_Int_choice_left2)
  apply (rule cspF_rw_right)
  apply(rule cspF_decompo)
  apply simp
  ...
```

```
  (* repeat the |~|-refinement law 3 times *)
  ...
done
```

A.3 Coloring a test case in CSP-CASL-PROVER

In this section we report proofs from the CSP-CASL based testing chapter. In particular we report the proof script from the coloring of test case of the binary calculator example (see Section 9.3) We now show how the four lemmas are proved.

- `Parallel_one`:

```
lemma Parallel_one: "P3 || T1 =T T1"
   apply(unfold P3_def T1_def)
   apply(cspF_auto | auto simp add: csp_prefix_ss_def
                    image_iff inj_on_def)+
   apply(auto simp add: lifting1 Ax1)
   apply(cspF_auto | auto)+
done

lemma Parallel_one_Button:
       "(Button?x -> P(Button x)) || (Button y -> Q)
                    =T
        Button y -> (P(Button y) || Q)"
  by(cspT_auto | auto simp add: csp_prefix_ss_def
               image_iff inj_on_def )+

lemma Parallel_one_Display:
        "Display x -> P || Display x -> Q
    =T  Display x -> ( P || Q)"
  by(cspT_auto | auto )+

lemma Parallel_one_Stop:"SKIP || STOP  =T STOP" by(cspT_auto | auto)+
```

- `Renaming`:

```
lemma Renaming :
   "T1 [[MyRenaming]]  =T  a -> a -> a -> a -> STOP"
     apply(simp add: T1_def)
     apply(cspT_simp Rename_Button | auto)+
     apply(cspT_simp Rename_Display | auto)+
     apply(cspT_simp Rename_Button | auto)+
     apply(cspT_simp Rename_Display | auto)+
     apply(cspT_simp Rename_STOP | auto)+
done

lemma Rename_Display : "((Display z) -> P) [[MyRenaming]]
                    =T a -> (P [[MyRenaming]]) "
     by(simp add: MyRenaming_def | cspT_auto | auto)+

lemma Rename_Button : "((Button z) -> P) [[MyRenaming]]
                    =T a -> (P [[MyRenaming]])"
```

```
     by(simp add: MyRenaming_def | cspT_auto | auto)+

lemma Rename_STOP : "STOP [[MyRenaming]] =T STOP"
by(simp add: MyRenaming_def | cspT_auto | auto)+
```

- `Parallel_two`:

```
lemma Parallel_two :
   "a->a->a->a->STOP |[{a}]| a->a->a->a->OK-> STOP"
 =T a->a->a->a->OK-> STOP"
  apply(cspT_simp Parallel_with_a | auto)+
  by(cspT_simp Parallel_with_OK | auto)

lemma Parallel_with_a :
 "(a -> P) |[{a}]| a -> Q  =T  a -> (P |[{a}]| Q)" by(cspF_auto | auto)+

lemma Parallel_with_OK :
 "STOP |[{a}]| OK -> STOP =T OK -> STOP" by(cspF_auto | auto)+
```

In order to prove the second parallel execution, we consider two basic generalized parallel which synchronize on the event *'a'*. In lemma *Parallel_with_a* we use the process name *'P'* as a variable, which later can be instantiated with an arbitrary process.

- `Hiding`:

```
lemma Hiding: "(a->a->a->a->OK->STOP) -- {a}  =T OK -> STOP"
  apply(cspT_simp Hide_a | auto)+
  apply(cspT_simp Hide_OK | auto)
  by(cspT_simp hide_STOP | auto)+

lemma Hide_a:"(a -> P) -- {a}  =T  P -- {a}" by(cspT_auto | auto)+

lemma Hide_OK:"(OK -> P) -- {a} =T  OK -> (P -- {a})" by(cspT_auto | auto)+

lemma Hide_STOP:"STOP  -- {a}  =T STOP" by(cspT_auto | auto)+
```

In order to prove the hiding lemma, we consider three basic cases: the hiding of *'a'*, the hiding of *'OK'* and hiding applied to *STOP*. In hide *'a'* we use the process name *'P'* as a variable, which later can be instantiated with an arbitrary process.

(APPENDIX ...*B*)

Modelling ROLLS-ROYCE BR725 starting system

In this appendix we report the full CSP-M specification of the ROLLS-ROYCE BR725 starting system. Specifically we report the specification for the normal (automatic) ground start functionality. Figure 2 illustrate the activity diagram for this specific functionality.

B.4 Normal (automatic) ground start

```
-- ***************** Cockpit  Buttons  **********************
channel press, release

-- general purpose buttons

ButtonOFF = press -> ButtonON
ButtonON = release -> ButtonOFF

-- Instantiate the general ButtonON ButtonOFF for the individual buttons:

channel mc_press, mc_release
MasterCrank = ButtonOFF [[press <- mc_press, release <- mc_release]]

channel ms_press, ms_release
MasterStart = ButtonOFF [[press <- ms_press, release <- ms_release]]

channel engineStartON
EngineStart = engineStartON -> EngineStart

channel fc_press, fc_release
FuelControl = ButtonOFF [[press <- fc_press, release <- fc_release]]

channel ci_press, ci_release
ContIgnition = ButtonOFF [[press <- ci_press, release <- ci_release]]

-- ***********  All Buttons *************
```

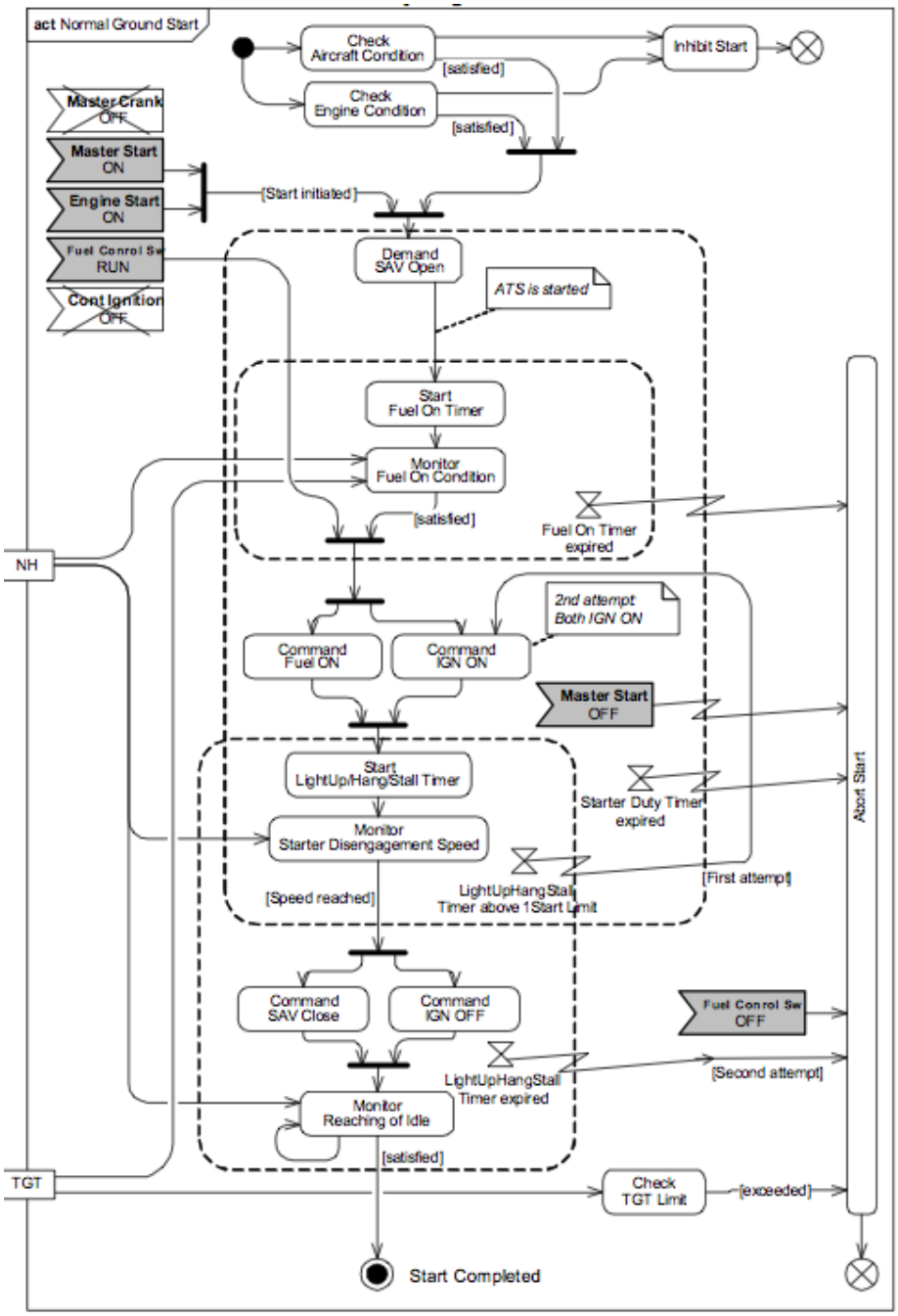

Figure 2: Activity diagram for normal ground start

```
Buttons =     MasterCrank ||| MasterStart
          ||| EngineStart ||| FuelControl
          ||| ContIgnition

-- ******* Checking for the aircraft and engine condition *******
```

```
channel aircraftCondition:Bool
channel engineCondition:Bool
channel inhibitStart, startOK

CheckConditions =   aircraftCondition ? ac -> engineCondition ? ec
                    -> Checking(ac,ec)
                 [] engineCondition ? ec -> aircraftCondition ? ac
                    -> Checking(ac,ec)

Checking(ac,ec) = if (ac and ec)
                     then startOK -> SKIP
                  else InhibitStart

InhibitStart = inhibitStart -> Idle

--********* IDLE Process *************

channel idle
Idle = idle -> Idle

--********** Monitoring the Fuel Condition ***************

datatype SAVMode = open | close
channel sav:SAVMode

-- event indicating the start of the fuel on timer
channel fuelOnTimer

-- the fuel condition depends on the NH and the TGT values
channel readNH:{0..100}

-- TGT ranges between 1200 to 80
channel readTGT:{-80..1200}

channel fuelOnCondition: Bool
channel fuelCondSat

-- NH value for the fuel on condition monitor
NHFuelMonitor = 20

-- TGT value for the fuel on condition monitor
TGTFuelMonitor = 60

--**************** Light Up and schedule fuel ********

datatype Mode = ON | OFF

channel commandIgn : Mode.{1..2}
channel commandFuelON

--********* Monitor Starter disengagement Speed ***********

channel lhsTimer  -- lightUp, hang and stall timer
SPEED1 = 15 -- this is the percentage of the max speed
```

```
SPEED2 = 65 -- this is the percentage of the max speed

--************** Error Handling ********************

InterruptMS = ms_release -> AbortStart
InterruptFC = fc_release -> AbortStart
AbortStart = abort -> Idle

--***********  Time out Handling ***********

channel fuelTimerExp, lightUpTimerExp, startDutyExp, abort
channel ignAttempt:{1..2}

FuelTimer = fuelTimerExp -> AbortStart

LightUpTimer = ignAttempt ? x -> (if(x==1)
                                    then LightUpBox
                                  else lightUpTimerExp -> AbortStart)

StarterDutyTimer = startDutyExp -> AbortStart


--***************** Successful Starting **********************

-- event to indicate the successful start of the engine
channel started

StartCompleted = started -> Idle [] engineStartON -> StartCompleted


-- ****** Initiating Interaction with the EEC ***********

StartInteractionEEC =   ms_press -> NowESpress
                    [] engineStartON -> NowMSpress

NowESpress = engineStartON -> InitStartOK
NowMSpress =  ms_press -> InitStartOK

InitStartOK = startOK -> SKIP

--***************************************
--  Main Process of the Normal Ground Start
--***************************************

NormalGroundStart = (((CheckConditions [| {|startOK|} |] StartInteractionEEC)
                      ) \ { startOK }; FuelBox
                    )  /\ InterruptFC
                  [| { mc_press, mc_release,ms_press, ms_release, engineStartON,
                       fc_press, fc_release, ci_press, ci_release}
                  |] Buttons

--********** Monitoring the Fuel Condition **************

FuelBox = sav.open -> fuelOnTimer -> MonitorFuel
```

```
        [] engineStartON -> FuelBox
        [] FuelTimer
        [] StarterDutyTimer
        [] InterruptMS

MonitorFuel = readNH ? x -> readTGT ? y ->
                    if (x>NHFuelMonitor and y>TGTFuelMonitor)
                       then fuelOnCondition.true -> FuelFlow
                    else fuelOnCondition.false -> MonitorFuel
        [] FuelTimer
        [] StarterDutyTimer
        [] InterruptMS

SynchFuel = fuelCondSat -> SKIP

Fuel = fc_press -> fuelCondSat -> SKIP

FuelFlow = (((Fuel [|{|fuelCondSat|}|] SynchFuel) \ {fuelCondSat}
            ) ; LightUpBox)
         [] engineStartON -> Fuel
         [] FuelTimer
         [] StarterDutyTimer
         [] InterruptMS

--********************* Light Up and schedule fuel ******************

datatype Mode = ON | OFF

channel commandIgn : Mode.{1..2}
channel commandFuelON


LightUpBox =  commandFuelON -> NowCommandIgnOn
          []  commandIgn.ON.1 -> NowFuelOn
          []  engineStartON -> LightUpBox
          []  LightUpTimer
          []  StarterDutyTimer
          [] InterruptMS

NowCommandIgnOn = commandIgn.ON.1 -> MonitorSpeedBox
                 [] engineStartON -> NowCommandIgnOn
                 []  LightUpTimer
                 []  StarterDutyTimer
                 [] InterruptMS

NowFuelOn = commandFuelON -> MonitorSpeedBox
         [] engineStartON -> NowFuelOn
         []  LightUpTimer
         []  StarterDutyTimer
         [] InterruptMS

--************ Monitor Starter disengagement Speed **************
MonitorSpeedBox =  LHSTimer -> MasterSpeed
                [] LightUpTimer
```

```
                [] StarterDutyTimer
                [] InterruptMS

MasterSpeed = readNH ? x -> if (x>SPEED1)
                               then SpeedReached
                            else MasterSpeed

SpeedReached = sav.close -> NowIgnOFF
             [] commandIgn.OFF.1 -> NowSAV
             [] engineStartON -> SpeedReached
             [] LightUpTimer
             [] StarterDutyTimer
             [] InterruptMS

NowIgnOFF = commandIgn.OFF.1 -> MasterIdle
          [] engineStartON -> SpeedReached
          [] LightUpTimer
          [] StarterDutyTimer

NowSAV = sav.close -> MasterIdle
       [] engineStartON -> SpeedReached
       [] LightUpTimer
       [] StarterDutyTimer

MasterIdle = (readNH ? x -> if (x>SPEED2)
                               then StartCompleted
                             else MasterIdle)
           [] LightUpTimer
--****************** End  **********************
```

(APPENDIX ...C)

Modelling and testing of EP2 in CSP-CASL

In this appendix we report the modeling and testing of EP2 in CSP-CASL.

C.5 Modelling EP2 in CSP-CASL

The following is a complete specification in CSP-CASL of the *get configuration* functionality of EP2.

```
library GetConfiguration
logic CASL

%%=============================================
%%          DATA SPECIFICATION
%%=============================================

%%—— Arch level ———

spec D_Arch_GetConfig =
 sort D_SI_Config
end

%%——— ACL ———

spec D_ACL_GetConfig =
    sorts SessionStart, SessionEnd, ConfigRequest, ConfigResponse,
          ConfigNotif, ConfigAck, TerminalClearNotif, TerminalClearAck,
          RemoveConfigNotif, RemoveConfigAck, ActivateConfigNotif,
          ActivateConfigAck < D_SI_Config
    forall x:SessionEnd; y:ConfigRequest . not (x=y)
    forall x:SessionEnd; y:ConfigNotif . not (x=y)
    forall x:SessionEnd; y:TerminalClearNotif . not (x=y)
    forall x:SessionEnd; y:RemoveConfigNotif . not (x=y)
    forall x:SessionEnd; y:ActivateConfigNotif . not (x=y)
    forall x:ConfigRequest; y:ConfigNotif . not (x=y)
    forall x:ConfigRequest; y:TerminalClearNotif. not (x=y)
    forall x:ConfigRequest; y:RemoveConfigNotif . not (x=y)
    forall x:ConfigRequest;y:ActivateConfigNotif . not (x=y)
```

```
    forall x:ConfigNotif; y:RemoveConfigNotif . not (x=y)
    forall x:ConfigNotif; y:TerminalClearNotif . not (x=y)
    forall x:ConfigNotif; y:ActivateConfigNotif . not (x=y)
    forall x:TerminalClearNotif; y:RemoveConfigNotif. not (x=y)
    forall x:TerminalClearNotif;  y: ActivateConfigNotif. not (x=y)
    forall x:RemoveConfigNotif; y: ActivateConfigNotif . not (x=y)
end


%%———————— Pair and Maybe ————————————

spec Pair [sort S] [sort T] =
  sort Pair[S, T]
  ops pair: S * T -> Pair[S, T];
      first : Pair[S, T] -> S;
      second : Pair[S, T] -> T;

  axiom forall p:Pair[S, T] . exists s:S;t:T . pair(s,t) = p
        forall s1,s2:S; t:T . pair(s1,t) = pair(s2,t) => s1 = s2;
        forall s:S; t1,t2:T . pair(s,t1) = pair(s,t2) => t1 = t2;
        forall s:S; t:T . first (pair(s,t)) = s;
        forall s:S; t:T . second (pair(s,t)) = t
end

spec Maybe [sort S] = %mono
  sort Maybe[S]
  ops nothing : Maybe[S];
      just : S -> Maybe[S];
      getJust: Maybe[S] ->? S
  pred defined: Maybe[S]
  axiom not(def(getJust(nothing)))
  axiom forall x:S . (getJust(just(x))) = x
  axiom forall x:Maybe[S] . defined(x) <=> def(getJust(x))
end


%%—— CCL —

spec D_CCL_GetConfig =
  Pair[sort State fit sort S |-> State]
      [sort Trigger fit sort T |-> Trigger]
  and Maybe[sort ACD] and Maybe[sort AISD] and Maybe[sort CAD]
  and Maybe[sort CPTD] and Maybe[sort CAD] and Maybe[sort TACD]
  and Maybe[sort TCD] and Maybe[sort AcqID] and Maybe[sort AID]
  and Maybe[sort RID] and D_ACL_GetConfig
    then
       {sorts AcquirerID, AID, RID, TerminalRangeID ,TerminalUnitID,
              ServiceCenterID, Time, Date
        free type  ConfigObj ::= ACD | AISD | CPTD | CAD | TACD | TCD
        free type TerminalID ::= terID(range:TerminalRangeID; un:TerminalUnitID)
        free type ConfigDownloadMode ::= 0 | 1

      ops get_TerminalID: State ->? TerminalID;
          get_ServiceCenterID: State ->? ServiceCenterID;
```

```
  get_AISD: State ->? AISD;
  set_AISD: State * AISD -> State;
  get_TCD:  State ->? TCD;
  set_TCD:  State * TCD -> State

  vars s:State; aisd:AISD; tcd:TCD
  . get_AISD(set_AISD(s,aisd)) = aisd
  . get_AISD(set_TCD(s,tcd)) = get_AISD(s)
  . get_TCD(set_AISD(s,aisd)) = get_TCD(s)
  . get_TCD(set_TCD(s,tcd)) = tcd


free type ConfigRequest ::=
  mk_ConfigRequest(
    get_AcqID: Maybe[AcqID];
    get_AID:   Maybe[AID];
    get_req:   ConfigObj;
    get_RID:   Maybe[RID];
    get_SCID:  ServiceCenterID;
    get_TrmID: TerminalID
  )

axiom forall cdr: ConfigRequest . get_req(cdr) = ACD =>
  defined(get_AcqID(cdr)) /\
  not(defined(get_AID(cdr))) /\
  not(defined(get_RID(cdr))) %[ACD_Arguments]%

axiom forall cdr: ConfigRequest . get_req(cdr) = AISD =>
  defined(get_AcqID(cdr)) /\
  not(defined(get_AID(cdr))) /\
  not(defined(get_RID(cdr))) %[AISD_Arguments]%

axiom forall cdr: ConfigRequest . get_req(cdr) = CPTD =>
  not(defined(get_AcqID(cdr))) /\
  defined(get_AID(cdr)) /\
  not(defined(get_RID(cdr))) %[CPTD_Arguments]%

axiom forall cdr: ConfigRequest . get_req(cdr) = CAD =>
  not(defined(get_AcqID(cdr))) /\
  not(defined(get_AID(cdr))) /\
  defined(get_RID(cdr)) %[CAD_Arguments]%

axiom forall cdr: ConfigRequest . get_req(cdr) = TACD =>
  not(defined(get_AcqID(cdr))) /\
  defined(get_AID(cdr)) /\
  not(defined(get_RID(cdr))) %[TACD_Arguments]%

axiom forall cdr: ConfigRequest . get_req(cdr) = TCD =>
  not(defined(get_AcqID(cdr))) /\
  not(defined(get_AID(cdr))) /\
  not(defined(get_RID(cdr))) %[TCD_Arguments]%

type ConfigResponse ::=
  mk_ConfigResponse(
```

```
      get_SCID:  ServiceCenterID;
      get_TrmID: TerminalID;
      get_ACD:   Maybe[ACD];
      get_AISD:  Maybe[AISD];
      get_CAD:   Maybe[CAD];
      get_CPTD:  Maybe[CPTD];
      get_TACD:  Maybe[TACD];
      get_TCD:   Maybe[TCD]
    )

 free type D_SI_Init_SessionEnd ::=
   mk_sessionEnd(
     get_SCID:ServiceCenterID;
     get_TrmID: TerminalID;
     get_LocDate: Date;
     get_LocTime:Time
   )

 free type D_SI_Init_SessionStart ::=
   mk_sessionStart(
     get_SCID: ServiceCenterID;
     get_TrmID: TerminalID;
     get_ConfDlMode: ConfigDownloadMode
   )

free type ConfigNotif::=
  mk_configNotif(
    get_AISD:  Maybe[AISD];
    get_SCID:  ServiceCenterID;
    get_TCD:   Maybe[TCD];
    get_TrmID: TerminalID
  )

 free type ConfigAck ::=
   mk_configAck(
     get_SCID:  ServiceCenterID;
     get_TrmID: TerminalID
   )

free type TerminalClearNotif ::=
  mk_terClearNotif(
    get_SCID:  ServiceCenterID;
    get_TrmID: TerminalID
  )

 free type TerminalClearAck ::=
   mk_terClearNotif(
     get_SCID:  ServiceCenterID;
     get_TrmID: TerminalID
   )

 free type RemoveConfig ::=
   mk_removeConfig(
     get_AID: Maybe[AID];
```

```
    get_RID: Maybe[RID];
    get_AcqID: Maybe[AcqID]
  )

axiom forall rcd:RemoveConfig .
     (defined(get_AID(rcd)) /\ not(defined(get_RID(rcd)))
       /\ not(defined(get_AcqID(rcd))))
  \/ (not(defined(get_AID(rcd))) /\ defined(get_RID(rcd))
       /\ not(defined(get_AcqID(rcd))))
  \/ (not(defined(get_AID(rcd))) /\ not(defined(get_RID(rcd)))
       /\ defined(get_AcqID(rcd)))

free type RemoveConfigNotif ::=
  mk_removeConfigNotif(
    get_SCID: ServiceCenterID;
    get_TrmID: TerminalID;
    get_remcfgdata: RemoveConfig
  )

free type RemoveConfigAck ::=
  mk_removeConfigAck(
    get_SCID:ServiceCenterID;
    get_TrmID: TerminalID
  )

free type ActivateConfigNotif ::=
  mk_actConfNotif(
    get_SCID:ServiceCenterID;
    get_TrmID: TerminalID
  )

free type ActivateConfigAck ::=
  mk_actConfAck(
    get_SCID:ServiceCenterID;
    get_TrmID: TerminalID
  )

ops msg_configResponse: ConfigRequest * State -> ConfigResponse;

axiom forall cdr: ConfigRequest; s:State . get_req(cdr) = ACD =>
  defined(get_ACD(msg_configResponse(cdr,s)))       /\
  not(defined(get_AISD(msg_configResponse(cdr,s)))) /\
  not(defined(get_CAD(msg_configResponse(cdr,s))))  /\
  not(defined(get_CPTD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TACD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TCD(msg_configResponse(cdr,s))))

axiom forall cdr: ConfigRequest; s:State . get_req(cdr) = AISD =>
  not(defined(get_ACD(msg_configResponse(cdr,s))))  /\
  defined(get_AISD(msg_configResponse(cdr,s)))      /\
  not(defined(get_CAD(msg_configResponse(cdr,s))))  /\
  not(defined(get_CPTD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TACD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TCD(msg_configResponse(cdr,s))))
```

```
axiom forall cdr: ConfigRequest; s:State . get_req(cdr) = CPTD =>
  not(defined(get_ACD(msg_configResponse(cdr,s)))  /\
  not(defined(get_AISD(msg_configResponse(cdr,s)))) /\
  defined(get_CAD(msg_configResponse(cdr,s)))       /\
  not(defined(get_CPTD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TACD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TCD(msg_configResponse(cdr,s))))

axiom forall cdr: ConfigRequest; s:State . get_req(cdr) = CAD =>
  not(defined(get_ACD(msg_configResponse(cdr,s)))  /\
  not(defined(get_AISD(msg_configResponse(cdr,s)))) /\
  not(defined(get_CAD(msg_configResponse(cdr,s)))  /\
  defined(get_CPTD(msg_configResponse(cdr,s)))      /\
  not(defined(get_TACD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TCD(msg_configResponse(cdr,s))))

axiom forall cdr: ConfigRequest; s:State . get_req(cdr) = TACD =>
  not(defined(get_ACD(msg_configResponse(cdr,s)))  /\
  not(defined(get_AISD(msg_configResponse(cdr,s)))) /\
  not(defined(get_CAD(msg_configResponse(cdr,s)))  /\
  not(defined(get_CPTD(msg_configResponse(cdr,s)))) /\
  defined(get_TACD(msg_configResponse(cdr,s)))      /\
  not(defined(get_TCD(msg_configResponse(cdr,s))))

axiom forall cdr: ConfigRequest; s:State . get_req(cdr) = TCD =>
  not(defined(get_ACD(msg_configResponse(cdr,s)))  /\
  not(defined(get_AISD(msg_configResponse(cdr,s)))) /\
  not(defined(get_CAD(msg_configResponse(cdr,s)))  /\
  not(defined(get_CPTD(msg_configResponse(cdr,s)))) /\
  not(defined(get_TACD(msg_configResponse(cdr,s)))) /\
  defined(get_TCD(msg_configResponse(cdr,s)))

ops msg_configAck: ConfigNotif * State -> ConfigAck;
    st_configAck:  ConfigNotif * State -> State;

forall cdn:ConfigNotif; s:State .
  get_TrmID(cdn) = get_TerminalID(s) /\
  get_SCID(cdn) = get_ServiceCenterID(s) /\
  defined(get_AISD(cdn)) =>
    st_configAck(cdn,s) = set_AISD(s,getJust(get_AISD(cdn)))

forall cdn:ConfigNotif; s:State .
  get_TrmID(cdn) = get_TerminalID(s) /\
  get_SCID(cdn) = get_ServiceCenterID(s) /\
  defined(get_TCD(cdn)) =>
    st_configAck(cdn,s) = set_TCD(s,getJust(get_TCD(cdn)))

ops msg_terClearNotif : TerminalClearNotif * State -> TerminalClearAck;
    st_terClearNotif : TerminalClearNotif * State -> State

ops msg_removeConfigAck : RemoveConfigNotif * State -> RemoveConfigAck;
    st_removeConfigAck : RemoveConfigNotif * State -> State;
```

```
        ops msg_actConfAck : ActivateConfigNotif * State -> ActivateConfigAck;
            st_actConfAck : ActivateConfigNotif * State -> State;

        ops msg_sessionStartConf : Trigger -> SessionStart
  }
end


%%=====================================
%%   CspCASL SPECIFICATION
%%=====================================

logic CspCASL

%-- Arch --

ccspec Arch_GetConfig =
data D_Arch_GetConfig
channel
   C_SI_Config : D_SI_Config
process
  TerminalConfig: C_SI_Config;
  SC_Config : C_SI_Config;
  TR_Config : C_SI_Config;

  SC_Config = RUN(C_SI_Config)
  TR_Config = RUN(C_SI_Config)
  TerminalConfig = SC_Config [| C_SI_Config |] TR_Config

end

%-- ACL --

ccspec ACL_GetConfig =
data D_ACL_GetConfig
channel
  C_SI_Config: D_SI_Config
process
    TerminalConfig: C_SI_Config;
    Ter_Config: C_SI_Config;
    Ter_ConfMgm: C_SI_Config;
    SC_Config: C_SI_Config;
    SC_ConfMgm: C_SI_Config;

TerminalConfig = Ter_Config [| C_SI_Config |] SC_Config

Ter_Config = C_SI_Config ! sessionStart::SessionStart -> Ter_ConfMgm
Ter_ConfMgm = C_SI_Config ? configMess :: D_SI_Config ->
      (if (configMess in SessionEnd) then
        Ter_Config
      else (if (configMess in  ConfigRequest)
              then C_SI_Config!resp::ConfigResponse -> Ter_ConfMgm
         else (if (configMess in ConfigNotif)
                then C_SI_Config!ack::ConfigAck -> Ter_ConfMgm
```

```
                else (if (configMess in TerminalClearNotif)
                  then C_SI_Config!ackT::TerminalClearAck -> Ter_ConfMgm
                else (if (configMess in  RemoveConfigNotif)
                    then  C_SI_Config!ackR::RemoveConfigAck -> Ter_ConfMgm
                else (if (configMess in ActivateConfigNotif)
                   then   C_SI_Config!ackA::ActivateConfigAck -> Ter_ConfMgm
                else STOP))))))
 SC_Config = C_SI_Config ? sessionStart::SessionStart -> SC_ConfMgm
 SC_ConfMgm = C_SI_Config ! seM::SessionEnd -> SC_Config
      |~|  C_SI_Config ! cdrM :: ConfigRequest
           -> C_SI_Config ? response :: ConfigResponse-> SC_ConfMgm
      |~|  C_SI_Config ! cdnM :: ConfigNotif
           -> C_SI_Config ? confAck :: ConfigAck -> SC_ConfMgm
      |~|  C_SI_Config ! tclearM :: TerminalClearNotif
           -> C_SI_Config ?  tclearAck :: TerminalClearAck -> SC_ConfMgm
      |~|  C_SI_Config ! rcdnM  :: RemoveConfigNotif
           -> C_SI_Config ? rmConfAck :: RemoveConfigAck -> SC_ConfMgm
      |~|  C_SI_Config ! acdnM :: ActivateConfigNotif
           -> C_SI_Config ? acknowledge :: ActivateConfigAck -> SC_ConfMgm
end

%—— CCL ——

 ccspec  CCL_GetConfig =
 data D_CCL_GetConfig
 channel
 C_SI_Config : D_SI_Config
 process
 TerminalConfig(Pair[State, Trigger]): C_SI_Config;
 Ter_Config(Pair[State, Trigger]): C_SI_Config;
 Ter_ConfMgm(Pair[State, Trigger]): C_SI_Config;
 SC_Config: C_SI_Config;
 SC_ConfMgm: C_SI_Config;

 TerminalConfig(p) = Ter_Config(p) [| C_SI_Config |] SC_Config

 Ter_Config(p) = C_SI_Config ! msg_sessionStartConf(second(p))
                 -> Ter_ConfMgm(p)

 Ter_ConfMgm(p) = C_SI_Config ? configMess::D_SI_Config ->
 (if (configMess in SessionEnd) then
    SKIP
  else (
    if (configMess in  ConfigRequest)
      then C_SI_Config!msg_configResponse(configMess as ConfigRequest,
                                          first(p))
           -> Ter_ConfMgm(p)
      else (
        if (configMess in  ConfigNotif)
          then C_SI_Config!msg_configAck(configMess as ConfigNotif,
                                         first(p))
               -> Ter_ConfMgm(pair((st_configAck
                  (configMess as ConfigNotif,first(p))),second(p)))
          else (
```

```
            if (configMess in TerminalClearNotif)
 then C_SI_Config!msg_terClearNotif(configMess as TerminalClearNotif,
                                    first(p))
       -> Ter_ConfMgm(pair((st_terClearNotif
            (configMess as TerminalClearNotif,first(p))),second(p)))
  else (
    if (configMess in RemoveConfigNotif)
      then C_SI_Config!msg_removeConfigAck(configMess as RemoveConfigNotif,
                                           first(p))
             -> Ter_ConfMgm(pair((st_removeConfigAck
                  (configMess as RemoveConfigNotif,first(p))),second(p)))
        else (
           if (configMess in ActivateConfigNotif)
            then C_SI_Config!msg_actConfAck(configMess as ActivateConfigNotif,
                                            first(p))
             -> Ter_ConfMgm(pair((st_actConfAck
                  (configMess as ActivateConfigNotif,first(p))),second(p)))
                else SKIP
                ))))))
SC_Config = C_SI_Config ? sessionStart :: SessionStart -> SC_ConfMgm
SC_ConfMgm = C_SI_Config ! seM :: SessionEnd -> SC_Config
     |~|  C_SI_Config ! cdrM :: ConfigRequest
          -> C_SI_Config ? response :: ConfigResponse-> SC_ConfMgm
     |~|  C_SI_Config ! cdnM :: ConfigNotif
          -> C_SI_Config ? confAck :: ConfigAck -> SC_ConfMgm
     |~|  C_SI_Config ! tclearM :: TerminalClearNotif
          -> C_SI_Config ?  tclearAck :: TerminalClearAck -> SC_ConfMgm
     |~|  C_SI_Config ! rcdnM  :: RemoveConfigNotif
          -> C_SI_Config ? rmConfAck :: RemoveConfigAck -> SC_ConfMgm
     |~|  C_SI_Config ! acdnM :: ActivateConfigNotif
          -> C_SI_Config ? acknowledge :: ActivateConfigAck -> SC_ConfMgm
end
```

C.6 Test verdict generated by TEV

In the following we report the complete test verdict generated by TEV for the test case T_2.

We recall that the test case T_2 experiments a communication between the acquirer and the terminal in the context of *payment transaction*. Here, the terminal sends a message to the acquirer to authorize a payment for a purchased goods. The acquirer authorize the transaction by sending a message of type *AuthResponse*. This is a required behavior of the system, as specified in the EP2 terminal (and acquirer) book [Con08].

Test Information

- result = resources/TestVerdict/PaymentTestResult.xml
- description = Test Case 7 - Authorization for process transaction
- name = Process TransactionTest

CSP-CASL Test Case

T7

```
T7 =    C_FE_FrontEnd ! authreq:D_FE_FrontEnd_AuthReq
          -> C_FE_FrontEnd ! authresp: D_FE_FrontEnd_AuthRes
          -> STOP
```

PCO and TimeOut Information

- pcoFile = resources/PCO/PCO.xml
- SpecificationLevel = Abstract Component Level
- SpecificationFile = ACL_ProcessTransaction
- Timeout = 10ms

Test Case Evaluation Information

- ep2Dialogue = ProcessTransaction
- color = GREEN
- colorProofDir = resources/ColorProof/GREEN/T8

EP2 Components

Acquirer

- namespace = http://www.eftpos2000.ch
- templatePath = resources/template/FEFrontEnd/
- serverIp = 192.168.1.1
- encoding = 3
- AcqID = 00000000004
- prefix = ep2
- interfaceName = FEFrontEnd
- serverPort = 6625
- name = Acquirer
- communicationMode = server

Terminal

- namespace = http://www.eftpos2000.ch
- templatePath = resources/template/FEFrontEnd/
- encoding = 3
- TrmID = TERM1234
- prefix = ep2
- interfaceName = FEFrontEnd
- port = 6625
- ip = 192.168.1.2
- name = cCredit Terminal

Test Sequence

(source = Acquirer) ----------> (target = cCredit Terminal)

Conversation 1

receive message (Type: authreq)

```
<?xml version="1.0" encoding="UTF-8"?>
<ep2:message xmlns:ep2="http://www.eftpos2000.ch" specversion="0400">
  <ep2:authreq msgnum="2910">
    <ep2:AcqID>00000000004</ep2:AcqID>
    <ep2:TrmID>TERM1234</ep2:TrmID>
    <ep2:TrxDate>20100224</ep2:TrxDate>
    <ep2:TrxTime>015553</ep2:TrxTime>
    <ep2:TrxSeqCnt>24546</ep2:TrxSeqCnt>
    <ep2:AmtAuth>50</ep2:AmtAuth>
    <ep2:TrxCurrC>756</ep2:TrxCurrC>
    <ep2:Track2Dat>CVeTmEkE8d7NO2FpLCCCTQ==</ep2:Track2Dat>
    <ep2:TVR>AAAAgAA=</ep2:TVR>
    <ep2:CVMRes>HgAA</ep2:CVMRes>
    <ep2:POSEntry>90</ep2:POSEntry>
    <ep2:TrxTypeExt>3</ep2:TrxTypeExt>
    <ep2:AID>oAAAAVcAIA==</ep2:AID>
  </ep2:authreq>
</ep2:message>
```

send message (Type: authrsp)

```
<?xml version="1.0" encoding="UTF-8"?>
<ep2:message xmlns:ep2="http://www.eftpos2000.ch" specversion="0400">
  <ep2:authrsp msgnum="2911">
    <ep2:AcqID>00000000004</ep2:AcqID>
    <ep2:AmtAuth>50</ep2:AmtAuth>
    <ep2:AuthC>009646</ep2:AuthC>
    <ep2:TrxSeqCnt>24546</ep2:TrxSeqCnt>
    <ep2:AuthRespC>00</ep2:AuthRespC>
    <ep2:AuthReslt>0</ep2:AuthReslt>
    <ep2:TrmID>TERM1234</ep2:TrmID>
  </ep2:authrsp>
</ep2:message>
```

Test Analysis Between Expected Message and Received Message:
MATCH
Difference(s) : 3

1 Expected text value '20100223' but was '20100224'
2 Expected text value '130842' but was '015553'
3 Expected text value 'OZYbmsV8ODZ3EC3vY4z9yA==' but was 'CVeTmEkE8d7NO2FpLCCCTQ=='

ON THE FLY TEST VERDICT

On the fly test verdict result:
PASS

Timeout Information:
NO-TIMEOUT

References

[abb] Abbot Java Gui Test Framework. `http://abbot.sourceforge.net`.

[ABK+02] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286(2), 2002.

[ABR99] Egidio Astesiano, Manfred Broy, and Gianna Reggio. Algebraic specification of concurrent systems. In Egidio Astesiano, Hans-Joerg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*. Springer, 1999.

[Abr03] Jean-Raymond Abrial. B#: Toward a synthesis between Z and B. *ZB 2003: Formal Specification and Development in Z and B*, pages 629–629, 2003.

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[AJS05] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *LNCS*. Springer, 2005.

[AR01] Egidio Astesiano and Gianna Reggio. Labelled transition logic: an outline. *Acta Informatica*, 37:831–879, 2001.

[AS02] David Aspinall and Donald Sannella. From specifications to code in CASL. *Algebraic Methodology and Software Technology*, pages 11–40, 2002.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1):25–59, 1988.

[BBC+02] Jonathan P. Bowen, Kirill Bogdanov, John Clark, Mark Harman, Robert Hierons, and Paul Krause. FORTEST: Formal methods and testing. In *Proceedings of 26th Annual International Computer Software and Applications Conference (COMPSAC 02*, pages 91–101. IEEE Computer Society Press, 2002.

[BBP96] Stéphane Barbey, Didier Buchs, and Cécile Péraire. A theory of specification-based testing for object-oriented software. In *EDCC-2*, pages 303–320, London, UK, 1996. Springer-Verlag.

[BCFG86] Luc Bougé, N. Choquet, Laurent Fribourg, and Marie-Claude-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Systems and Software*, 6(4):343–360, 1986.

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering*, 6(6):387–405, 1991.

[BH05] Michel Bidoit and Rolf Hennicker. Externalized and internalized notions of behavioral refinement. *Theoretical Aspects of Computing –ICTAC 2005*, pages 334–350, 2005.

[BHK89] Jan A. Bergstra, Jan Heering, and Paul Klint. The algebraic specification formalism ASF. *Algebraic specification*, 1989.

[BHT97] Ed Brinksma, Lex Heering, and Jan Tretmans. Developments in testing transition systems. In *Workshop on Testing Communicating Systems*, pages 143–166. Chapman & Hall, 1997.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., 2005.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BKS97] Bettina Buth, Michel Kouvaras, and Hui Shi. Deadlock analysis for a fault-tolerant system. In *AMAST'97*, LNCS 1349. Springer, 1997.

[BM04] Michel Bidoit and Peter D. Mosses. *CASL User Manual*, volume 2900 of *LNCS*. Springer, 2004.

[Bri88] Ed Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification*, VIII(63–74), 1988.

[Bri99] Ed Brinksma. Formal methods for conformance testing: Theory can be practical. *Computer Aided Verification*, pages 687–687, 1999.

[BS99] Bettina Buth and Mike Schrönen. Model-checking the architectural design of a fail-safe communication system for railway interlocking systems. In *FM'99*, LNCS 1709. Springer, 1999.

[BSS87] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS Specifications,

their Implementations, and their Tests. In *Protocol Specification, Testing and Verification*, pages 349–360. Elsevier, 1987.

[BST06] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Observational interpretation of CASL specifications. Research Report LSV-06-16, Laboratoire Spécification et Vérification, ENS Cachan, France, 2006.

[BST08] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Observational interpretation of CASL specifications. *Mathematical Structures in Computer Science*, 18(2), 2008.

[BT01] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. *Modeling and Verification of Parallel Processes*, pages 187–195, 2001.

[But99] Michael J. Butler. CSP2B: A practical approach to combining CSP and B. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 490–508. Springer-Verlag, 1999.

[BW90] J. C. M. Beaten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[CEW93] Ingo Classen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic specification techniques and tools for software development: the ACT approach*. World Scientific Publishing Co., Inc., 1993.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *FoSSaCS*, pages 140–155, 1998.

[CG07] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in CSP. *Formal Methods and Software Engineering*, pages 151–170, 2007.

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.

[Chu05] Lim Beng Chuan. Towards hardware-in-a-loop testing for an international standard of an electronic payment system. Master's thesis, University of Wales Swansea, 2005.

[CMU] Software Engineering Institute, Carnegie Mellon. `http://www.sei.cmu.edu`.

[CN01] Paul Clements and Linda M. Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2001.

[Con08] EP2 Consortium. EFT/POS 2000 specification, version 4.0.0, 2008. Project Overview available at http://www.eftpos2000.ch.

[CSW03] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A refinement strategy for Circus. *Formal Aspects of Computing*, 15(2):146–181, 2003.

[CW98] Paul C. Clements and Nelson Weiderman. Second international workshop

on development and evolution of software architectures for product families. Technical Report CMU/SEI-98-SR-003, Carnegie Mellon University, 1998.

[DBBS96] John Derrick, Howard Bowman, Eerke Boiten, and Maarten Steen. Comparing LOTOS and Z refinement relations. In *FORTE/PSTV'96*, pages 501–516. Chapman & Hall, 1996.

[DDH72] Ole Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., 1972.

[DF98] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[DHK96] Arie Van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., 1996.

[Dij02] Edsger W. Dijkstra. Cooperating sequential processes. *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138, 2002.

[Dub00] Oliver Dubuisson. *ASN.1 communication between heterogeneous systems*. ISBN 0-12-6333361-0. Morgan Kaufmann, September 2000.

[Ehr82] Hartmut Ehrig. On the theory of specification, implementation, and parametrization of abstract data types. *J. ACM*, 29(1):206–227, 1982.

[EK99] Hartmut Ehrig and Hans-Joerg Kreowski. Refinement and implementation. In Egidio Astesiano, Hans-Joerg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*. Springer, 1999.

[EKP80] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. Algebraic implementation of Abstract Data Types: Concept, syntax, semantics and correctness. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 142–156. Springer-Verlag, 1980.

[EM85] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*. Springer-Verlag New York, Inc., 1985.

[EO01] Hartmut Ehrig and Fernando Orejas. Dynamic abstract data types: an informal proposal in 1994. *Current trends in theoretical computer science: entering the 21st century*, pages 180–191, 2001.

[Fis98] Clemens Fischer. How to combine Z with process algebra. In *ZUM '98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.

[Fis00] Clemens Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universit'at Oldenburg, 2000.

[Gar96] H. Garavel. An overview of the eucalyptus toolbox. Technical report, University of Maribor, 1996.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96. Springer-Verlag, 1995.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., 1993.

[Gim08] Andy Gimblett. Tool support for CSP-CASL, 2008. MPhil Thesis, Swansea University, 2008.

[GJ99] Marie-Claude Gaudel and Perry R. James. Testing algebraic data types and processes : a unifying theory. *Formal Aspects of Computing*, 1999.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[GP94] Jan Friso Groote and Alban Ponse. Proof theory for μCRL: A language for processes with data. In *Proceedings of the International Workshop on Semantics of Specification Languages (SoSL)*, pages 232–251. Springer-Verlag, 1994.

[GP95] Jan Friso Groote and Alban Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing. Springer, 1995.

[GRS05] Andy Gimblett, Markus Roggenbach, and Bernd-Holger Schlingloff. Towards a formal specification of an electronic payment systems in CSP-CASL. In *WADT'04*, LNCS 3423. Springer, 2005.

[GTW78] Joseph A. Goguen, James W Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. *Current Trends in Programming Methodology*, IV:80–149, 1978.

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge University Press, 1993.

[Har00] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72. ACM, 2000.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[Hoa76] C. A. R. Hoare. Proof of correctness of data representation. In *Language Hierarchies and Interfaces, International Summer School*, pages 183–193, London, UK, 1976. Springer-Verlag.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa96] C. A. R. Hoare. How did software get so reliable without proof? *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 1–17, 1996.

[Hoa06] C. A. R. Hoare. Why ever CSP? *Electronic Notes in Theoretical Computer Science*, 162:209–215, September 2006.

[IR] Yoshinao Isobe and Markus Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.

[IR05] Yoshinao Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440. Springer, 2005.

[IR06] Yoshinao Isobe and Markus Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006*, LNCS 4137. Springer, 2006.

[IR07] Yoshinao Isobe and Markus Roggenbach. Proof Principles of CSP – CSP-Prover in Practice. In Hans-Dietrich Haasis, Hans-Jörg Kreowski, and Bernd Scholz-Reiter, editors, *LDIC 2007*. Springer, 2007.

[IR08] Yoshinao Isobe and Markus Roggenbach. CSP-Prover – a proof tool for the verification of scalable concurrent systems. *Journal of Computer Software, Japan Society for Software Science and Technology*, 25, 2008.

[IRG05] Yoshinao Isobe, Markus Roggenbach, and Stefan Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.

[ISO89] ISO 8807. LOTOS — a formal description technique based on the temporal ordering of observational behaviour, 1989.

[JRvdL00] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software architecture for product families: principles and practice*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[JTC01] JTCI/CS7/WG14. *The E-LOTOS Final Draft International Standard*, 2001.

[KHRS09] Temesghen Kahsai, Greg Holland, Markus Roggenbach, and Bernd-Holger Schlingloff. Towards formal testing of jet engine Rolls-Royce BR725. In *Concurency, Specification and Programming*, pages 217–229, 2009.

[Kli93] Paul. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.

[KR09] Temesghen Kahsai and Markus Roggenbach. Property preserving refinement notions for CSP-CASL. In *WADT 2008*, LNCS 5486, pages 206–210, 2009.

[KRS07] Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff. Specification-based testing for refinement. In Mike Hinchey and Tiziana Margaria, editors, *SEFM 2007*, pages 237–247. IEEE Computer Society, 2007.

[KRS08] Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff. Specification-based testing for software product lines. In *SEFM 2008*, pages 149–159. IEEE Computer Society, 2008.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ML: a gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[Ltd03] Formal Systems (Europe) Ltd. *Process Behaviour Explorer — the ProBE User Manual*. Formal Systems (Europe) Ltd., 2003.

[Ltd06] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — the FDR2 User Manual*. Formal Systems (Europe) Ltd., 2006.

[LY94] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320, 1994.

[Mac99] Patrícia D. L. Machado. On oracles for interpreting test results against algebraic specifications. In *AMAST'99*, LNCS 1548, pages 502–518. Springer, 1999.

[Mac00] Patrícia D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, University Edinburgh, 2000.

[McG01] John D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, Software Engineering Institute, December 2001.

[MD00] Brendan Mahony and Jin Song Dong. Timed communicating Object Z. *IEEE Trans. Softw. Eng.*, 26(2):150–177, 2000.

[MFS+07] Clavel Manuel, Duran Francisco, Eker Steven, Lincoln Patrick, Martí-Oliet Narciso, Meseguer Jose, and Talcott Carolyn. *All about Maude - A High-Performance Logical Framework*, volume XXII. Springer, 2007.

[Mil84] Robin Milner. A complete inference system for a class of regular behaviours. *J. Comput. Syst. Sci.*, 28(3):439–466, 1984.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[MML07] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set, HETS. In *TACAS 2007*, LNCS 4424. Springer, 2007.

[Mos02] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286:367–475, 2002.

[Mos04] P. D. Mosses, editor. CASL *Reference Manual*. LNCS 2960. Springer, 2004.

[MR07] Till Mossakowski and Markus Roggenbach. Structured CSP – A Process Algebra as an Institution. In *WADT 2006*, LNCS 4409, 2007.

[MRS03] Till Mossakowski, Markus Roggenbach, and Luth Schröder. COCASL at work — Modelling Process Algebra. In *Coalgebraic Methods in Computer Science*, volume 82 of *Electronic Notes Theoretical Computer Science*, 2003.

[MST04] Till Mossakowski, Donald Sannella, and Andrzej Tarlecki. A simple refinement language for CASL. In *WADT 2004*, LNCS 3423. Springer, 2004.

[MV90] S. Mauw and G. J. Veltink. A process specification formalism. *Fundam. Inf.*, 13(2):85–139, 1990.

[MV91] Eric Madelaine and Didier Vergamini. Specification and verification of a sliding window protocol in LOTOS. In *Formal Description Techniques, IV, volume C-2 of IFIP Transactions. Elsevier Science Publishers B.V. (North-Holland*, pages 495–510, 1991.

[MV92] Sjouke Mauw and Gert J. Veltink. A proof assistant for PSF. In *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 158–168. Springer-Verlag, 1992.

[NBAR04] Syed Nabi, Mahesh Balike, Jace Allen, and Kevin Rzemien. An overview of hardware-in-the-loop testing systems at Visteon. *SAE Technical Paper Series*, 2004.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[OIR09] Liam O'Reilly, Yoshinao Isobe, and Markus Roggenbach. CSP-CASL-Prover – a generic tool for process and data refinement. *Electronic Notes in Theoretical Computer Science*, 250(2):69–84, 2009.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*, volume XXVI. Springer, 2005.

[Pel96] Jan Peleska. Test automation for safety-critical systems: Industrial application and future developments. *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 39–59, 1996.

[Pel02] Jan Peleska. Hardware/software integration testing for the new airbus aircraft families. In *TestCom '02: Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*. Kluwer, B.V., 2002.

[PM06] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

[PS91] Luís Ferreira Pires and Wanderley Lopes de Souza. Step-wise refinement design example using LOTOS. In *FORTE '90: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 255–262, 1991.

[PSF] PSF toolkit manual pages. `http://staff.science.uva.nl/ psf/`.

[RAC00] Gianna Reggio, Egidio Astesiano, and Christine Choppy. CASL-LTL — a CASL extension for dynamic Reactive Systems — Summary. Technical Report DISI-TR-99-34, Università di Genova, 2000.

[Rog06] Markus Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354, 2006.

[Ros98] A.W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[RR00] Gianna Reggio and Lorenzo Repetto. CASL-CHART: a combination of statecharts and of the algebraic specification language CASL. In *Algebraic Methodology and Software Technology*, volume 1816 of *LNCS*, pages 243–257. Springer, 2000.

[RS02] Markus Roggenbach and Lutz Schröder. Towards trustworthy specifications: Consistency checks. In *WADT'01*, LNCS 2267. Springer, 2002.

[RSG+01] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[SAA01] Gwen Salaün, Michel Allemand, and Christian Attiogbé. A formalism combining CCS and CASL. Technical Report 00.14, University of Nantes, 2001.

[SAA02a] Gwen Salaün, Michel Allemand, and Christian Attiogbé. A method to combine any process algebra with an algebraic specification language: the p-calculus example. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 385–392. IEEE Computer Society, 2002.

[SAA02b] Gwen Salaün, Michel Allemand, and Christian Attiogbé. Specification of an access control system with a formalism combining CCS and CASL. In *Parallel and Distributed Processing*, pages 211–219. IEEE, 2002.

[Sca98] Bryan Scattergood. The Semantics and Implementation of Machine-Readable CSP, 1998. DPhil thesis, University of Oxford.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[SMH99] Bernd-Holger Schlingloff, Oliver Meyer, and Thomas Hülsing. Correctness

analysis of an embedded controller. In *DASIA'99*, esa SP-447, pages 317–325, 1999.

[Smi99] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.

[spa] SPASS: An Automated Theorem Prover for First-Order Logic with Equality. `www.spass-prover.org`.

[SPK99] Hui Shi, Jan Peleska, and Michel Kouvaras. Combining methods for the analysis of a fault-tolerant system. In *AMAST'98*. Springer, 1999.

[SPLa] International Workshop on Software Product Line Testing 2007. `http://www.biglever.com/split2007`.

[SPLb] Software Product Line Conference 2008. http://www.lero.ie/SPLC2008.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9(3):229–269, 1997.

[Sto97] Bill Stoddart. An introduction to the event calculus. *ZUM '97: The Z Formal Specification Notation*, pages 10–34, 1997.

[SWC02] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in Circus. *FME 2002:Formal Methods – Getting IT Right*, pages 1–15, 2002.

[TA97] K. Taguchi and K. Araki. The state-based ccs semantics for concurrent z specification. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, page 283. IEEE Computer Society, 1997.

[Tre92] Jan Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, Haag, The Netherlands, 1992.

[TS99] Helen Treharne and Steve 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. Springer-Verlag, 1999.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[vGG00] Rob van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Inf.*, 37(4-5):229–327, 2000.

[vGW96] Rob van Glabbeek and Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

[WC01] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, 2001.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, LNCS 2272, pages 184—203. Springer-Verlag, 2002.

[WD96] Jim Woodcock and Jim Davies. *Using Z - Specification, Refinement, and Proof*. Prentice Hall, 1996.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), 1971.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36. ACM, 2006.