# Structured specification with processes and data: Theory, tools and applications.

**O'Reilly, Liam**

# Structured Specification with Processes and Data

## Theory, Tools and Applications

Liam O'Reilly

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

2012

# 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 1/8/2012

# 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 1/8/2012

# 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 1/8/2012

# Abstract

The integration of processes and data is a long standing research topic. In this thesis, we study this integration in the context of the language CSP-CASL, where CSP is used to describe processes aspects and CASL is used to describe data aspects.

Our specific questions are: Is it possible to make structuring operations available for building up complex specifications in a compositional way? What is an appropriate notion of refinement in such a setting? Finally, is it possible to reason on such specifications in a modular way? Based on institution theory, we develop a framework for CSP-CASL in which all three questions have positive answers.

We develop CSP-CASL as various institutions (one for each of the main CSP semantics) and a notion of refinement which we show to be healthy and useful. Furthermore, we develop several proof calculi, supporting refinement and deadlock analysis, which allow for compositional reasoning over structured CSP-CASL specifications. An example together with a prototypical implementation demonstrate the practicality of our approach.

Our work has implications beyond the specific setting of CSP-CASL: The equivalences studied are independent of the specific choice of process algebra, and thus can be applied to other settings, for instance, state machine diagrams in UML.

# Acknowledgements

I would like to thank my family for supporting me throughout my doctorate. They often endured long periods of time without my company.

I would like to gratefully acknowledge the contributions of my supervisor Dr. Markus Roggenbach for his continuous guidance, support and mentoring throughout my time as his student. He has always been optimistic and helpful. I would also like to thank my second supervisor, Dr. Monika Seisenberger, and her family for their kind support. Their opinions and guidance have been extremely valuable.

I wish to gratefully acknowledge my examiners, Prof. John V. Tucker and Dr. Grant Malcolm, for devoting their valuable time examining my thesis and providing me with valuable feedback and advice. I also like to extend my sincere thanks to the HETS development team, particularly Christian Maeder, for their support in implementing CSP-CASL in HETS.

I extend my gratitude to the entire *Processes and Data Group* (past and present) and the department's theory group. They have always been enormously supportive and been ready to help out when required. Several special friends must be acknowledged here:

- Phillip James and Matt Gwynne for their ongoing help and encouragement during this work, which often involved frequent recreational activities;
- Fredrik Nordvall Forsberg for checking many proofs in this thesis. Often, he worked as an automated theorem checker;
- Mark New for his time and effort in fighting the odd 'battle of LaTeX' with me; and
- Helen Dodd and Emma Thom (along with the individuals already mentioned above) for their kind efforts in proof reading this thesis.

Thanks must be given to Swansea University's Department of Computer Science for the opportunity to pursue this Ph.D. I also thank both Swansea University and the department for the financial support I have received. I would like to thank all the computer science staff and postgraduate students for making this an extremely enjoyable phase of my life.

Finally, I would like to thank Erwin R. Catesbeiana (jr) for his help in structuring my inconsistent thought processes.

# Table of Contents

*Chapter 1*

# Introduction

**Contents**



The specification and formal development of computer systems is noted as an important and rapidly developing area of computer science [JOW06]. This thesis advances the current state of the art with respect to formal specifications, development and verification of critical systems, in particular reactive systems. This chapter introduces the fields of critical and reactive systems, and the development and verification of such systems. It also illustrates the limitations of current formal development techniques. Finally, we discuss the aims and contributions of this thesis.

## 1.1 Critical Systems and Formal Methods

The field of formal methods has grown out of the need for verification of critical systems. Here, we provide an overview of critical systems and formal methods.

### 1.1.1 Critical Systems

Critical systems are systems for which failure or malfunction is not acceptable [Sto96, Fow09, Som07]. Often these are control systems, also known as *reactive systems*, which manage and maintain conditions in some environment. For instance, electronic payment systems such as those that implement the EP2 standard (short for 'EFT/POS 2000' or more fully 'Electronic Fund Transfer/Point Of Service 2000') [EP208]. EP2 is an industrial standard of an electronic payment system mainly developed by financial organisations. The goal of this specification

is to set the standard for all electronic payment systems, including debit and credit cards and electronic purses in Switzerland. An implementation of EP2 can be considered as a critical system as failure is unacceptable.

Examples of such unacceptable failures include transferring money to wrong bank accounts, transferring incorrect amounts, losing money during transfers, and so on. Each of these is an example of a failure which could have disastrous effects for customers, businesses and countries. EP2 is a running example that we use throughout this thesis and will be covered in more detail later in this introduction. Generally, the failure of a critical system may have serious consequences including substantial financial losses, environmental damage, injury or even loss of life. Critical systems can be classified into three main groups [Som07], namely:

**Business critical** where the failure of the system could jeopardise the stability of the business [Som07], for example, EP2 and control systems for the stock market.

**Mission critical** where the malfunction could lead to the failure of a mission (or goal orientated project) [Fow09] for example, the navigation system on a space probe.

**Safety critical** where the failure or malfunction could lead to the loss of life or large scale environmental damage [Sto96, Fow09], for example, flight control systems of an aircraft and medial devices such as automated infusion pumps.

Formal methods can be used during the design and development of such systems in order to reduce the risk of failure. In this thesis we aim to advance formal methods and broaden their application. We focus within the area of safety critical systems, although the work applies equally to all types of critical systems.

### 1.1.2 Formal Methods

Formal methods are the application of mathematical techniques to software engineering processes with the aim of improving the quality of produced software. There are estimated to be 3.3 software errors per thousand lines of code in large software systems [BS93]. Furthermore there are as many as $10^{20}$ unique end-to-end paths in a moderate-sized program [BS93]. Finding and correcting errors in such systems is a hard task. Systematic testing can help, but in order to be able to test a system we first need something to test against. That is, we need a specification of the system.

Creation of a specification is itself a challenging and often error prone task. Formal methods can help with this and can give us confidence that what we specify is of a certain quality (i.e., consistent). Formal methods allow us to write specifications and model systems using notions with a firm and well understood mathematical basis. Once a system has been formalised in a formal specification language we can try to verify its correctness and prove that it exhibits desirable properties. An example of a desirable property is deadlock freedom: a reactive system should not stop working, which would usually be seen as a failure. Thus, formal methods can help us create specifications of systems, which we can be certain are of a good quality. This in turn helps to increase our confidence that what we have specified is exactly what we intended to specify.

One downside of formal methods is that they are expensive. Verification of any system is usually a lengthy and costly task. Abrial [ACM10] estimates, using the specification language Event-B, that for a generated program with 100,000 lines of code, it would take 3.12 man-months of work to prove its correctness assuming that 97.5% of the proof obligations are automatically discharged. The man-months rise to 12.5 when only 90% of proof obligations are discharged automatically. Typically between 90% and 97.5% of proof obligations will be discharged automatically on such a program. These statistics clearly show that formal methods require tool support if they are to be practically useful and used in real world applications. Unfortunately, the costs are too high for most systems and as a result most systems are not verified. Critical systems however, due to their nature, require such verification. In certain industries verification is a legal requirement.

New developments and scientific results are enhancing verification techniques, as well as producing new ones. This is slowly reducing the burden of formal verification often making it easier, cheaper and more accessible to industry. In this thesis we develop such techniques further.

## 1.2 Development of Critical Systems

The development of critical systems usually follows an augmented version of the classical life cycle models, for example, the waterfall or spiral models [Boe88]. The first step is usually the specification of what the system should do. Such a specification itself needs to be developed. There are three complementary themes in such a development, namely: structure, vertical development and horizontal development. A system is an entity which is composed of other entities and thus has a structure. The first aspect is the identification of this structure. This is modelled and adapted throughout the development process. The second is vertical development. We usually think and develop systems in stages, starting from high levels of abstraction and then filling in details as we produce more concrete designs. This is vertical development. The third aspect is horizontal development, also known as *enhancement*, where instead of changing the abstraction level, we add, or more unusually remove, features of a system. For instance, Kahsai et al. [KRS08] describes enhancement for CSP-CASL [Rog06] with an example of extending a remote control unit by adding new buttons.

These three aspects overlap in parts, with horizontal development being captured partially by structuring and partially by vertical development (where we just choose not to change the abstraction level). Throughout this thesis we refer to vertical development as its other common name, namely *refinement*. We explore both of the concepts of structuring and refinement in the next two sub-sections. Following this, we look at the current verification techniques with CSP-CASL for reactive systems.

### 1.2.1 Systems as a Composition

A system is an entity which is composed of other entities. Understanding the structure of a system (or real world problem) and being able to break it down and identify its component parts is the first step in developing a well structured computer system. This structure is somehow natural for the problem being solved and is usually explorable during development. Systems are

not developed as one large monolithic entity, but are instead developed as smaller components which are combined to form the overall system.

As an example consider EP2. The system specified by the EP2 standard consists of eight components: a terminal, a cardholder, a point of service (POS), an attendant, a POS management system, an acquirer, a service center and a card. All these components work in parallel to form the overall system, with the terminal having a slightly special focus as the central component. Each component is described in isolation (as much as is possible) and developed individually (we discuss development within specifications in Section 1.2.2). This separation helps focus the specifier and the reader of the documents and aids understanding. The system is also naturally split into such components. It is natural for a retailer to have a Terminal and Chip and Pin machines, whilst the banks have back-end systems.

However, there is a significant downside to describing all components in isolation. That is, data needs to be duplicated. For instance, the interface between components needs to be described once for each component in each component's specification. Data types shared between components will also need to be described multiple times. Without doing this duplication each component can not be specified in isolation. This causes several specification issues. The most prevalent here is that data may be specified differently and inconsistently in each place it is described. Another problem is that the interfaces may not be compatible with each other. Informal specifications usually make a compromise between specifying components in isolation and data duplication. This is far from ideal, but probably the best that can be achieved with informal specifications.

Formal specifications, and thus formal methods, can hope to achieve both of these simultaneously. Specifications can be imported and reused. Thus, data types and interfaces can be specified in isolation and imported into the components that utilise them. This eliminates the risk that data types will be specified inconsistently in multiple places as there is no duplication. Rigorous mathematics can also be used to prove that the interfaces are compatible. Overall such use of formal specifications can increase our confidence that what we build is of a high quality.

Each component usually has both a reactive and a computational aspect. This is true for all the components in EP2. The reactive part captures how the component behaves and interacts with the world and other components around it. The computational part is important for making choices and transforming data, possibly obtained through prior interactions. Such decisions and computations can affect the behaviour of a component. For example, consider the following CSP [Hoa78, Hoa85, Ros98, Sch99, AJS05, Ros05, Hoa06] process (CSP will be introduced in Chapter 2)

$$\Box x :: Int \rightarrow \text{if } x \geq 0 \text{ then } P \text{ else } Q \ .$$

This is a CSP process which first receives a value out of the set $Int$ (representing the set of integers) and binds it to the variable $x$. Following this, the behaviour of the process depends on whether or not $x$ is at least zero. If this condition is true then the process behaves like the process $P$ otherwise it behaves like the process $Q$. This illustrates that data computations can affect the reactive behaviour of systems and that data properties can affect process properties. By splitting a system into its components we avoid trying to describe and understand, in one huge effort, the entire system and all its reactive and computational aspects.

Formal methods should be able mirror this natural composition of systems and modular development within formal specifications. The advantages of this are three-fold. Firstly, for methodological reasons, it is far easier to concentrate on and understand smaller components than a large monolithic system. Once all components are understood we are then able to understand the entire system and how the components operate together. Secondly, smaller components can be reused more easily and more often than larger ones and thus, structured specifications lend themselves more easily to code reuse. Finally, the structure of specifications can be used in the verification process to actually help establish the correctness of such specifications.

Formal specification languages which support structuring usually have a kernel structuring language [Mos02] which supports importation and union of specifications, renaming and hiding of symbols, and other more exotic features. These can be used as building blocks in formal specifications languages to create well structured system specifications with the advantages discussed above. The choice of structuring operations by Mossakowski is based upon work by Sannella and Tarlecki [ST88] who study structuring mechanisms for specification languages, and work by Bergstra et al. [BHK90] who study modularisation in the context of programming languages.

### 1.2.2 Refinement Based Development of Processes and Data

The first step in developing any system is to have an idea, usually at a high level of abstraction, of what properties the system should have. We then add details and develop the idea further until we have a fuller view of the system.

Figure 1.1 shows this process for EP2 and how this can be mirrored using formal methods. The left hand vertical chain shows the informal development process. Usually, we start the whole development process with an informal specification at a high level of abstraction, where the intricate details of the system are not considered. Instead the specifier focuses on the system architecture and large scale design, for EP2 we call this the architectural level. These specifications (or requirements) can then be enhanced and developed to include further detail, represented as the vertical chain. This process can be repeated until all details have been specified in full and the description of the final system has been reached, that is, the lowest abstraction level, which we call the concrete component level in Figure 1.1.

EP2 is specified informally in several documents which contain images, tables, text and UML-like diagrams. It was developed from high level descriptions down to very precise levels of detail. The development of EP2 includes the development of both the underlying data and the dynamics of the system. For example, on the data side, abstract communication messages are developed down to XML messages that contain prescribed data, for instance, payment details. On the process side, the behaviour of sub-systems and protocols are refined to more concrete versions. The specifications at all abstraction levels should split the system into its components as discussed in Section 1.2.1. Each component can be developed individually and their interactions adapted accordingly at each abstraction level. This is done largely in EP2 by focusing each document on one component.

Such development comes in two variations, which cannot be considered in isolation. The first is *data* development. At high levels of abstraction data is only vaguely described (i.e., the

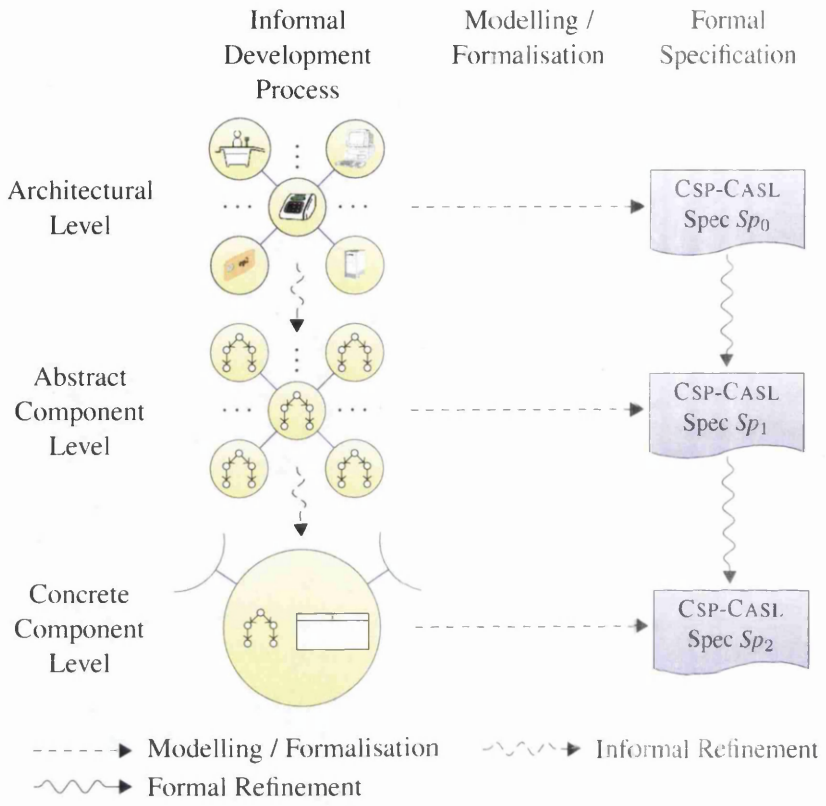



Figure 1.1: Illustration of the informal and formal development processes for EP2.

data is loose). This means that there can be many interpretations that fit with what has been specified. EP2 uses loose data in its high levels of abstraction in this way to specify data types of communication messages that pass between components. At the highest level of abstraction, EP2 only specifies that such messages exist. As systems are developed, the level of abstraction becomes lower (i.e., more concrete), with the lowest levels often explicitly stating the exact representation of the data. The second development variation is that of *process* (or behaviour) development. At high levels of abstraction the behaviour of systems usually involve areas which are vaguely described, they are black boxes, where the exact details of how they function is of no interest at such a high level. As the system and its components are developed down to lower levels, the design of such black boxes is filled in and made more precise. As we have noted in Section 1.2.1, data properties can influence process properties, thus it is impossible to develop solely the data of a specification and only then consider the process development (and also vice versa). One must do both intermixed and constantly consider how development of data influences the behaviour of components. Such development, in both data and processes, is a natural process: we start with high level ideas of what a system should achieve and slowly develop these ideas by filling in the details until we have a fully working system in mind.

The separation of such abstraction levels can have benefits not only for methodological

reasons, but also for formal verification. For instance, EP2 specifies a sub-sort structure of the communication messages at a fairly high level of abstraction, nothing else about the messages is specified. This has allowed us to prove, at a high level of abstraction, that the protocols that EP2 uses are actually deadlock free [GKOR09]. This can also be proven at lower levels of abstraction, but requires a greater effort as there are many more details to consider. Generally it is easier to establish properties at the higher levels of abstraction, where there are less specified details.

Between each level in the informal development chain is an informal refinement relation. This means that in an intuitive sense, the development step from the previous to the next level is a reasonable one (e.g,, that the development does not introduce inconsistencies).

Formal methods should be able to mirror this informal development process. This entails being able to model systems, such as EP2, at a range of abstraction levels from very high levels to the most concrete levels. The right hand chain in Figure 1.1 captures this formal development. We mirror each informal abstraction level with formal specifications of the system and its components.

At high levels of abstraction one should be able to formally specify both data and behaviour in a loose way. That is, many types of data and behaviour may fit the specification and there may be multiple ways to fulfil the requirements. This is exactly what is done informally at such abstraction levels where the intricate details are of no interest. For instance, EP2 specifies an architectural level where the architecture of the system is outlined and what types of data are used. At no point in this level is the data further detailed. Formal specification languages should be capable of capturing such abstract notions.

Finally, as we have formal specifications, we can establish formal refinements between each abstraction level. Whilst this has the same nature as the informal refinements, these formal refinements are part of formal methods and are founded in mathematics. We can prove they exist and that each development step is a "correct" and reasonable one within the particular formal framework in which we are working. Establishing the existence of such formal refinements is a form of verification which we cover next.

### 1.2.3 Verification

Once we have various abstraction levels of a system specified formally, we are able to perform verification. Verification comes in two forms. The first is checking that the development from one level to the next is a reasonable one. The designers of the system obviously intend for these to be reasonable development steps, but mistakes can be made. For instance, it is possible to describe either data or processes in a way which contradicts the same aspects described at a higher level of abstraction. With informal specifications the detection of such mistakes can only be done manually by humans, as informal specifications are not based in mathematics and thus may contain ambiguities. Formally checking development steps (i.e., establishing formal refinement relations) can bring to light such modelling errors.

The second form of verification is that properties can be established on formal specifications. For instance, it is possible to prove that such a specification is deadlock free or functionally correct. The benefits of this are two-fold. Firstly, it increases our confidence that the specifications are "correct", that is, that we are specifying the system that we intend to specify.

If it is not possible to prove that the specified system has some property that we expect it to have, then this may indicate that we have made a modelling error. Secondly, any "real" implementation of such a specification will then also be guaranteed to have such properties. Such verification is not possible with informal specifications.

Both these forms of verification can be achieved by establishing certain refinement relations. For instance, in CSP one can prove deadlock freedom of a process $P$ by showing that the most abstract deadlock free process refines to the process $P$ in the Stable-Failures semantics of CSP [Ros05]. As a data example, consider the specification of generic specifications and instantiation in CASL [Mos04, BM04]. In order to specify a generic specification (i.e., a parametrised specification), one specifies a formal parameter and uses this in the specification of the body of the generic specification. One can then specify an actual parameter and use this in the instantiation of the generic specification. However, the instantiation is only well formed if the formal parameter refines to the actual parameter. This is a healthiness check that the actual parameter's implementations are already contained within the formal parameter's implementations. If this is not the case then the construction does not make sense and indicates that the specifier has made an error. Hence, refinement is a central notion that tools must support in order for formal methods to be applicable and useful to real world problems.

Thus, ideally, formal development should allow for multiple levels of abstraction, from abstract levels to concrete ones, whilst also allowing for component based design. Such development steps should be able to develop both the data and the behaviour of components of the system. Loose data and processes should be able to be specified as well as concrete instances. Finally, we should be able to establish properties on such specifications, ideally at high abstraction levels and be guaranteed that such properties are inherited by the more developed lower levels.

### 1.2.4 CSP-CASL

Current formal specification languages, which integrate the specification of both processes and data, achieve the above (structuring, refinement and verification) to a certain degree, but not fully. CASL [Mos04, BM04], for instance, is tailored to the specification and development of data at various abstraction levels. Whilst it does lend itself to the development of components, it is not able to describe complex dynamic behaviour in an elegant and concise way. In contrast, CSP [Hoa78, Hoa85, Ros98, Sch99, AJS05, Ros05, Hoa06] is a formalism for the description of reactive systems. Whilst CSP can describe such dynamics of systems, also at various levels of abstraction, it neglects the development of data. CSP assumes that data is fixed the moment the system is first described. Therefore, data cannot be developed from abstract levels to concrete levels using traditional CSP.

These two languages were coupled with the creation of CSP-CASL [Rog06] which allows one to specify systems via a data part and a process part. In the data part one specifies using CASL all the data that is used in the system, whilst in the process part one uses this data and CSP to specify the behaviour of the system.

While development of both data and processes is possible in CSP-CASL, it only allows for limited compositional specifications as structuring is not available in the process part of a specification. Consider briefly Figure 1.2 which shows part of a CSP-CASL specification of

```
spec ARCH_INIT =
     data ...
     channel ...
     process let Acquirer = ...
                 Terminal = ...
             in Terminal [| ... |] Acquirer
end
```

Figure 1.2: Excerpt of the CSP-CASL specification of EP2 [GKOR09].

EP2 [GKOR09]. This CSP-CASL specification describes two components of EP2, namely the acquirer and the terminal. These are two components (out of eight) that form the EP2 system and run in parallel with each other to form part of the overall system. A CSP-CASL specification represents one system, whose behaviour is given by one nameless process, possibly using a *let* construction to allow for sub-processes. For example, the *Acquirer* and *Terminal* are two named sub-processes that run in parallel to form some overall unnamed system (i.e., the process *Terminal* [| ... |] *Acquirer*). Thus, in CSP-CASL, it is impossible to specify these two components (and their data) separately and later refer to them when constructing the overall system (i.e., the parallel composition). It is possible to do this with data in CSP-CASL, but currently not with processes. The behaviour of all components must be specified in a monolithic CSP-CASL specification. One approach to dealing with this is to specify components separately and to then "cut and paste" the specifications together. This is cumbersome and causes the loss of structure. As there is no structure in the resulting specification, it cannot be exploited in the verification effort. As CSP-CASL is limited in its compositional abilities, compositional reasoning is also limited and in most cases is impossible.

A second weakness of the original CSP-CASL is that one is forced to write a process equation for the unnamed process and all sub-processes. Thus, it is not possible to leave the interpretation loose. This means that you cannot specify your system as a black box where you do not restrict its behaviour. This ability is particular useful and in some cases required to capture systems at their very highest levels of abstraction. This is a limitation of the semantic construction: in CSP-CASL each data model gives rise to exactly one process denotation. While loose process semantics is nice for methodological reasons, is is a necessity for generic (or parametrised) specifications involving processes and data. Thus CSP-CASL, even if it had more comprehensive structuring mechanisms, would not be able to support generic specifications.

CSP-CASL, despite its limitations, does have tool support for dealing with specifications that capture both data and processes. The tool HETS [MML07] allows for parsing, static analysis and proof support by utilising CSP-CASL-Prover [O'R08]. These tools have been used extensively in the formal modelling of EP2 [Kah10]. Therefore it is possible to use CSP-CASL in its current form for industrial projects. However, such efforts have to work around the limitations discussed above.

We wish to address the limitations of CSP-CASL within this thesis and bring compositional

reasoning to CSP-CASL specifications involving processes and data.

## 1.3 Project Aims

We aim to improve and extend the specification language CSP-CASL and address its weaknesses. Specifically we aim to:

- Adapt the semantics of CSP-CASL to support loose process semantics. This will make it possible for CSP-CASL to support generic (or parametrised) specifications and will also allow CSP-CASL to capture the very highest levels of abstraction where processes are only declared to exist and are treated as black boxes.
- Extend CSP-CASL to support full structuring in the process part. Full structuring is currently only available in the data part. This enhancement will allow compositional descriptions of reactive systems at various levels of abstraction.
- Develop a new refinement notion which deals with the new loose process semantics. This will allow for full rigorous system development with refinement chains that can be proven to hold.
- Develop proof calculi which allow CSP-CASL refinements to be proven in a way which can utilise the structure of the specifications involved. We believe that by exploiting the structure of the specifications we can ease the proof burden of establishing refinement relations.
- Implement the enhancement of CSP-CASL, the new refinement notion and the proof calculi in a prototypical way.

Achieving the above aims will also demonstrate that it is possible to combine data and process specification in a common formalism and utilise the structure of a specification in the verification effort at all levels of abstraction.

## 1.4 Publications

Material from the following published papers contribute to this thesis:

**Compositional modelling and reasoning in an institution for processes and data (with Mossakowski and Roggenbach), WADT 2010 [OMR12]** outlines the details of loose process semantics and the new institution for CSP-CASL. It also includes the new refinement notion and proof calculi dedicated to deadlock analysis (which have been refined and developed in this thesis). I contributed to the development of the institution and refinement notion and was responsible for the deadlock and refinement calculi.

**On the whereabouts of CSP-CASL– A Survey (with Gimblett, Kahsai and Roggenbach), BKB Festschrift 2011 [GKOR09]** provides a comprehensive but concise overview of CSP-CASL, from its original design and refinement notion, though modelling

EP2 and the design and implementation of tool support, to specification based testing. My contributions to this paper were the construction of CSP-CASL-Prover and work on the verification of EP2. The work I contributed to this paper was based upon my M.Phil [O'R08].

**Compositional reasoning for processes and data (with Mossakowski and Roggenbach), ARW 2011 [MOR11]** was a workshop where a brief overview of this work, including the addition of loose process to CSP-CASL was presented. The work presented focused on structured specifications, in particular parametrised specifications. I was the main contributor to this work.

## 1.5 Thesis Overview

We begin this thesis by introducing the languages involved in the specification language CSP-CASL. Chapter 2 introduces the process algebra CSP and presents its syntax, various semantics, and refinement notions which allow for development of processes. Tool support is also briefly discussed.

Chapter 3 introduces CASL via an example of specifying a track plan in the railway domain. The syntax and semantics of CASL is presented in an informal way which covers both sub-sorting and partiality. The example makes extended use of structuring, generic specifications and instantiations, and also demonstrates formal verification. Tool support is also briefly discussed.

Chapter 4 introduces the formal framework of institutions along with various examples. The institutions $PCFOL^{=}$ (partial first order logic with sort generation constraints and equality) and $SubPCFOL^{=}$ (sub-sorted partial first order logic with sort generation constraints and equality) are presented, the latter being the underlying institution of CASL. The data-logic for CSP-CASL is also presented as an institution, along with some institutions capturing CSP. Finally, institution independent structuring mechanisms are described.

Chapter 5 introduces the original CSP-CASL, where we sketch the semantics and its refinement notion. We present an example of using CSP-CASL to specify and partially verify EP2. We also discuss the limitations of the original CSP-CASL and how we intend to overcome them.

Chapter 6 discusses related work. Various different approaches of integrating the specification of data and processes are presented, along with discussions of the degree of structuring available with each approach.

Chapter 7 presents various constructions required to construct the CSP-CASL institutions. We first present how to lift alphabet translations to covariant CSP domain translations and, in the case of injective alphabet translations, to contravariant CSP domain translations. Following this we describe how to flatten CASL models to alphabets and lift model morphisms and reducts to alphabet translations. We prove various proprieties of these translations which will be required for the construction of the CSP-CASL institutions.

Chapter 8 presents the construction of the three CSP-CASL institutions, one for each of the CSP semantics. Each institution shares the same constructions for signatures and sentences,

but differs in their models and satisfaction relations. The models and satisfaction relations, however, follow a common construction scheme.

In Chapter 9 we develop a new refinement notion for CSP-CASL that supports named processes and loose process semantics. We provide several proof calculi supporting compositional refinement and deadlock analysis over Structured CSP-CASL. Following this, we provide a method for deadlock analysis of networks [RSR04]. Finally, we show that a complete refinement calculus for Structured CSP-CASL is possible, provided structured specifications are restricted to certain forms.

Chapter 10 presents an example showing the use of the compositional proof calculi. We model an online shopping system with four components, namely, a customer, coordinator, payment system and a warehouse. These four components are specified separately and are combined via structuring to form the final specification. We specify the system and components on various levels of abstraction. Furthermore, we prove deadlock freedom of the system.

Chapter 11 discusses the implementation of the CSP-CASL institutions. We extend the tool HETS with the new notions for Structured CSP-CASL. This gives us the ability to parse, statically analyse and pretty print Structured CSP-CASL specifications. The static analysis checks various conditions are meet, such as the conditions on CSP-CASL signature morphisms.

Chapter 12 presents a summary, while Chapter 13 discusses various future works. Appendix A presents various proofs which are deferred in the main text. Appendix B presents the CSP semantical clauses deferred from Chapter 2, while Appendix C presents specifications of an example ATM system used in Chapter 8. Finally, Appendix D presents full specifications for the example presented in Chapter 10.

# Part I

# Background

*Chapter 2*

# CSP

**Contents**



CSP (Communicating Sequential Processes) [Hoa78, Hoa85, Ros98, Sch99, AJS05, Ros05, Hoa06] is a process algebra, and is one of many formalisms developed for the description of reactive systems. It was first developed by Hoare in 1978 [Hoa78] and allows for the description of processes by specifying their behaviour via their communications.

A process in CSP can be seen as a black box which can communicate with its environment (or other processes) using a set of fixed communication events, usually called an *alphabet*. The idea of CSP is to observe what a process does; in this way an observer can look at a process and simply record the communications of the black box by writing them down in a list.

A process in CSP is built using a number of building blocks which will be introduced in Section 2.1 and Section 2.2. CSP has a variety of semantics, described in Section 2.4, which give meaning to processes. Each of the semantics is tailored to capture different aspects of the behaviour of processes and comes equipped with a refinement notion which relates processes with each other and allows for process development. The refinement notions perverse selected properties, usually the properties that the semantics focus on. This allows for development which can preserve such properties.

Today, CSP has many flavours where most of the differences are in the available syntax. In this chapter we present CSP's syntax, semantics and refinement notions closely following Roscoe [Ros05], which we will use later on when defining the CSP-CASL institutions in Chapter 8.

$$
\begin{array}{rcll}
\mathit{Proc} & ::= & \mathrm{SKIP} & \%\% \text{ terminating process} \\
& | & \mathrm{STOP} & \%\% \text{ deadlock process} \\
& | & \mathrm{DIV} & \%\% \text{ diverging process} \\
& | & a \rightarrow \mathit{Proc} & \%\% \text{ action prefix process} \\
& | & \sqcap x :: X \rightarrow \mathit{Proc} & \%\% \text{ internal prefix choice} \\
& | & \Box x :: X \rightarrow \mathit{Proc} & \%\% \text{ external prefix choice} \\
& | & \mathit{Proc} \fatsemi \mathit{Proc} & \%\% \text{ sequential composition} \\
& | & \mathit{Proc} \sqcap \mathit{Proc} & \%\% \text{ internal choice} \\
& | & \mathit{Proc} \Box \mathit{Proc} & \%\% \text{ external choice} \\
& | & \mathit{Proc} \,||\, \mathit{Proc} & \%\% \text{ synchronous parallel} \\
& | & \mathit{Proc} \,|||\, \mathit{Proc} & \%\% \text{ interleaving} \\
& | & \mathit{Proc} \,[\![ X ]\!]\, \mathit{Proc} & \%\% \text{ generalised parallel} \\
& | & \mathit{Proc} \,[\![ X \mid Y ]\!]\, \mathit{Proc} & \%\% \text{ alphabetised parallel} \\
& | & \mathit{Proc} \setminus X & \%\% \text{ hiding} \\
& | & \mathit{Proc}[R] & \%\% \text{ relational renaming} \\
& | & \text{if } \varphi \text{ then } \mathit{Proc} \text{ else } \mathit{Proc} & \%\% \text{ conditional}
\end{array}
$$

where $a$ is an action in the Alphabet $A$, the sets $X, Y \subseteq A$ are synchronisation sets, $x$ is a variable, $\varphi$ is a formula, and $R \subseteq A \times A$ is a renaming relation.

Figure 2.1: The basic syntax of CSP processes [Rog06].

## 2.1 The Syntax of CSP

Here, we describe CSP's syntax and explain its intuitive meaning using examples. CSP processes are built over a fixed alphabet (i.e., a set) of actions or events. Figure 2.1 shows the basic syntax of CSP processes over such a fixed alphabet $A$. Processes are constructed from basic processes (SKIP, STOP, and DIV), various prefix operators, sequential composition, various choice operators, parallel composition, hiding, renaming and a conditional construct.

We will now describe the idea behind each of the operators with examples. For these examples we will think of the semantics (for now) as the Trace semantics, which will be made precise in Section 2.4. The traces of a process are simply sequences of actions which the processes can engage in.

There are three basic processes: STOP, SKIP and DIV. STOP is the process which does nothing. It does not communicate and refuses to do so. It represents deadlock. SKIP on the other hand represents successful termination (by communicating the special action $\checkmark$ and then deadlocking, see Section 2.4). DIV is the process which continually engages in internal (also called hidden) actions.

The action prefix operator is the heart of CSP: $a \rightarrow P$ is the process that first communicates the action $a$ and then behaves like the process $P$. With this operator it is possible to build up

chains of communications. For instance, consider a light switch modelled as the process:[1]

$$Light\_Switch = on \rightarrow off \rightarrow Light\_Switch \quad .$$

This light switch can first communicate an *on* action (representing the unit being turned on) followed by an *off* action (which represents the unit being turned off). CSP does not state that the light switch turns itself on and off, but simply that it can be turned on and off. After having been turned on and then off the process simply repeats and acts once again as a light switch. Hence, we have modelled a light switch which has the ability to be toggled on and off forever. This is achieved using recursion, which is the means by which CSP describes infinite behaviour.[2]

The sequential composition operator allows two processes to be composed together in series. The second process takes over only when and if the first process terminates successfully. For instance, the process

$$Seq_1 \equiv a \rightarrow \text{SKIP} \fatsemi b \rightarrow \text{STOP}$$

is the process which communicates an $a$ action followed by a $b$ action and then deadlocks, while the process

$$Seq_2 \equiv a \rightarrow \text{STOP} \fatsemi b \rightarrow \text{STOP}$$

communicates an $a$ action and then deadlocks. The process $Seq_2$ only communicates an $a$ action as the first sub-process (i.e., $a \rightarrow$ STOP) never terminates successfully and thus the second sub-process ($b \rightarrow$ STOP) never takes over.

CSP has two binary choice operators, namely, internal choice (i.e., $Proc \sqcap Proc$) and external choice (i.e., $Proc \Box Proc$), which allow us to make a choice between the behaviour of two processes. For instance, consider a *mover* person who wishes to move *pianos* and *tables* (based on examples taken from Schneider [Sch99]):

$$Mover = move\_piano \rightarrow Mover \Box move\_table \rightarrow Mover \quad .$$

Here, the action prefix operator binds tighter than the both the internal and external choice operators. *Mover* is the process which can first perform one of two actions namely *move_piano* or *move_table*, where the choice is given to the environment as we have used the external choice operator. This offer of choice to the environment allows the environment to influence the behaviour of the process. Once a choice has been made the process continues down that branch. We can also consider a *stubborn mover* person who will not be influenced in such a decision and instead chooses non-deterministically:

$$Stubborn\_Mover = move\_piano \rightarrow Stubborn\_Mover \sqcap move\_table \rightarrow Stubborn\_Mover \quad .$$

Here, the process chooses non-deterministically which branch to use and offers only one action: either a *move_piano* action or a *move_table* action, but not both. This offer of choice (to the

[1] We write $P \equiv Q$ for syntactic equality of $P$ and $Q$, and $P =_{\mathcal{D}} Q$ for semantical equality in the CSP semantics $\mathcal{D}$. We drop the subscript $\mathcal{D}$ if the equality holds in the three main semantics, that is, $\mathcal{T}, \mathcal{N}$ and $\mathcal{F}$. However, we use $=$ when defining recursive processes.

[2] In the standard denotational semantics of CSP, fixed point theorems are used for recursion. CSP has two denotational approaches to recursion namely: Banach's fixed point theorem with complete metric spaces and Tarski's fixed point theorem with complete partial orders.

environment) is the difference between the internal and external choice operators in CSP (other variations will be discussed shortly). The external choice operators allow the environment to make the choice while the internal choice operators do not. The offer of choice becomes particularly important when combining processes using the parallel operators (which again will be discussed shortly).

CSP has two types of prefix choice operators, namely, internal prefix choice (i.e., $\sqcap x :: X \rightarrow Proc$) and external prefix choice (i.e., $\Box x :: X \rightarrow Proc$). These allow the choosing of a communication from a set of values. For instance, the process

$$\sqcap x :: \{a, b, c\} \rightarrow x \rightarrow \text{STOP}$$

is the process which first communicates one of the actions $a$, $b$ or $c$, then re-communicates this action, and finally deadlocks. The action is bound to the variable $x$ for use later on as a valid action. The choice of the action in this instance is non-deterministic as this is the internal choice operator. In contrast, the process

$$\Box x :: \{a, b, c\} \rightarrow x \rightarrow \text{STOP}$$

is willing to engage in any of the actions $a$, $b$ or $c$ and allows the environment to choose.

Now we approach the more complex operators, namely the parallel composition operators. These come in four variations, the two simplest are the interleaving operator and the synchronous parallel operator.

We can use the synchronous parallel operator to force two processes to work together and synchronise on all events. As pianos and tables are usually heavy, they normally require two people (in this example) to move them. The synchronous parallel operator allows us to model this very easily. Consider the process

$$Movers \equiv Mover \parallel Mover$$

where we have put two mover people together and require they work together. As they both allow the environment (in this case the other person) to make the choice between moving pianos and tables, they end up agreeing and actually moving furniture around. When they both synchronise on either a *move_piano* or a *move_table* action, only one action is communicated to the outside world (as in real life only one table would be moved by two people).

If instead we used two stubborn movers, as in

$$Stubborn\_Movers \equiv Stubborn\_Mover \parallel Stubborn\_Mover$$

then it is possible that no furniture will ever get moved, as one of the stubborn movers may choose to move a piano and the other to move a table. They each choose independently and non-deterministically as they have both been defined using an internal choice operator. As they must agree on all actions (dictated by the use of the synchronous parallel operator) and assuming pianos are not tables then they may not agree and will become deadlocked. This shows how the different choice operators can be used to achieve quite subtly different behaviours.

The other parallel operators work in similar ways but with different notions of cooperation. The interleaving operator allows two processes to proceed independently of each other: they

do not synchronise on any events, while the generalised parallel operator specifies a set upon which the processes must synchronise and outside of this set they can proceed independently. Finally, the alphabetised parallel operator specifies two sets for synchronisation where only actions in the intersection need to be synchronised. For example, the process

$$P \,|[\, X \mid Y \,]|\, Q$$

is the process which combines the process $P$ and $Q$ where $P$ synchronises over $X$ and $Q$ synchronises over $Y$. Thus, overall each can proceed independently with actions outside of $X \cap Y$ but any actions in the intersection must be agreed upon and synchronised. The alphabetised parallel operator becomes important when creating networks, discussed in Section 2.2.

The hiding operator hides a set of events in a process. When combining processes in a system, say with the synchronous parallel operator, some events may be intended to be internal communications between the sub-systems involved. However, these events are still visible after the parallel compositions. This is intended by the design of CSP and is the mechanism that allows multi-way synchronisation to work [Sch99]. The hiding operator allows us to encapsulate such communications within the process. The process

$$P \setminus X$$

is a CSP process where none of the events in $X$ can be observed. The process $P$ however, can use these events as normal.

As an example consider a stop-and-wait protocol implementing a one-place buffer (as presented in [Sch99]). This example uses channels which will be introduced in the next section, however channels are very intuitive and allow the sending of values though "pipes". The idea here is to build two sub-systems (*Receiver* and *Sender*) which when combined form a one-place buffer. The one sub-system (*Receiver*) receives input and places the input in the buffer. The other sub-system (*Sender*) reads from the buffer and outputs the information. The two systems are specified as:

$$\begin{aligned} \mathit{Receiver} &= \mathit{in}\,?x :: T \rightarrow \mathit{mid}\,!x \rightarrow \mathit{ack} \rightarrow \mathit{Receiver} \\ \mathit{Sender} &= \mathit{mid}\,?y :: T \rightarrow \mathit{out}\,!y \rightarrow \mathit{ack} \rightarrow \mathit{Sender} \end{aligned}$$

The *Receiver* process reads a value of type $T$ (i.e., out of the set $T$), over the channel *in* and binds it to the variable $x$. It then sends this value on the channel *mid*. After this it communicates an acknowledgement and repeats. The sender reads a value from the channel *mid* and outputs it on the channel *out*. After this it communicates an acknowledgement and repeats. In this way the channel *mid* can hold a single value which will be communicated between the two sub-processes. These two sub-processes are to synchronise using the internal message *ack* in order to know when the buffer is free again. Thus, the system is:

$$\mathit{Sys} \equiv (\mathit{Receiver} \parallel \mathit{Sender}) \setminus (\mathit{mid}.T \cup \{\mathit{ack}\})$$

Here, we put the *Receiver* and *Sender* in parallel and hide all communications over the channel *mid* and the action *ack*. Overall the two-sub processes communicate values over the channel *mid* and synchronise on the communication *ack*. However, none of these communications can

$$
\begin{array}{lcll}
Run(X) & \widehat{=} & Run_X = \Box x :: X \rightarrow Run_X & \% \text{ run operator} \\
Chaos(X) & \widehat{=} & Chaos_X = \text{STOP} \sqcap (\Box x :: X \rightarrow Chaos_X) & \% \text{ chaos operator} \\
P \rhd Q & \widehat{=} & (P \sqcap \text{STOP}) \,\Box\, Q & \% \text{ untimed timeout operator}
\end{array}
$$

Figure 2.2: Basic extensions for CSP processes.

be seen. To the outside world this is a black box which receives values on the channel *in* and repeats them on the channel *out*.

The hiding operator can be tricky to use correctly as its use can introduce non-determinism in an otherwise deterministic process, by hiding events which control the flow of a process. When the events offered by an external choice are hidden, the environment no longer has any control over how the choice is resolved: it is resolved internally [Sch99].

The renaming operator allows a process to be renamed. For instance, returning to our piano and table movers example, we can create a new mover person who moves tables and bookshelves by renaming the action *piano* to *bookshelf*, that is, the process

$$Mover[R]$$

where $R = \{(piano, bookshelf)\} \subseteq A \times A$. This allows us to reuse processes in the creation of new ones just by renaming the communications.

The final operator is the conditional operator that allows a choice based on some formula over a suitable logic that deals with variables over the alphabet.

This concludes the syntax for basic process in CSP. In the next section we discuss some of the many extensions of this basic syntax.

## 2.2 Syntax Extensions

We now introduce extra convenient syntax which is shorthand for expressions that can already be expressed in the original syntax of CSP in Figure 2.1. These extensions include basic extensions, networks and channels.

### 2.2.1 Basic Extensions

Figure 2.2 shows some basic syntactic sugar for CSP. $Run(X)$ is the process which continuously offers all actions from $X$. As it offers an external choice it allows the environment to choose the particular action out of $X$. $Chaos(X)$, on the other hand, can always choose to communicate or reject any member of $X$. Its behaviour cannot be predicted, hence its name.

The expansion for $P \rhd Q$ is the only non-intuitive expansion of the three. This operator models an untimed timeout. Hence, $P \rhd Q$ intuitively states $P$ will start within a given time frame, otherwise $Q$ will take over. As we do not have time in our vocabulary we model this timeout as a non-deterministic choice, that is, the timeout may occur at anytime. $P \rhd Q$ will behave like $P$ whenever $P$ starts, that is, a visible action from $P$ occurs. However, this may not happen in time and then a timeout occurs and it behaves like $Q$. This is a non-deterministic choice, but once the choice between behaving like $P$ or $Q$ is made, then it is stuck to. For further detail see [Ros05].

### 2.2.2 Networks

Another extension of CSP is that of *networks* which will be utilised in Chapter 9 and Chapter 10. Networks are built from the alphabetised parallel operator (i.e., $Proc \; [\![ X \mid Y ]\!] \; Proc$). For instance, consider four processes: a customer, a coordinator, a payment system and a warehouse, with individual alphabets $A_C$, $A_{CO}$, $A_{PS}$ and $A_W$ respectively. These processes work together to provide an *online shopping system*, that is, they form a network. The network of these processes would be the process:

$$\begin{array}{l}((\mathit{Customer} \; [\![ A_C \mid A_{CO} ]\!] \; \mathit{Coordinator}) \\ \quad [\![ A_C \cup A_{CO} \mid A_{PS} ]\!] \; \mathit{Payment\_System}) \\ \quad\quad [\![ A_C \cup A_{CO} \cup A_{PS} \mid A_W ]\!] \; \mathit{Warehouse}\end{array}$$

This is a common way to build networks [Ros05]. We place each process in parallel over its own alphabet. Thus, the components must only cooperate on the intersection of their alphabets. They are free to proceed independently outside of this set. When placing an alphabetised parallel component (say, $\mathit{Customer} \; [\![ A_C \mid A_{CO} ]\!] \; \mathit{Coordinator}$) in parallel with another component (say, *Payment_System*), the alphabet of the alphabetised parallel component is taken to be the union of its sub-components (i.e., $A_C \cup A_{CO}$). In this way we can build up any finite network from just using the alphabetised parallel operator and union on sets.

These network expressions quickly become large and tedious to write out as more parallel processes are added. Hence, it makes sense to introduce a concise notation for networks. The fundamental observation that makes this possible is that the alphabetised parallel operator is associative and symmetric, i.e., the following equations hold:

- $(P \; [\![ X \mid Y ]\!] \; Q) \; [\![ X \cup Y \mid Z ]\!] \; R = P \; [\![ X \mid Y \cup Z ]\!] \; (Q \; [\![ Y \mid Z ]\!] \; R)$.
- $P \; [\![ X \mid Y ]\!] \; Q = Q \; [\![ Y \mid X ]\!] \; P$.

The proofs of these equivalences are presented by Roscoe [Ros05]. This means that the order of processes combined with the alphabetised parallel operator does not matter.

With these properties in place we are able to define a network construction for a finite collection of processes. A network $N$ is a finite set of pairs $\{(P_i, A_i) \mid i \in I\}$, where $I$ is a non-empty, finite index set, $P_i$ is a CSP process, and $A_i \subseteq A$ is the set of communications which $P_i$ can engage in, for all $i \in I$. The process defined by such a network $N$ is

$$\mathit{Network}(N) \equiv \; \|_{i \in I} \; (P_i, A_i)$$

where $\|_{i \in I} \; (P_i, A_i)$ is the replicated alphabetised parallel operator which can be expanded in the finite case (which networks are by definition) to multiple applications of the alphabetised parallel operator [Ros05], that is,

$$\begin{array}{l}\|_{i \in \{1 \ldots n\}} \; (P_1, A_1), (P_2, A_2), (P_3, A_3), \ldots, (P_{n-1}, A_{n-1}), (P_n, A_n) \equiv \\ \quad ((P_1 \; [\![ A_1 \mid A_2 ]\!] \; P_2) \\ \quad\quad [\![ A_1 \cup A_2 \mid A_3 ]\!] \; P_3) \\ \quad\quad\quad \ldots \\ \quad\quad\quad\quad [\![ A_1 \cup A_2 \cup A_3 \cup \ldots \cup A_{n-1} \mid A_n ]\!] \; P_n \, .\end{array}$$

$$
\begin{array}{lcll}
c\,!\,v \rightarrow P & \widehat{=} & c.v \rightarrow P & \%\% \text{ channel sending} \\
c\,?\,x :: T \rightarrow P(x) & \widehat{=} & \Box y :: c.T \rightarrow P(x)[\mathit{strip}(y)/x] & \%\% \text{ channel receiving} \\
c\,!\,x :: T \rightarrow P(x) & \widehat{=} & \sqcap y :: c.T \rightarrow P(x)[\mathit{strip}(y)/x] & \%\% \text{ non-deterministic channel} \\
 & & & \%\% \text{ sending}
\end{array}
$$

where the strip operation removes the channel label from the variable, i.e., $\mathit{strip}(c.v) = v$.

Figure 2.3: Channel extensions for CSP processes.

### 2.2.3 Channels

When modelling systems in CSP one is often interested in sending and receiving values from certain sets but where synchronisation can be controlled. Channels [Ros05] are syntactic sugar and provide a convenient way to do this. Channels can (usually) be thought of as "pipes" which allow values to be communicated to other processes. In order to use channels you must allow your alphabet $A$ to have compound names composed of several parts, each separated by a dot. Let $c$ be the name of a channel and $T$ be the type of the channel (a set of values which can be used with the channel) then we have the set

$$c.T = \{c.x \mid x \in T\} \subseteq A$$

which consists of all values in $T$ with a prefix of $c$ added as a "label". This notion of "labelling" forms the basis for channels.

There are three main channel operations: sending values over channels, non-deterministic sending of values over channels and receiving on channels. Figure 2.3 shows the expansions for each of these operators. The first, $c\,!\,v \rightarrow P$, allows us to send a value $v \in T$ over the channel $c$ (which is typed as $T$). This is expanded to the communication of the value $v$ tagged with the channel $c$. The second, $c\,?\,x :: T \rightarrow P(x)$, allows us to receive any value in $T$ over the channel $c$ and bind the value to $x$. This is expanded to the external prefix choice operator. One complication that arises here is that a value from the set $c.T$ is tagged with $c$. In the channel version we want the variable $x$ to be bound to the untagged value. Thus, we use a *strip* operator to remove the tag when substituting the value for the variable $x$ in the process $P(x)$. Finally, $c\,!\,x :: T \rightarrow P(x)$, allows us to non-deterministically choose a value from the set $T$ and send it down channel $c$. This is similar to receiving, but we use the internal prefix choice operator instead.

This concludes our discussions about some of the various extensions to the basic syntax of CSP. We now look at some typical equivalences between processes.

## 2.3 Typical Laws

We now present a selection of *laws* which have been proven to hold for all the main CSP semantics (discussed in the next section) [Ros05, IR06].

Below are several representative example of laws from various 'categories':

**Unit law** $\text{STOP} \,\Box\, P = P$

**Commutativity law** $P \,\Box\, Q = Q \,\Box\, P$

**Associativity law** $(P \,\Box\, Q) \,\Box\, R = P \,\Box (Q \,\Box\, R)$

**Distributivity law** $(P \sqcap Q) \,\Box\, R = (P \,\Box\, R) \sqcap (Q \,\Box\, R)$

**Step law** $(\Box x :: X \to P) \,\Box\, (\Box y :: Y \to Q) =$
$\Box x :: X \cup Y \to (\text{if } x \in X \cap Y \text{ then } P \sqcap Q \text{ else if } x \in X \text{ then } P \text{ else } Q)$

These laws allow us to rewrite processes into equivalent ones, thus providing the foundation for reasoning about processes without delving into the denotational semantics. The axiomatic semantics use rules of a similar nature to these. The step laws provide the basis for the rewriting. Most other rules allow us to rewrite processes into the form needed to apply a step law. It is these step laws that actually make progress with the rewriting.

The above step law states that if we have an external choice of two sets $X$ and $Y$, then this is equivalent to externally choosing an action out of $X \cup Y$ then if the action comes from $X \cap Y$ we behave as an internal choice of $P$ and $Q$ else we behave like $P$ or $Q$ depending on whether the action is in the set $X$ or the set $Y$, respectively.

In a another sense, the fact that these equivalences hold provide confidence that the semantics make sense. We would expect that external choice is symmetric, if this was not the case in a particular semantics then it would indicate that there is something odd going one inside the semantics.

## 2.4 The Semantics of CSP

We now study the various semantics for CSP. The main purpose of a semantics $\mathcal{D}$ is to provide answers to the following questions:

- Are two processes $P$ and $Q$ are equal (with respect to semantics $\mathcal{D}$), written as $P =_{\mathcal{D}} Q$?
- Does process $P$ refine to process $Q$ (with respect to semantics $\mathcal{D}$), written as $P \sqsubseteq_{\mathcal{D}} Q$?

These two questions are in fact equivalent, in that each question can be formulated in terms of the other. Hence, if a semantics can answer one question then it can also answer the other.

We show this equivalence in the denotational setting of CSP which assigns mathematical objects called denotations to processes, these represent the meaning of a process. Denotational semantics will be discussed in detail shortly.

Let $A$ be some fixed alphabet. Any denotational semantics $\mathcal{D}$ of CSP defines a function $[\![\_]\!]_{\mathcal{D}} : \mathit{Proc} \to \mathcal{D}(A)$, which assigns a denotation in $\mathcal{D}(A)$ (i.e., a meaning over alphabet $A$) to each process. We write $P =_{\mathcal{D}} Q$ for $[\![P]\!]_{\mathcal{D}} = [\![Q]\!]_{\mathcal{D}}$.

**Theorem 2.1** Let $\mathcal{D}$ be some semantics for CSP with the following properties (which all the main denotational semantics exhibit):

- $P \sqsubseteq_{\mathcal{D}} Q$ iff $[\![Q]\!]_{\mathcal{D}} \subseteq [\![P]\!]_{\mathcal{D}}$.
- $[\![P \sqcap Q]\!]_{\mathcal{D}} = [\![P]\!]_{\mathcal{D}} \cup [\![Q]\!]_{\mathcal{D}}$.

then the following equations hold:

- $P =_{\mathcal{D}} Q$ iff $P \sqsubseteq_{\mathcal{D}} Q \wedge Q \sqsubseteq_{\mathcal{D}} P$.
- $P \sqsubseteq_{\mathcal{D}} Q$ iff $P \sqcap Q =_{\mathcal{D}} P$.

*Proof.* We first prove the former equivalence. By unfolding the definition of $=_{\mathcal{D}}$ in $P =_{\mathcal{D}} Q$ and taking equality to be subset inclusion in both directions, we get $[\![P]\!]_{\mathcal{D}} \subseteq [\![Q]\!]_{\mathcal{D}}$ and $[\![Q]\!]_{\mathcal{D}} \subseteq [\![P]\!]_{\mathcal{D}}$. Thus by definition of refinement we know $P \sqsubseteq_{\mathcal{D}} Q$ and $Q \sqsubseteq_{\mathcal{D}} P$. The other direction is analogous.

We now prove the latter equation.

| | | | |
|---|---|---|---|
| $P \sqsubseteq_{\mathcal{D}} Q$ | iff | $[\![Q]\!]_{\mathcal{D}} \subseteq [\![P]\!]_{\mathcal{D}}$ | by definition of $\sqsubseteq_{\mathcal{D}}$ |
| | iff | $[\![Q]\!]_{\mathcal{D}} \cup [\![P]\!]_{\mathcal{D}} = [\![P]\!]_{\mathcal{D}}$ | by the definition of subset inclusion |
| | iff | $[\![P \sqcap Q]\!]_{\mathcal{D}} = [\![P]\!]_{\mathcal{D}}$ | by given property of $\mathcal{D}$. |
| | iff | $P \sqcap Q =_{\mathcal{D}} P$ | by definition of $=_{\mathcal{D}}$. |

□

The first equivalence in Lemma 2.1 shows the relationship between semantical equality and refinement, while the second shows the relationship between refinement and the internal choice operator. The second equivalence is also used as a typical law similar to those discussed in Section 2.3.

Now that we know the purpose of semantics for CSP, we describe the typical semantics. The semantics for CSP come in three different forms, namely:

**Axiomatic semantics** which allow for derivation of facts from derivation rules. Syntactic processes are transformed via rules into other equivalent syntactic processes. Two processes are equivalent if they can be derived from each other via the rules. Such rules are similar to the ones presented in Section 2.3. The Stable-Failures semantics (discussed in the denotational setting in the following sections) has a complete axiomatic semantics [IR06], whilst it is unknown whether the traces semantics and the Failures/Divergences semantics also have a complete axiomatic semantics.

**Denotational semantics** where mathematical objects denote the meaning of processes. We will see denotation semantics later in this chapter.

**Operational semantics** where transition systems are created using structural operational semantics that represent the behaviour of processes. Equivalence notions on the underlying transition systems allow processes to be related to each other in various ways. The operational semantics of CSP are in fact equivalent to the denotational semantics, that is, they are congruent [Ros05].

We will only be concerned with denotational semantics throughout this thesis. CSP has three main denotational semantics namely:

**Traces semantics $\mathcal{T}$:** generally used for safety analysis,

**Failures/Divergences semantics $\mathcal{N}$:** generally used for live lock analysis, and

**Stable-Failures semantics $\mathcal{F}$:** generally used for deadlock analysis.

Each semantics comes equipped with a domain and a set of clauses that produce a mathematical denotation (in the domain) for a CSP process. In the following sections let $A$ be an alphabet and let an $\checkmark \notin A$ be an element denoting successful termination. Let $A^{\checkmark} := A \cup \{\checkmark\}$ and $A^{*\checkmark} := A^* \cup \{s^\frown\langle\checkmark\rangle \mid s \in A^*\}$, where $A^*$ is the set of all finite strings over $A$, $^\frown$ is the string concatenation operator and $\langle x\rangle$ is the singleton string containing $x$. Furthermore, let $\mathcal{P}(X)$ be the power set of $X$. We now present each of the domains as they are defined by Roscoe [Ros05].

### 2.4.1 The Traces Semantics $\mathcal{T}$

The Traces semantics (as presented by Roscoe [Ros05]) captures the notion of observing what a process can do, that is, its *traces*. We simply observe and record all the actions that a process may perform. For instance

$$\langle move\_piano, move\_table\rangle$$

is a single trace where we first observe the moving of a piano followed by the moving of a table. Processes have multiple traces and the Traces semantics records all possible traces of a process. The domain $\mathcal{T}(A)$ of the Traces semantics is the set of all subsets $T$ of $A^{*\checkmark}$ for which the following healthiness condition holds:

**T1** $T$ is non-empty and prefix closed.

The semantics (or denotation) for a process $P$ in the Traces semantics is simply the traces of the process, i.e.,

$$[\![P]\!]_{\mathcal{T}} := traces(P) \ .$$

The clauses for the *traces* function can be found in Appendix B.1.

**Example 2.2** Consider the *Mover* person as defined earlier in Section 2.1. The set of traces for the process *Mover* is

$$\{\langle\rangle, \langle move\_piano\rangle, \langle move\_table\rangle, \langle move\_piano, move\_piano\rangle, \langle move\_piano, move\_table\rangle, \\ \langle move\_table, move\_piano\rangle, \langle move\_table, move\_table\rangle, \ldots\}$$

or more precisely $\{move\_piano, move\_table\}^*$. The *Stubborn_Mover* process also has the same traces, thus the *Mover* and *Stubborn_Mover* processes are equivalent in the Traces semantics. This is because the Traces semantics is focused only on what traces a process can engage in and not on the distinction of internal and external choice.

Refinement in the Traces semantics is simply defined as reversed trace inclusion, that is,

$$P \sqsubseteq_{\mathcal{T}} Q \text{ iff } traces(Q) \subseteq traces(P)$$

The notation $P \sqsubseteq_{\mathcal{T}} Q$ is to be read as "the process $P$ refines to the process $Q$" or "$Q$ refines $P$". This notion of refinement is well suited for safety properties. If $Q$ is a refinement of $P$, then we know that $Q$'s behaviour was already included in $P$'s and thus $Q$ can do nothing new. Thus, if $P$ is safe (i.e., only performs sequences of actions deemed to be safe) then $Q$ is also safe.

The definition of refinement causes the process STOP, which has only the empty trace, to be a refinement of all processes. This fits with the notion that safety means nothing "bad" can happen. STOP is the safest process as it does nothing, thus it should be the most refined process.

### 2.4.2 The Failures/Divergences Semantics $\mathcal{N}$

The Failures/Divergences semantics (as presented by Roscoe [Ros05]) records two sets. The first set, referred to as the *failures*$_\perp$ set, consists of pairs $(s, X)$, called *failures*, where $s$ is a trace and $X$ is a *refusal set* for the trace $s$. These pairs represent that a process after executing the trace $s$ can refuse all the actions in the set $X$. The second component, the *divergences*, records all traces after which the process may diverge, that is, the process can perform an infinite sequence of internal actions. The subscript $\perp$ on the failures component represents that these failures include ones generated by unstable states, that is, states that can diverge. The Stable-Failures semantics, discussed next, does not record such failures.

The domain $\mathcal{N}(A)$ of the Failures/Divergences semantics consists of those pairs

$$(F, D), \quad \text{where } F \subseteq A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}) \text{ and } D \subseteq A^{*\checkmark},$$

satisfying the following healthiness conditions (where $s, t$ range over $A^{*\checkmark}$ and $X, Y$ over $\mathcal{P}(A^{\checkmark})$):

**F1** $tr_\perp(F, D)$ is non-empty and prefix closed,

**F2** $(s, X) \in F \wedge Y \subseteq X \implies (s, Y) \in F$,

**F3** $(s, X) \in F \wedge \forall x \in Y \bullet s^\frown\langle x\rangle \notin tr_\perp(F, D) \implies (s, X \cup Y) \in F$,

**F4** $s^\frown\langle\checkmark\rangle \in tr_\perp(F, D) \implies (s, A) \in F$,

**D1** $s \in D \cap A^* \wedge t \in A^{*\checkmark} \implies s^\frown t \in D$,

**D2** $s \in D \implies (s, X) \in F$, and

**D3** $s^\frown\langle\checkmark\rangle \in D \implies s \in D$,

where $tr_\perp(F, D) = \{s \mid \exists(s, \emptyset) \in F\}$. Roscoe [Ros05] provides extensive discussion on the development of these conditions and all the main CSP denotational semantics.

The semantics (or denotation) for a process $P$ in the Failures/Divergences semantics is the pair consisting of $P$'s failures and divergences, that is,

$$[\![P]\!]_{\mathcal{N}} := (\mathit{failures}_\perp(P), \mathit{divergences}(P)) \ .$$

The clauses for the *failures*$_\perp$ and *divergences* functions can be found in Appendix B.2.

**Example 2.3** Let $A = \{a\}$ be an alphabet and $P = (a \rightarrow \text{STOP}) \sqcap \text{DIV}$ be a process The semantics of this process (i.e., $[\![P]\!]_{\mathcal{N}}$) are $(\mathit{failures}_{\perp}(P), \mathit{divergences}(P))$ where

$$\begin{array}{lcl} \mathit{divergences}(P) & = & \{\langle\rangle, \langle a\rangle, \langle\checkmark\rangle, \langle a, a\rangle, \langle a, \checkmark\rangle, \ldots\} = A^{*\checkmark} \\ \mathit{failures}_{\perp}(P) & = & \{(\langle\rangle, \emptyset), (\langle\rangle, \{a\}), (\langle\rangle, \{\checkmark\}), (\langle\rangle, \{a, \checkmark\}), \\ & & (\langle a\rangle, \emptyset), (\langle a\rangle, \{a\}), (\langle a\rangle, \{\checkmark\}), (\langle a\rangle, \{a, \checkmark\}), \\ & & (\langle\checkmark\rangle, \emptyset), (\langle\checkmark\rangle, \{a\}), (\langle\checkmark\rangle, \{\checkmark\}), (\langle\checkmark\rangle, \{a, \checkmark\}), \\ & & (\langle a, a\rangle, \emptyset), (\langle a, a\rangle, \{a\}), (\langle a, a\rangle, \{\checkmark\}), (\langle a, a\rangle, \{a, \checkmark\}), \\ & & (\langle a, \checkmark\rangle, \emptyset), (\langle a, \checkmark\rangle, \{a\}), (\langle a, \checkmark\rangle, \{\checkmark\}), (\langle a, \checkmark\rangle, \{a, \checkmark\}), \\ & & \ldots\} = A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}) \end{array}$$

Here, the divergence causes all possible traces to be included in the divergences, which are in turn recorded with all possible refusals in the $\mathit{failures}_{\perp}$ component. The failures of the sub-process $a \rightarrow \text{STOP}$ are over-shadowed in the failures of the divergence.

Refinement in the Failures/Divergences semantics is reverse inclusion of failures and divergences, that is,

$$P \sqsubseteq_{\mathcal{N}} Q \text{ iff } \mathit{failures}_{\perp}(Q) \subseteq \mathit{failures}_{\perp}(P) \wedge \mathit{divergences}(Q) \subseteq \mathit{divergences}(P) \ .$$

The idea for this notion of refinement can be captured with the phrase "$Q$ has less internal non-deterministic choice than $P$". This means for instance, that during development (i.e., refinement) an internal choice in a system can be changed in to a deterministic one (say via a conditional construct).

The Failures/Divergences semantics allows for the analysis of deadlock and livelock freedom, both of which can be captured by this semantics. Furthermore, as deadlock and livelock freedom are preserved by the refinement notion [Ros05], we can establish such properties over development steps. For instance, if $P$ refines to $Q$ and we know $P$ is deadlock free then we can assert that $Q$ is also deadlock free. This will form the basis for deadlock analysis of CSP-CASL specifications in Chapter 9.

### 2.4.3 The Stable-Failures Semantics $\mathcal{F}$

The Stable-Failures semantics (as presented by Roscoe [Ros05]) also records two sets. The first set, the *traces*, is a trace set as in the Traces semantics $\mathcal{T}$ and just records all possible traces of a process. The second set, the *failures* set, works similar to that of the Failures/Divergences semantics, but only records stable failures. A stable failure is one in which the trace component is not a divergence. This semantics does not concern itself with recording the details of any divergences of processes. The domain $\mathcal{F}(A)$ of the Stable-Failures semantics consists of those pairs

$$(T, F), \quad \text{where } T \subseteq A^{*\checkmark} \text{ and } F \subseteq A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}),$$

satisfying the following healthiness conditions (where $s$ ranges over $A^{*\checkmark}$ and $X, Y$ range over $\mathcal{P}(A^{\checkmark})$):

**T1** $T$ is non-empty and prefix closed,

**T2** $(s, X) \in F \implies s \in T$,

**T3** $s^\frown\langle\checkmark\rangle \in T \implies (s^\frown\langle\checkmark\rangle, X) \in F$ for all $X \subseteq A^\checkmark$,

**F2** $(s, X) \in F \wedge Y \subseteq X \implies (s, Y) \in F$,

**F3** $(s, X) \in F \wedge \forall x \in Y \bullet s^\frown\langle x\rangle \notin T \implies (s, X \cup Y) \in F$, and

**F4** $s^\frown\langle\checkmark\rangle \in T \implies (s, A) \in F$.

The semantics (or denotation) for a process $P$ in the Stable-Failures semantics is the pair consisting of $P$'s traces and failures, that is,

$$[\![P]\!]_{\mathcal{F}} := (\mathit{traces}(P), \mathit{failures}(P)) \ .$$

The clauses for the *traces* function can be found in Appendix B.1, while the clauses for the *failures* function can be found in Appendix B.3.

**Example 2.4** Let $A = \{a\}$ be an alphabet and $P = (a \rightarrow \text{STOP}) \sqcap \text{DIV}$ be a process. The semantics of this process (i.e., $[\![P]\!]_{\mathcal{F}}$) are $(\mathit{traces}(P), \mathit{failures}(P))$ where

$$\begin{array}{lcl} \mathit{traces}(P) & = & \{\langle\rangle, \langle a\rangle\} \\ \mathit{failures}(P) & = & \{(\langle\rangle, \emptyset), (\langle\rangle, \{\checkmark\}), \\ & & (\langle a\rangle, \emptyset), (\langle a\rangle, \{a\}), (\langle a\rangle, \{\checkmark\}), (\langle a\rangle, \{a, \checkmark\})\} \end{array}$$

Here, the divergences are not considered and are ignored as only stable failures are recorded in the *failures* component. If we compare this to Example 2.3, we see that only considering stable states makes the denotation much smaller and in this case finite. The process $a \rightarrow \text{STOP}$ yields the same denotation in this model. This shows that the divergence in $P$ is not represented in this model and we only have a partial representation of $P$.

**Example 2.5** As a second example, consider the processes:

$$\begin{aligned} P &= a \rightarrow P \\ Q &= (a \rightarrow Q) \sqcap \text{DIV} \end{aligned}$$

over the alphabet $A = \{a\}$. These both have the same denotation $(T, F)$ where

$$\begin{array}{lcl} T & = & \{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \ldots\} = A^* \\ F & = & (\langle\rangle, \emptyset), (\langle\rangle, \{\checkmark\}), \\ & & (\langle a\rangle, \emptyset), (\langle a\rangle, \{\checkmark\}), \\ & & (\langle a, a\rangle, \emptyset), (\langle a, a\rangle, \{\checkmark\}), \\ & & \ldots\} = \{(s, X) \mid s \in A^* \wedge X \subseteq \{\checkmark\}\} \end{array}$$

This is an example where the Stable-Failures semantics cannot distinguish between a process which has a livelock and one without a livelock. In general the Stable-Failures semantics can not distinguish between a process $P$ and the process $P \sqcap \text{DIV}$ [Ros05].

Refinement in the Stable-Failures semantics is reverse inclusion of traces and failures, that is,

$$P \sqsubseteq_{\mathcal{F}} Q \text{ iff } traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P) \ .$$

The Stable-Failures semantics coincides with the Failures/Divergences semantics for all processes that are divergence free [Ros05]. Thus, we usually work in this simpler semantics when we are restricting ourselves to such divergence free processes. The normal work flow to show say deadlock freedom of a process $P$ would be first to show that $P$ is livelock free (i.e., it has no divergences) in the Failures/Divergences semantics and then show it has no deadlocks in the Stable-Failures semantics.

This concludes the discussion about the various CSP semantics that are used throughout this thesis. We now briefly look at how verification can be performed using refinement.

## 2.5 Refinement

Each of the denotational semantics presented in Section 2.4 comes equipped with a refinement notion. All of these refinement notions obey a number of laws including

$$\begin{array}{ll} P \sqsubseteq P & \text{(reflexivity)} \\ P \sqsubseteq Q \wedge Q \sqsubseteq R \Longrightarrow P \sqsubseteq R & \text{(transitivity)} \\ P \sqsubseteq Q \wedge Q \sqsubseteq P \Longrightarrow P = Q & \text{(anti-symmetry)} \end{array}$$

In the context of this thesis we write $P \sqsubseteq Q$ for $P \sqsubseteq_{\mathcal{D}} Q$ if the specific choice of $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$ does not matter. Furthermore, refinement is compositional such that it is preserved by all operations of CSP. This means for any CSP context $F(.)$ where a process can be substituted,

$$P \sqsubseteq Q \implies F(P) \sqsubseteq F(Q) \ .$$

Roscoe [Ros10] presents some relationships between these notions of refinement, specifically that $\sqsubseteq_{\mathcal{N}} \subsetneq \sqsubseteq_{\mathcal{T}}$, $\sqsubseteq_{\mathcal{F}} \subsetneq \sqsubseteq_{\mathcal{T}}$, $\sqsubseteq_{\mathcal{N}} \not\subseteq \sqsubseteq_{\mathcal{F}}$, and $\sqsubseteq_{\mathcal{F}} \not\subseteq \sqsubseteq_{\mathcal{N}}$.

We now present a short example, closely following [CRS$^+$ar], which demonstrates how refinement can be used in verification. In this example we specify a general buffer as the CSP process *Buffer*. We then specify a two element buffer $B$. Following this, we prove that $B$ is a valid buffer by showing that $B$ is a refinement of *Buffer*.

A buffer generally consists of two channels, namely a *read* channel and a *write* channel, and satisfies the following three properties:

1. Messages can be input on the *read* channel and output on the *write* channel without loss or reordering,

2. Available input of any messages on the *read* channel when the buffer is empty.

3. Available output of some message on the *write* channel when it is non-empty.

The following CSP process defines the most general buffer which satisfies these properties.

$$
\begin{aligned}
&\mathit{Buffer}(\langle\rangle) = \mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle m\rangle)\\
&\mathit{Buffer}(\langle x\rangle \frown q) =\\
&\qquad (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown q \frown \langle m\rangle) \sqcap \mathrm{STOP})\\
&\quad \Box\ \ \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(q)
\end{aligned}
$$

The buffer process $\mathit{Buffer}(\_)$ has one parameter, which is the sequence of messages currently in the queue. Variables $m$ and $x$ range over the set of allowable messages $M$, while $q$ ranges over all finite sequences over $M$.

An empty buffer always allows a new message to be read in. A non-empty buffer either accepts a new message on $\mathit{read}$ or stops working, however, it is not able to refuse to output on $\mathit{write}$. Requirement (2) does not require a non-empty buffer to be able to read an input. Hence the use of non-deterministic choice between accepting a fresh input or refusing it.

Initially, the queue of stored messages is empty:

$$\mathit{Buffer} \;=\; \mathit{Buffer}(\langle\rangle)\ .$$

This process can be seen as a specification, in that only the behaviour defined by this process, in terms of the traces and failures that it exhibits, should be allowed in any implementation. Such a specification can then be used to check whether typically larger or more complicated systems have the same behaviour. We can verify that an implementation, say a two element buffer $B$, is in fact an actual implementation of the specification by showing that the implementation refines the specification, that is

$$\mathit{Buffer} \sqsubseteq_{\mathcal{F}} B\ .$$

This is the general idea underlying the notion of *refinement*.

We now specify a concrete two element buffer $B$.

$$
\begin{aligned}
B &= \mathit{read}\,?\,x :: M \rightarrow B_{One}(x)\\
B_{One}(x) &= \mathit{read}\,?\,y :: M \rightarrow B_{Two}(x,y) \Box \mathit{write}\,!\,x \rightarrow B\\
B_{Two}(x,y) &= \mathit{write}\,!\,x \rightarrow B_{One}(y)
\end{aligned}
$$

This buffer can read and store up to two values and output them in the order they were read. While it is intuitively clear that $B$ is a two element buffer, the question is "is $B$ a valid buffer?", that is, do we have $\mathit{Buffer} \sqsubseteq_{\mathcal{F}} B$?

In order to answer this question, we state two algebraic laws of CSP, which both are immediate consequences of the semantic clauses and the definition of refinement. Over the model $\mathcal{F}$, the following laws hold for refinement:

1. $P \sqcap Q \sqsubseteq_{\mathcal{F}} Q$ (int-choice refinement)

2. $P \Box \mathrm{STOP} \sqsubseteq_{\mathcal{F}} P$ (stop refinement)

Unfolding the equations of $\mathit{Buffer}(\_)$, we obtain:

$$
\begin{array}{l}
\mathit{Buffer}(\langle\rangle) = \mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle m\rangle) \\
\mathit{Buffer}(\langle x\rangle) = \\
\qquad (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown \langle m\rangle) \sqcap \mathrm{STOP}) \\
\quad \Box \;\; \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle\rangle) \\
\mathit{Buffer}(\langle x\rangle \frown \langle y\rangle) = \\
\qquad (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown \langle y\rangle \frown \langle m\rangle) \sqcap \mathrm{STOP}) \\
\quad \Box \;\; \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle y\rangle) \\
\mathit{Buffer}(\langle x\rangle \frown q) = \qquad\qquad (\text{for } \mathit{length}(q) > 1) \\
\qquad (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown q \frown \langle m\rangle) \sqcap \mathrm{STOP}) \\
\quad \Box \;\; \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(q)
\end{array}
$$

Starting with the R.H.S of $\mathit{Buffer}(\langle x\rangle)$, we prove:

$$
\begin{array}{lll}
 & (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown \langle m\rangle) \sqcap \mathrm{STOP}) & \\
 & \Box \;\; \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle\rangle) & \\
\sqsubseteq_{\mathcal{F}} & (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown \langle m\rangle)) & \\
 & \Box \;\; \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle\rangle) & \text{by (int choice refinement)}
\end{array}
$$

Starting with the R.H.S of $\mathit{Buffer}(\langle x\rangle \frown \langle y\rangle)$, we argue:

$$
\begin{array}{lll}
 & (\mathit{read}\,?\,m :: M \rightarrow \mathit{Buffer}(\langle x\rangle \frown \langle y\rangle \frown \langle m\rangle) \sqcap \mathrm{STOP}) & \\
 & \Box \;\; \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle y\rangle) & \\
\sqsubseteq_{\mathcal{F}} & \mathrm{STOP} \,\Box (\mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle y\rangle)) & \text{by (int choice refinement)} \\
\sqsubseteq_{\mathcal{F}} & \mathit{write}\,!\,x \rightarrow \mathit{Buffer}(\langle y\rangle) & \text{by (stop refinement)}
\end{array}
$$

Thus,

$$\mathit{Buffer}(\langle\rangle) \sqsubseteq_{\mathcal{F}} B \quad .$$

Via the above refinement we have shown that $B$ is indeed a valid buffer and satisfies the specification.

This example has shown briefly how refinement can be used to verify that implementations (expressed in CSP) meet their specifications (also expressed in CSP).

## 2.6 Tool Support

CSP has various tools that support exploration and refinement proofs. Here, we briefly describe three of these tools, namely PROBE, FDR and CSP-Prover.

PROBE [Pro03] is an animator for CSP which allows the user to explore the behaviour of processes interactively. PROBE allows CSP processes to be written in a variant of CSP known as $\mathrm{CSP_M}$ (machine readable CSP) [Ros05]. This variant of CSP has a light-weight functional programming language for specifying concrete data types which are used with channels as CSP communications. $\mathrm{CSP_M}$ is describe in [FDR06] and presented in detail in [Sca98].

FDR (Failures Divergences Refinement) [FDR06] is a tool for checking refinement properties of CSP processes in the Traces, Failures/Divergences and Stable-Failures semantics. It is also capable of checking that processes are deadlock and livelock free. This is an automated tool which displays counter models (traces) for checks that fail. FDR uses the same input language as PROBE, namely $CSP_M$.

CSP-Prover [IR, IR05] is an interactive theorem prover built upon the well established theorem prover Isabelle/HOL [Pau94, NPW02]. Isabelle/HOL is a generic interactive theorem prover based on ML [MTH90]. CSP-Prover provides support for interactively proving refinement relations between CSP processes. It is generic in the CSP semantics that is used: currently it supports the Traces and the Stable-Failures semantics. There has also been work to support the use of the Stable Revivals semantics [Sam08].

In this chapter we have introduced the process algebra CSP along with its syntax and semantics which will be used within this thesis. We have presented the three main denotational semantics, namely the Trace semantics, the Failures/Divergences semantics and the Stable-Failures semantics. Finally, we briefly discussed how refinement can be used for verification and the available tool support for CSP.

# Chapter 3

# CASL

## Contents



CASL (Common Algebraic Specification Language) [Mos04, BM04] is a specification formalism developed by the COFI initiative [Mos97], through the late 1990s, as a common platform for integrating classical features of algebraic specification languages. The algebraic specification community has adopted CASL as the de facto specification language for the description of data.

Within the CASL language data can be described at high levels of abstraction (via loose semantics) and also at concrete levels of abstraction (using initial semantics). CASL also has the ability to work with partial functions, sub-sorting and sort generation constraints. Here, we describe only informally, mainly using a running example, the essence of CASL and defer the formal description of CASL until Chapter 4.

CASL allows one to specify data by providing a signature and axioms, that is, first order logic formulae with sort generation constrains. The signature declares what symbols are available, while axioms constrain the possible interpretations of the symbols. An interpretation (usually called a model) of a CASL specification is a many-sorted algebra. One CASL specification gives rise to a class of models. Each model gives an interpretation to the symbols of the specification's signature whilst satisfying the axioms of the specification. We will see full details of this in Chapter 4.

This chapter demonstrates the use of CASL for system modelling and how one can reason about CASL using automated theorem provers. Throughout this chapter we work with an intuitive understanding of $PFOL^{=}$ (Partial First Order Logic), including its signatures, for-

mulae, models and satisfaction relation $\models$ – formally $PCFOL^{=}$ ($PFOL^{=}$ with sort generation constrains) will be introduced in Section 4.2.

## 3.1 The Syntax and Semantics of Basic CASL Specifications

We illustrate CASL using example specifications of the railway domain. These examples are based on work presented in ATE [JR11] and by Sze [Sze11]. They utilise many features of CASL, although not all. The following examples follow the domain model proposed by Bjørner [Bjø09, Bjø00] which was developed in order to model the railway domain. Once the necessary components of the railway domain are specified in CASL, we construct a concrete track plan containing two routes and a platform in Section 3.4.

We start with the specification of time, which is an essential element of railway systems. We model time in the following specification:

**spec** TIME =
  **sort** *Time*
  **ops** 0 : *Time*;
    *suc* : *Time* → *Time*;
    *pre* : *Time* →? *Time*
  **pred** __<=__ : *Time* × *Time*
  ∀ *n* : *Time* • 0 <= *n*
  ∀ *n*, *m* : *Time* • *suc*(*m*) <= *suc*(*n*) ⇔ *m* <= *n*
**end**

Above is a CASL specification with the name TIME. The specification contains declarations for sort symbols, function symbols and predicate symbols along with some axioms. We have declared: a single sort symbol *Time*, a constant (a function symbol with no arguments) 0, a total function symbol *suc* from sort *Time* to *Time* , a partial function symbol *pre* from sort *Time* to *Time*, and finally a predicate named __<=__ which is a binary predicate over *Time* × *Time* and uses infix notation (where the double underscores represent the parameter positions). The intention of the declared symbols are for *suc* to increment time by a single unit, *pre* to decrement time and __<=__ to relate two times. Even though the names of the symbols have an intuitive meaning, they are just symbols and have no formal meaning. We can control their formal meaning by axioms. Here, we have specified two axioms: the first states that 0 is the smallest time, while the second states that *suc* must be monotonic with respect to the predicate __<=__.

Each specification has an associated signature. Such a signature in CASL is a five-tuple $\Sigma = (S, TF, PF, P, \leq_S)$ where $S$ is a set of sort symbols, $TF$ and $PF$ are families of sets of total and partial function symbols respectively, indexed by the argument and target sorts, $P$ is a family of predicate symbols and $\leq_S$ is a reflexive and transitive sub-sort relation. For the

above specification we have the signature $\Sigma_{\text{TIME}} = (S, TF, PF, P, \leq_S)$ where

$$\begin{array}{lcl} S & = & \{\mathit{Time}\} \\ TF & = & \{0, \mathit{suc}\} \\ PF & = & \{\mathit{pre}\} \\ P & = & \{\_\_ <= \_\_\} \\ \leq_S & = & \{(\mathit{Time}, \mathit{Time})\} \end{array}$$

Here, we have simplified the notation of CASL signatures and have omitted the index on the families of sets and instead write them as flat sets as there is no overloading on the names. Further details of signatures will be presented in Chapter 4.

Models are used to give a formal meaning to CASL specifications. Each CASL signature $\Sigma$ gives rise to a class of many-sorted algebras (also called models), denoted as $\mathbf{mod}(\Sigma)$. Each model gives interpretations to the symbols of a signature by assigning a non-empty carrier set to each sort symbol, a type correct element to the constant symbols and appropriate functions to the function symbols. Predicates are interpreted as sets of elements for which the predicate holds.

One possible model $M$ for the signature $\Sigma_{\text{TIME}}$ is the one point model where:

$$\begin{array}{lcl} M_{\mathit{Time}} & = & \{*\} \\ (0)_M & = & * \\ (\mathit{suc})_M(*) & = & * \\ (\mathit{pre})_M(*) & = & * \\ (\_\_ <= \_\_)_M & = & \{(*,*)\} \end{array}$$

Here, we use the notation $M_{\mathit{Time}}$ for the carrier set of the sort *Time* in model $M$ and $f_M$ for the interpretation of the function symbol $f$ in model $M$ (similar for predicates). We have ignored and omitted the interpretation of the sub-sort relation, this is covered in Section 3.2. We defer all further notation to Chapter 4. This model interprets the sort *Time* with the singleton carrier set containing the element $*$. As there is only one element there is no choice for the interpretation of the symbol 0 and also the function symbol *suc* which must map $*$ to $*$, as both are total functions. There is choice however for the interpretation of the partial function symbol *pre*. It can either have the same interpretation as that of *suc*, as we are allowed to interpret partial function symbols as total functions, or we could instead state that interpretation of the previous time of $*$ is undefined. Here we choose the former. We finally interpret the predicate $\_\_<=\_\_$ as the set $\{(*,*)\}$, which states $*$ is related with $*$.

Another, possibly more natural, model $N$ would be:

$$\begin{array}{lcl} N_{\mathit{Time}} & = & \{\mathbb{N}\} \\ (0)_N & = & 0 \\ (\mathit{suc})_N(n) & = & \begin{cases} n-1 & \text{if } n > 0 \\ \mathit{undefined} & \text{otherwise} \end{cases} \\ (\mathit{pre})_N(n) & = & \\ (\_\_ <= \_\_)_M & = & \{(n,m) \in \mathbb{N} \times \mathbb{N} \mid n \leqslant m\}^{\text{e}} \\ & & \{(n,m) \in \mathbb{N} \times \mathbb{N} \mid n \leqslant m\} \end{array}$$

where we interpret time as natural numbers, the constant zero as the natural number zero, the successor function as incrementation of natural numbers, the previous time function using

the minus operator on natural numbers and finally the _<=_ predicate by mirroring the $\leqslant$ predicate of natural numbers. Other interpretations for the sort *Time* include the integers, real numbers and rational numbers. There are infinitely many more possible models.

As we have discussed above, (basic) specifications have two parts: a signature $\Sigma$ and a set of axioms $\Phi$. Each specification $SP = (\Sigma, \Phi)$, like signatures, gives rise to a class of models, namely all the models of its signature which satisfy all its axioms, that is,

$$\mathbf{Mod}(SP) = \{M \in |\mathbf{mod}(\Sigma)| \mid M \models \Phi\}$$

where $M \models \Phi$ if and only if $M \models \varphi$ for all $\varphi \in \Phi$.[1]

The models $M$ and $N$ above both satisfy the two axioms in the specification TIME, and thus are models of it. This shows how axioms in specifications can be used to restricted the models to a reasonable class that the specifier wants to allow. We could restrict the amount of models further by adding further axioms to the specification TIME, say to control the interpretation of the sort *Time* or for forcing the function *pre* to be the partial inverse of *suc*. For now we really only need that the sort *Time* exists and do not make any further restrictions.

In this example we have specified time in a loose fashion, where we only restricted the interpretation of the constant 0, the function *suc* and the predicate $<=$. This means that we do not pin down the model to a single model, but instead allow many models. We could fully specify time so that there is only a single model (up to isomorphism), however this loose specification of time is enough for us to demonstrate several features of CASL and to carry out some proof. The ability to specify using loose semantics in this way will be an important topic throughout this thesis.

Now that we have seen a basic CASL specification and some models, we commence with specifying our railway example which utilises, among other features, CASL sub-sorting.

## 3.2 Sub-sorting in CASL

We use an example from the railway domain (following the style of the domain model proposed by Bjørner [Bjø09, Bjø00]) for our presentation of further features of CASL. To this end, we describe aspects of railways using the sub-sorting features of CASL. We capture basic concepts and features of the railway domain. Concretely, we capture the notion of linear units (i.e., tracks), switches (also called points), junctions and the ability to join these together using so called connectors. We explain each of these elements as we describe the following specification:

**spec** RAILWAYELEMENTS =
  TIME
**then sorts** *Connector*;
  *Linear*, *Switch* < *Unit*
  **pred** *_has_connector_* : *Unit* × *Connector*
  **ops** *c1*, *c2* : *Unit* → *Connector*;

[1] **mod** is a functor which produces a category of models for each signature. The function $|\_|$ projects out the objects of a category and forgets about the morphisms. This is described in Chapter 4.

$c3 : Switch \to Connector$

**free type** $State ::= occ \mid unocc$

**free type** $Position ::= reverse \mid normal \mid move$

**ops** $\_\_state\_at\_\_ : Unit \times Time \to State$;
$\_\_position\_at\_\_ : Switch \times Time \to Position$

$\forall s : Switch; l : Linear \bullet \neg s = l$

$\forall u : Unit \bullet \neg c1(u) = c2(u)$

$\forall s : Switch \bullet \neg c3(s) = c1(s) \wedge \neg c3(s) = c2(s)$

$\forall l : Linear; c : Connector \bullet l\ has\_connector\ c \Leftrightarrow c = c1(l) \vee c = c2(l)$

$\forall s : Switch; c : Connector \bullet s\ has\_connector\ c$
$\Leftrightarrow c = c1(s) \vee c = c2(s) \vee c = c3(s)$

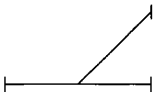
**end**

This specification, named RAILWAYELEMENTS, captures all the elements (that we are interested in) of the railway domain. It first imports the specification of time that we declared earlier (keyword **then**). This import is one of the most basic forms of compositional modelling and has the obvious meaning of importing the symbols and axioms available in the specification TIME. We then declare four sorts namely, *Connector*, *Linear*, *Switch* and *Unit*. *Linear* and *Switch* are sub-sorts of *Unit* (indicated by the $<$ symbol).

Units are a basic element of the railway. Units can either be linear units (represented by the sort *Linear*) or switches. A linear unit can be considered as a simple piece of track that has two ends with a connector on each end. A linear unit can be depicted as:

where the small bars represent the connectors.

A switch is a mechanism on the railway to move trains between tracks, also known as a point. Switches have three ends, each of which has an attached connector. A switch can be depicted as:



Linear units can be placed and connected together along with switches to form so called track plans. The sort *Connector* represents connectors that allow linear units and switches to be connected together. There must be a single connector between any linear units or switches that are connected, that is, connected units share their connectors.

Sub-sorting allows one sort to be "included" as part of another. On the semantical level, however, this is not done via subset inclusion, but instead by explicit embedding and projection functions. By declaring that one sort $s$ is a sub-sort of another sort $t$, we actually declare three symbols:

1. an injection function from sort $s$ to sort $t$,

2. a partial projection function from sort $t$ to sort $s$, and

3. a membership predicate that tests whether elements of sort $t$ have a counterpart in sort $s$.

As *Switch* is a sub-sort of *Unit*, we have an injection function which can cast a switch to a unit. Explicit casting is usually not required in formulae as the CASL framework (and associated tools) deal with it automatically. We can simply consider a switch to be a specialised unit when writing formulae and axioms.

Next, we declare a predicate *__has_connector__* which allows units and connectors to be related: each unit has a number of connectors (linear units will have two, while switches will have three). Furthermore, we declare five total functions. The first two functions, *c1* and *c2*, allow us access to the two connectors that a linear unit has. The next function *c3* allows us access to the third connector only present on switches. As switches are also units (i.e., the sort *Switch* is a sub-sort of *Unit*) we can also apply the functions *c1* and *c2* to switches.

Following this, we define some additional sorts, however to do this we use a free type. The first free type is *State* which is going to be used to record the state of a unit. A unit can either be occupied by a train or unoccupied. There are two constants *occ* and *unocc*, representing occupied and unoccupied respectively, that are declared at the same time as *State* using a free type. The free type also forces all interpretations of *State* to have exactly two carrier set elements, one for occupied and one for unoccupied. Thus, the free type is just short hand (in this case) for expressing that there are two constants, all carrier set elements are accessible via these two constants and finally, that these two constants must have different values.

Next, we define another free type that captures the different positions of a switch. A switch can either be in the *normal* position where a train can continue on in a straight line or the *reverse* position where the train is diverted off the straight track, or finally in a *move* state where it is currently moving between the previous two states.

We then define two further operations, *__state_at__* and *__position_at__*, which allow us to observe the state of a unit and the position of a switch at certain points in time.

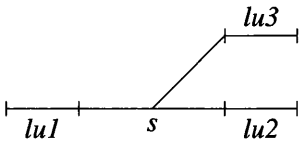
We now add axioms to control these interpretations and forbid unreasonable ones. The first axiom states that a linear units is never a switch and vice versa. There still may be units which are neither linear units nor switches – we do not forbid this here which is why we did not use a free type. This allows us to, if we wished, import this specification and add more types of units as sub-sorts. We do not do this in this example. The next two axioms state that the all the connectors on a linear unit or switch are unique. This prevents us from connecting a unit or switch to itself.

Next we couple the predicate symbol *__has_connector__* with the connector symbols *c1*, *c2* and *c3*. We require *__has_connector__* to only be true for connectors that are actually attached to the units according to the observation functions *c1*, *c2* and *c3*. This forces linear units to have two connectors and switches to have three connectors.

This completes the specification of our railway elements. There are many more restrictions and elements that we can model, but this is all that is required for the intended verification demonstrated by this example.

## 3.3 Structuring and Parametrisation

In the railway domain it is quite common to find units laid out in what is called a junction. That is a switch with three connected linear units, depicted as:




where $s$ is the switch and *lu1*, *lu2* and *lu3* are the associated linear units.

This construction is so common that it is useful to model this as a separate specification both for methodological reasons (i.e., reuse of code) and theorem proving support (see work by James et al. [JR11]). To do this, we make use of compositional modelling via parametrised (also called generic) CASL specifications. We first form a parametrised junction specification, followed by, instantiating it multiple times to form many junctions which may then be interconnected.

The following is a generic CASL specification that models a junction.

**spec** JUNCTION
  [**ops** $lu1, lu2, lu3 : Linear$;
    $s : Switch$
  • $\neg\ lu1 = lu2$
  • $\neg\ lu2 = lu3$
  • $\neg\ lu1 = lu3$]
  **given** RAILWAYELEMENTS =
  **preds** $route\_normal\_available\_at\_\_ : Time$;
    $route\_reverse\_available\_at\_\_ : Time$
  • $c2(lu1) = c1(s)$
  • $c2(s) = c1(lu2)$
  • $c3(s) = c1(lu3)$
  $\forall\ t : Time$ • $route\_normal\_available\_at\ t$
    $\Leftrightarrow\ s\ state\_at\ t = unocc \wedge s\ position\_at\ t = normal$
      $\wedge\ lu1\ state\_at\ t = unocc \wedge lu2\ state\_at\ t = unocc$
  $\forall\ t : Time$ • $route\_reverse\_available\_at\ t$
    $\Leftrightarrow\ s\ state\_at\ t = unocc \wedge s\ position\_at\ t = reverse$
    $\wedge\ lu1\ state\_at\ t = unocc \wedge lu3\ state\_at\ t = unocc$
**then %implies**
  $\forall\ t : Time$ • $\neg\ (route\_normal\_available\_at\ t \wedge route\_reverse\_available\_at\ t)$
    %(safety_property_two_routes_are_not_enabled_at_the_same_time)%$_e$%
**end**

This generic specification, with the name JUNCTION, has two main parts, namely, its *formal parameter* and its body. We describe each in turn. The idea is that an *actual parameter* can

later be used in an instantiation to fill in the formal parameter and thus create a specific instance of a junction.

Formal parameters are CASL specifications. They are written within square brackets and are commonly specified inline without naming them (as is done here). This specification declared the elements needed for a junction. We state that there are three linear units (*lu1*, *lu2* and *lu3*) and a switch (*s*). There are also three axioms which ensure that the linear units are all unique. These axioms must be implied by any actual parameter during instantiation. Therefore, the axioms can be thought of as assumptions which the formal parameter makes that the junction is built relatively to. Any actual parameter must guarantee these conditions in order to construct a specific instance of a junction. The keyword **given** allows the formal parameter to make use of the symbols available in RAILWAYELEMENTS and is similar to an import (keyword **then**).

The next part of the specification is the body. We can use any symbols in the formal parameter and in the specification RAILWAYELEMENTS which has been essentially imported. Here, we declare two predicates. The first predicate *route_normal_available_at__* is true when a train can travel in either direction from linear unit *lu1* to linear unit *lu2* (see the depiction above). The second predicate *route_reverse_available_at__* is true when trains can travel from linear unit *lu1* to linear unit *lu3*, again in either direction. The first three axioms connect the linear units and switch in the required way (mirroring the depiction above). The next two axioms state when the two routes are enabled. The first route from *lu1* to *lu2* (i.e., the normal route) is only enabled when the bottom tracks (*lu1* and *lu2*) and the switch *s* are all unoccupied and when the switch *s* is in its normal position. Similarly, the second route from linear unit *lu1* to *lu3* (i.e., the reverse route) is only enabled when the linear units *lu1* and *lu3* and the switch *s* are all unoccupied and when the switch *s* is in its reverse position.

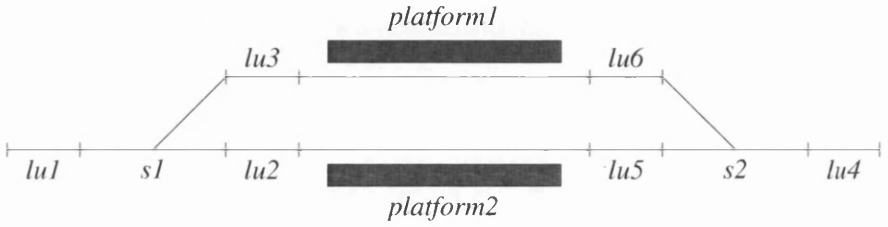
Now that we have specified our generic junction, we provide a property (i.e., a lemma) which shall hold for the whole model class, that is, it should hold for all possible instantiated junctions. The property we provide is a safety property that says that both routes are not available at the same time, that is, there are no conflicting routes available simultaneously. If there were then this would be an unsafe situation that could possibly lead to the collision of trains. If this property did not hold then it would either indicate that we have an inherent flaw in the specification or that we made a mistake in the specification. It is also possible that we have not provided enough information (axioms) to prove this. Such properties can be expressed in CASL specifications via the keywords **then %implies**. This allows us to make an extension of the specification where we do not add any new signature elements but instead only add new axioms. Whilst these axioms still constrain the model class of a specification, they are intended to be implied by the earlier axiom base, that is, it is intended that the previous axioms already constrain the model class in such a way as all models actually fulfil this axiom. The keyword **%implies** is a special comment that indicates to tools that the axioms in the extension should be implied by the existing axioms, that is, it sets up proof obligations in tools. The %(safety_property_two_routes_are_not_enabled_at_the_same_time)% text is a special comment that allows us to label the axiom with a name, which is useful for identifying axioms in tools. We discuss tool support for proving such an implication in Section 3.5.

This concludes the first part of our CASL specification of the railway. We have specified all the necessary components and created a generic specification for railway junctions. We have

also added an axiom which acts as a validation check that our specification is meaningful. Next we will instantiate this generic junction specification twice to form our track plan.

## 3.4 Instantiation

We now instantiate the junction specification twice to form two junctions which we interconnect to form a small track plan as shown below:




The idea here is that we have two major routes, where both routes enter at the far left and exit at the far right (or vice versa, as we do not consider direction) and go through either *platform1* or *platform2*. We do not actually model the platforms and consider them only as linear units, although we could have an explicit sort for platforms and predicates which link platforms to linear units etc. Trains should be able to use both routes, however both routes should not be available simultaneously as this again would constitute an unsafe situation.

The first step in accomplishing this is to form our parameter specifications (i.e., our actual parameters):

**spec** JUNCTIONONEPARAM =
RAILWAYELEMENTS
**then ops** $lu1, lu2, lu3 : Linear$;
$s1 : Switch$
• $\neg\ lu1 = lu2$
• $\neg\ lu1 = lu3$
• $\neg\ lu2 = lu3$
**end**

**spec** JUNCTIONTWOPARAM =
RAILWAYELEMENTS
**then ops** $lu4, lu5, lu6 : Linear$;
$s2 : Switch$
• $\neg\ lu4 = lu5$
• $\neg\ lu4 = lu6$
• $\neg\ lu5 = lu6$
**end**

Here, we have specified two junction parameter specifications each with a switch and three linear units. The instantiation will make sure these units are connected in the correct way and

place the appropriate routes over them. The actual parameter specifications must capture all the symbols in the formal parameter specification JUNCTIONPARAM (from Section 3.3) although with possibly different names for the symbols. Thus, we must have a switch and three linear units. We also have three axioms which state that the linear units are unique, these will play a role shortly. We now have the required signature elements for the instantiation.

The linear units, switches and predicates used in both parameter specifications must use different names as all symbols later on will be imported (via instantiation) into a single specification. CASL uses the notion of "same name, same thing". Thus, if the elements had the same name in each of the specifications, then after the import we would not have different entities and would not be able to form our intended track plan with two separate junctions.

We form the overall track plan as a structured specification with multiple extensions as follows:

**spec** TRACKPLAN =
  RAILWAYELEMENTS
**then** JUNCTION[JUNCTIONONEPARAM **fit** $s \mapsto s1$]
  **with** *route_normal_available_at__* $\mapsto$ *route_s1_normal_available_at__*,
    *route_reverse_available_at__* $\mapsto$ *route_s1_reverse_available_at__*
**and** JUNCTION [JUNCTIONTWOPARAM **fit** $lu1 \mapsto lu4$, $lu2 \mapsto lu5$, $lu3 \mapsto lu6$, $s \mapsto s2$]
  **with** *route_normal_available_at__* $\mapsto$ *route_s2_normal_available_at__*,
    *route_reverse_available_at__* $\mapsto$ *route_s2_reverse_available_at__*
**then ops** *platform1*, *platform2* : *Linear*
  • $c2(lu3) = c1(platform1)$
  • $c2(platform1) = c1(lu6)$
  • $c2(lu2) = c1(platform2)$
  • $c2(platform2) = c2(lu5)$
**then preds** *route_platform1_available_at__* : *Time*;
    *route_platform2_available_at__* : *Time*
  $\forall t : Time \bullet route\_platform1\_available\_at\ t$
    $\Leftrightarrow platform1\ state\_at\ t = unocc$
      $\wedge\ route\_s1\_reverse\_available\_at\ t \wedge route\_s2\_reverse\_available\_at\ t$
  $\forall t : Time \bullet route\_platform2\_available\_at\ t$
    $\Leftrightarrow platform2\ state\_at\ t = unocc$
      $\wedge\ route\_s1\_normal\_available\_at\ t \wedge route\_s2\_normal\_available\_at\ t$
**then %implies**
  $\forall t : Time \bullet \neg\ (route\_platform1\_available\_at\ t \wedge route\_platform2\_available\_at\ t)$
    %(safety_property)%
**end**

Here, we have used a variety of structuring mechanisms namely, importation of specifications (keyword **then**), union of specifications (keyword **and**), renaming of specifications (keyword **with**) and instantiation. We start by importing the railway elements and then instantiating the junction with our first parameter specification. When instantiating generic specifications we must provide a signature morphism (keyword **fit**) from the formal parameter (i.e., JUNCTIONPARAM) to the actual parameter (i.e., JUNCTIONONEPARAM). Specifying full signature

morphisms would quickly become tedious, however CASL alleviates this with the use of *symbol maps*. Such symbol maps are shorthand for full signature morphisms where the tools are expected to try to deduce the missing pieces. CASL states that a symbol map is only well formed when it uniquely induces a signature morphism, if this is not the case then there is an error in the specification (see [Mos04] for full details). Thus, we only need to map the switch $s$ from the formal parameter to the switch *s1* in the actual parameter (via the text $s \mapsto$ *s1*). The linear units have the same names (in the formal parameter and the actual parameter JUNCTIONONEPARAM), thus this part of the signature morphism regarding these symbols is trivially deduced and does not need to be explicitly defined.

The instantiation comes with a proof obligation that the model class of the actual parameter is contained within the model class of the formal parameter. Formally, we have to prove that the formal parameter refines to the actual parameter over the signature morphism induced by the symbol map. This amounts to checking that the axioms of the formal parameter are implied by the actual parameter (after translation along the signature morphism). Thus, if we did not specify in JUNCTIONONEPARAM that the linear units were unique we could not fulfil this duty and would have unprovable proof obligations and thus an ill-formed instantiation. This proof obligation can be once again be discharged via tool support discussed in Section 3.5.

This first instantiation creates two new routes over the instantiated junction, that is, the predicates *route_normal_available_at__* and *route_reverse_available_at__*. However, when we instantiate our second junction we will also get two new routes with the same name. As CASL uses the notion of "same name same thing", we must be careful to rename these predicates in order to keep them separate. To this end, we rename these predicates to names that reflect that they are routes over the switch *s1* using the keyword **with**.

Next, we instantiate the junction a second time however this time with the second junction parameter specification. In this specification our linear units names do not match and thus must also be explicitly mapped over, for example, *lu1* is mapped to *lu4*. After this we again rename the produced routes such that their names reflect that they run over the switch *s2*. At this point we have the following track plan:


with four routes, two for each switch.

We now need to link the junctions together. We do this again by extending the specification and provide two new linear units (i.e., the platforms) and connect them in the appropriate way. Finally, we define two routes over the new track plan, one route for each platform. Each route requires the sub-routes over the junctions to be available as well as the respective platform. We have now completed our specification of our track plan with two routes.

Similarly to how we added a simple safety property at the junction level, we now add an implication which will verify that a safety property holds at the track plan level. Here, we check that both platforms are not accessible at the same time. This implication can be checked with tool support discussed in Section 3.5.

We could now continue extending the specifications above and add dynamic elements. For instance, we could add trains, signals, signalling rules: for when signals should show certain aspects (colours), etc. However, we are moving out of the area of *data* specification and are moving into the area of *behaviour* or processes specification. Whilst this can be done in CASL, the resulting specifications are not necessarily succinct or elegant and can be hard to read and understand. Process algebras such as CSP (discussed in Chapter 2) are much better equipped to capture these types of dynamics. This is why CSP-CASL was created: to capture the best parts of both CASL and CSP and to unite them. CSP-CASL is discussed in Chapter 5.

## 3.5 Tool support

As a conclusion to this chapter, we briefly discuss tool support for CASL. HETS (Heterogeneous Tool Set) [MML07] is a proof management tool centred around CASL. It supports parsing and static analysis of specifications written in CASL and related languages. Proof obligations in HETS may be discharged by utilising several external theorem provers with which HETS interfaces.

HETS uses development graphs to track the structural information within specifications, for instance, importation and renaming. Figure 3.1 shows the development graph after the specifications have been loaded. The nodes represent the specifications while the coloured arrows represent relationships between them. The black arrows are imports, the red arrows are proof obligations (either resulting from instantiations or implied axioms) and the purple arrows represent non-interesting chains of imports which have been collapsed into a single arrow (these may be expanded). Such chains have been formed via our use of renaming and union, each operation creates new unnamed nodes and arrows between them. We have four red arrows resulting from the two instantiations and two implications.

Figure 3.2 shows HETS after we have localised the proof obligations. The localisation of proof obligations (red arrows) in HETS is a necessary step before external theorem provers can be used. This step localises all the axioms and proof obligations into single specifications so that they can be translated into the input language of a number of theorem provers. In the figure we have already discharged the proof obligations for the instantiations and the proof obligation from the implication in the generic junction, thus we have only our overall safety property left open. The red node (ellipse) in the deepest window of Figure 3.2 represents a specification with this open proof obligation.

The foreground window is HETS' theorem prover interface. This allows us to attempt to discharge the remaining proof obligation via various theorem provers. The open proof goals are shown in the upper left area, here we see that we only have the safety property resulting from our axiom safety_property left open. The lower left portion of the window shows us the available axioms to use as a base for the proof (most are unnamed as we did not label them in the previous specifications). One however is named, that is, safety_property_two_routes_are_not_enabled_at_the_same_time. As we have already proven this property we can use it to help us discharge the remaining proof obligation. This is one instance of compositional reasoning that is possible in CASL. The right area displays various choices including the choice of which theorem prover to use.

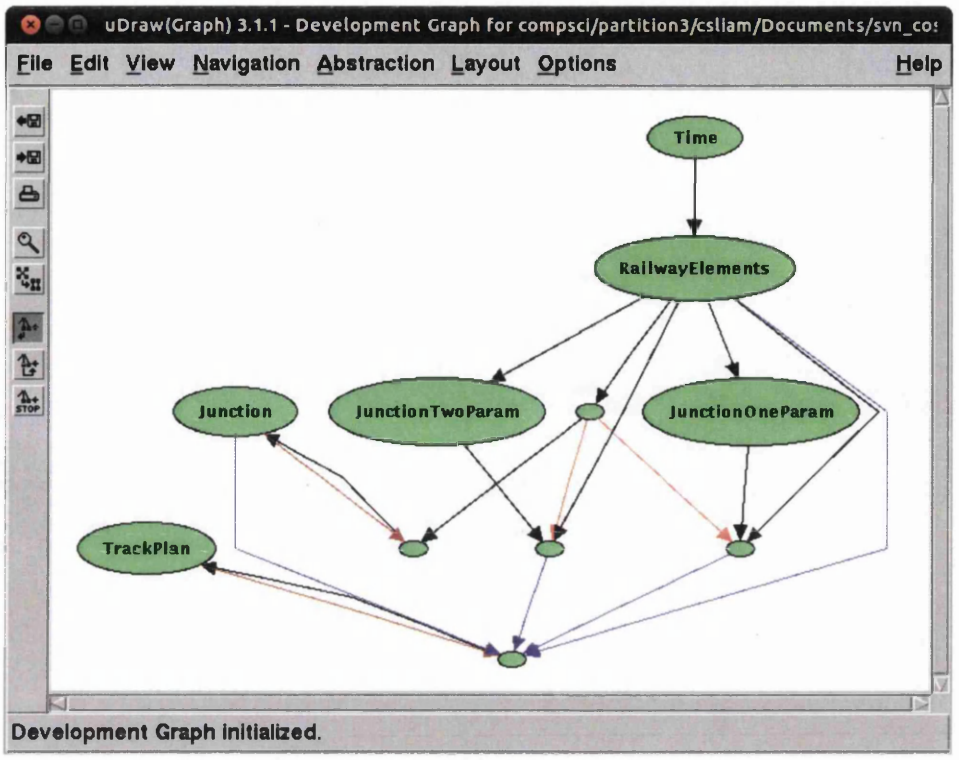



Figure 3.1: A screen-shot of HETS, after specifications have been loaded, showing the resulting development graph.

We choose to use the automated first order logic theorem prover SPASS [WBH+02]. HETS translates the axiom base and the proof obligation into SPASS' input language and then pass them onto SPASS. SPASS is capable of proving such an implication automatically and almost instantly. After SPASS proves the implication, the red specification node turns green representing that there are no further open proof obligations. Automated theorem provers, such as SPASS, can provide proof scripts for successfully proofs. These scripts can be difficult to read and understand as they are encoded and not written as a mathematician would write one. Figure 3.3 shows part of the proof script produced for the proof of the safety property.

This chapter has introduced various aspects of CASL. We have shown multiple examples of its syntax and discussed the meaning without looking too deeply at the formal semantics. We have outlined how CASL specifications give rise to model classes and how such classes can be restricted via axioms. We have briefly seen some compositional modelling features, such as the structuring mechanisms of CASL, including: importation, union, renaming, parametrisation and instantiation. Finally, we briefly looked at tool support and compositional reasoning, for the verification of CASL specifications.

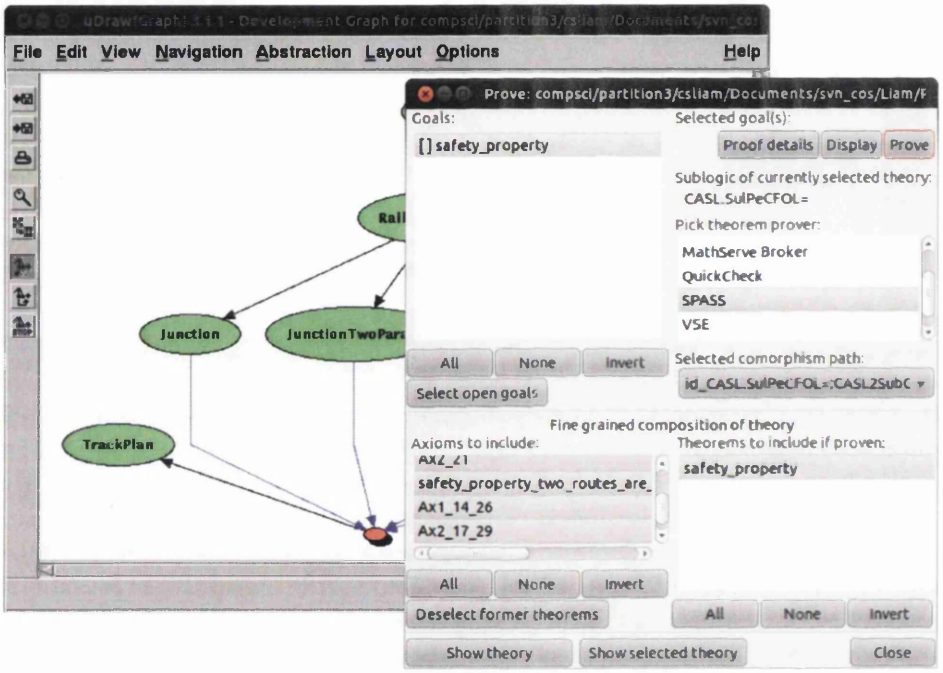



Figure 3.2: A screen-shot of HETS after specifications have been loaded, proof goals localised, all but one proof obligation discharged, and an open proof obligation selected.

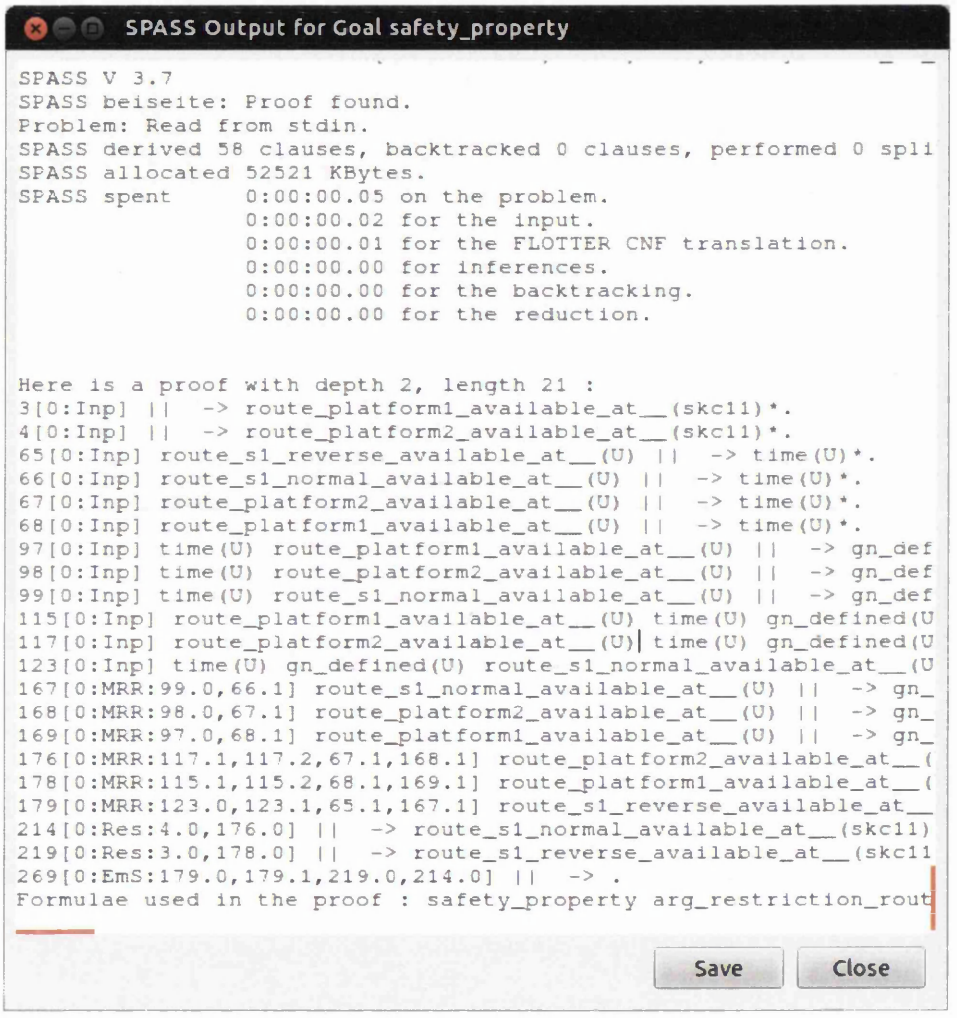



Figure 3.3: A screen-shot of HETS showing the SPASS proof script for the safety property.

*Chapter 4*

# A Common Framework: Institutions

## Contents



Logical systems share common features, including the notion of symbols or atomic elements which make up sentences or formulae. Models or interpretations assign meanings to such symbols. Finally, logics state which formulae hold in which models. All logical systems have these notions in one form or another. Institutions capture these notions in a formal framework based on category theory. The tool HETS [MML07] (see Chapter 3) is heavily based on institutions and implements various logics (centred around CASL).

This chapter first introduces institutions [GB92], followed by presenting two standard institutions relating to CASL, namely, *PCFOL*$^{=}$ and *SubPCFOL*$^{=}$ (as presented in [O'R08]), the last being the logical system underlying CASL. Following this we present a restricted version of *PCFOL*$^{=}$, followed by the data-logic of CSP-CASL. This data-logic will be used in our formalisation of CSP-CASL as institutions in Chapter 8. We then present CSP as institutions, illustrating that institutions are not restricted to algebraic specification. Finally, we conclude by presenting a kernel language for structuring which is institution independent.

## 4.1 The Formal Definition of Institutions

*Institutions* [GB92] are a theoretical framework based on category theory for the description of logics. They were first presented by Goguen and Burstall in the late 1970's as a response to the growing number of logical systems that were being developed. Institutions capture the very essence of logical systems and make it possible to create specification languages, proof calculi and tools which are completely independent of the underlying logical system. HETS, for instance, is constructed in this way. Even though it is currently centred around CASL, any institution can be plugged in.

Institutions can be related to one another by so called *institution representations* [Mos02]. These allow signatures, sentences and models to be translated between institutions. This makes it possible to borrow the satisfaction relations from related institutions. This has the practical effect that we can use theorem provers (say Isabelle/HOL) for any institution (say the CASL family) which have a chain of institutions representations in to the underlying logic of the theorem prover [CM97, Mos02].

Goguen and Burstall sum up the idea and essence of an institution in the following slogan

> "Truth is invariant under change of notation" [GB92].

That is, truth is independent of the symbols we use to express our reasoning. Hence, if we have a logical statement and replace all occurrences of symbols (such as variables, function symbols and relation symbols) with different symbols (in a consistent manner) then our new statement has the same meaning as the original.

The formal definition of institutions relies on category theory (see, for example, [ML98, Fia05]), although only basic concepts are used. 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 formally captures the above slogan.

We follow here [Mos02] where Mossakowski defines an institution $I$ as a quadruple $(\mathbf{SIGN}^I, \mathbf{sen}^I, \mathbf{mod}^I, \models^I)$ where:

- $\mathbf{SIGN}^I$ is the signature category.
- $\mathbf{sen}^I : \mathbf{SIGN}^I \to \mathbf{SET}$ is the sentence functor, where **SET** is the category where objects are sets and morphisms are total functions between sets.
- $\mathbf{mod}^I : (\mathbf{SIGN}^I)^{op} \to \mathbf{CAT}$ is the model functor, where **CAT** is the category where objects are categories and morphisms are functors between categories.[1]
- $\models^I_\Sigma \subseteq |\mathbf{mod}^I(\Sigma)| \times \mathbf{sen}^I(\Sigma)$ is the satisfaction relation, for each $\Sigma : \mathbf{SIGN}^I$,

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

$$\mathbf{mod}^I(\sigma)(M') \models^I_\Sigma \varphi \iff M' \models^I_{\Sigma'} \mathbf{sen}^I(\sigma)(\varphi)$$

[1] Some authors have concerns over the size of the category **CAT**. We do not consider such concerns within this thesis.

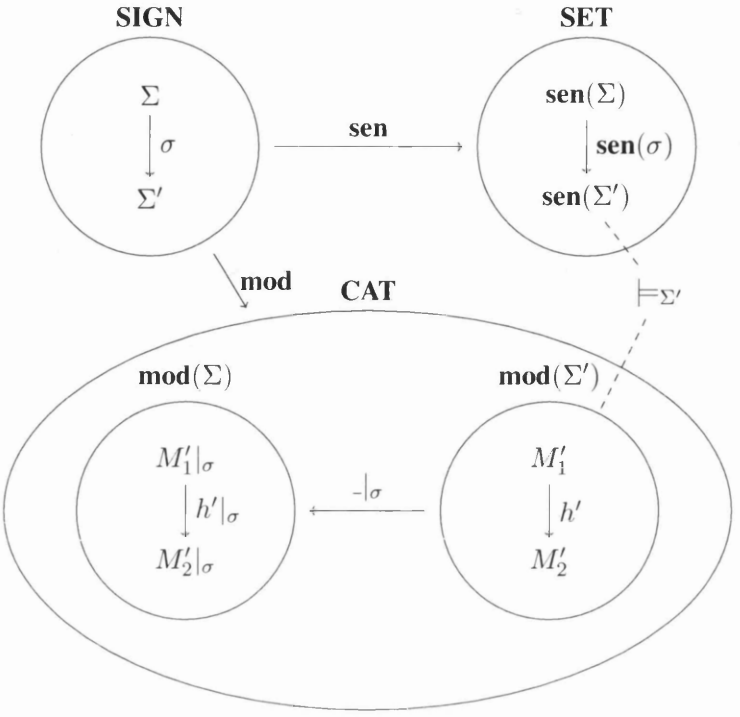



Figure 4.1: Diagram of the notion of an institution [O'R08].

holds for every sentence $\varphi \in \mathbf{sen}^I(\Sigma)$ and for every $\Sigma'$-model $M' \in |\mathbf{mod}^I(\Sigma')|$. The operation $|\_|$ when applied to a category results in the class of objects of that category, where the morphisms of the category are simply forgotten. For example $|\mathbf{SIGN}^I|$ is the class of signatures within the institution $I$. The operation $\_|^{op}$ when applied to a category gives the duel category which has the same objects and morphisms, but where the morphisms have been reversed. That is, if $\sigma : \Sigma \to \Sigma'$ is a morphism in **SIGN**, then $\sigma : \Sigma' \to \Sigma$ is a morphism in $\mathbf{SIGN}^{op}$. Figure 4.1 shows a diagram representation of an institution.

The idea here is to have a collection of signatures and signature morphisms which map symbols in a compatible way. This collection is the category $\mathbf{SIGN}^I$. Nothing else about the structure of the category **SIGN** is assumed. The top left of Figure 4.1 depicts the category **SIGN** with two signatures $\Sigma$ and $\Sigma'$ along with a signature morphism $\sigma$ between them.

The functor $\mathbf{sen}^I : \mathbf{SIGN}^I \to \mathbf{SET}$ gives for each signature $\Sigma \in |\mathbf{SIGN}^I|$, the set of sentences $\mathbf{sen}^I(\Sigma)$ over the signature $\Sigma$, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the sentence translation function $\mathbf{sen}^I(\sigma) : \mathbf{sen}^I(\Sigma) \to \mathbf{sen}^I(\Sigma')$ which translates sentences built over $\Sigma$ to sentences built over $\Sigma'$. The top right of Figure 4.1 depicts the category **SET** with two sets of sentences $\mathbf{sen}(\Sigma)$ and $\mathbf{sen}(\Sigma')$ along with a sentence morphism $\mathbf{sen}(\sigma)$ between them.

The functor $\mathbf{mod}^I : (\mathbf{SIGN}^I)^{op} \to \mathbf{CAT}$ gives for each signature $\Sigma \in |\mathbf{SIGN}^I|$ the category $\mathbf{mod}^I(\Sigma)$ of $\Sigma$-models and model morphisms, and for each signature morphism

$\sigma : \Sigma \to \Sigma'$ the reduct functor $\mathbf{mod}^I(\sigma) : \mathbf{mod}^I(\Sigma') \to \mathbf{mod}^I(\Sigma)$. The reduct functor $\mathbf{mod}^I(\sigma)$ reduces models over the signature $\Sigma'$ to models over the signature $\Sigma$. Similarly, model morphisms are reduced to model morphisms between reduced models. Morphism composition is reversed within this category as $\mathbf{mod}^I$ is a contravariant functor.

The lower half of Figure 4.1 depicts the category **CAT**. Depicted within this category are two categories of models, namely $\mathbf{mod}(\Sigma)$ and $\mathbf{mod}(\Sigma')$. Within the category $\mathbf{mod}(\Sigma')$ there are two models $M'_1$ and $M'_2$ along with a model morphism $h'$ between them. The reducts of these models and the reduct model morphism are depicted within the category $\mathbf{mod}(\Sigma)$ as $M'_1|_\sigma$, $M'_2|_\sigma$ and $h'|_\sigma$ respectively.

The satisfaction condition ensures that satisfaction with respect to the satisfaction relation, is preserved across translation of sentences and reducts of models.

We introduce some shorthand notations that are often used when dealing with institutions. We write $\sigma(\varphi)$ for $\mathbf{sen}^I(\sigma)(\varphi)$ and $M'|_\sigma$ for $\mathbf{mod}^I(\sigma)(M')$. Also the subscript on the satisfaction relation and the superscript $I$ may be omitted when it is clear from the context and no confusion arises. These are the most common shorthand notations as defined by Mossakowski in [Mos02], which are slightly different to those defined by Goguen and Burstall in [GB92].

With these shorthands the satisfaction condition becomes: for each signature morphism $\sigma : \Sigma \to \Sigma'$ in **SIGN**,

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

for each $\Sigma'$-model $M' \in |\mathbf{mod}^I(\Sigma')|$ and $\Sigma$-sentence $\varphi \in \mathbf{sen}^I(\Sigma)$.

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

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

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

## 4.2 The Institution *PCFOL*$^=$

We now outline the institution of *partial first order logic with sort generation constraints and equality* as it is defined in [Mos02]. While this stands as its own institution, this also forms the basis for the institution *SubPCFOL*$^=$ which underlies CASL.

### 4.2.1 Signatures

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

- a set $S$ of sort symbols,
- two $S^* \times S$-sorted families, $TF = (TF_{w,s})_{w \in S^*, s \in S}$ of *total function symbols* and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of *partial function symbols*, 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 a finite string $w = \langle s_1, \ldots, s_n \rangle$ and sets $M_{s_1}, \ldots, M_{s_n}$, we write $M_w$ for the Cartesian product $M_{s_1} \times \ldots \times M_{s_n}$.

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 $\sigma^P : P_w \to P_{\sigma^{S*}(w)}$.

Identities and composition are defined in the obvious way. This gives us the category of $PCFOL^=$-signatures.

### 4.2.2 Models

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

- a non-empty carrier set $M_s$ for each sort symbol $s \in S$,
- a partial function $(f_{w,s})_M$ from $M_w$ to $M_s$ (also written just $f_M$) 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 $(p_w)_M \subseteq M_w$ (also written just $p_M$) for each predicate symbol $p \in P_w$.

A *many-sorted $\Sigma$-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, where $w = \langle s_1, \ldots, s_n \rangle$, we have

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

and for all $p \in P_w$ and $(a_1, \ldots, a_n) \in M_w$, where $w = \langle s_1, \ldots, s_n \rangle$, we have

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

Let $\sigma : \Sigma \to \Sigma'$ be a a many-sorted signature morphism and $M'$ be a $\Sigma'$-model. Then the *reduct* $M'|_\sigma$ of $M'$ is the $\Sigma$-model $M$ with

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

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

Identities and composition are defined in the obvious way, for full details see [Mos02]. This gives us the functor **mod**.

### 4.2.3 Sentences

Given a many-sorted signature $\Sigma = (S, TF, PF, P)$, a *variable system* over $\Sigma$ is an $S$-sorted, pairwise disjoint family of variables $X = (X_s)_{s \in S}$. The sets $T_\Sigma(X)_s$ of *many-sorted $\Sigma$-terms* of sort $s \in S$, with variables in $X$ is the least set 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 $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1 \ldots n)$, $f \in TF_{w,s} \cup PF_{w,s}$, and $w = \langle s_1, \ldots, s_n \rangle$.

The set $AF_\Sigma(X)$ of *many-sorted atomic $\Sigma$-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 $t_i \in T_\Sigma(X)_{s_i}$, $p_w \in P_w$, $w = \langle s_1, \ldots, s_n \rangle \in S^*$ (predicates),
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 $\Sigma$-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 $\varphi \in FO_\Sigma(X \cup \{x : s\})$, $s \in S$,

We omit brackets whenever this is unambiguous and use the usual abbreviations: $\neg\varphi$ for $\varphi \Rightarrow F$, $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, $T$ for $\neg F$ and $\exists x : s \bullet \varphi$ for $\neg\forall x : s \bullet \neg\varphi$.

A sort generation constraint states that some set of sorts is generated by some set of functions. Technically, sort generation constraints also contain a signature morphism component; this is needed to be able to translate them along signature morphisms without sacrificing the satisfaction condition.

Formally, a sort generation constraint over a signature $\Sigma$ is a triple $(\dot{S}, \dot{F}, \theta)$, where $\theta : \overline{\Sigma} \to \Sigma$ for some $\overline{\Sigma} = (\overline{S}, \overline{TF}, \overline{PF}, \overline{P})$, $\dot{S} \subseteq \overline{S}$ and $\dot{F} \subseteq \overline{TF} \cup \overline{PF}$.

A *many-sorted $\Sigma$-sentence* is a closed many-sorted first order formula over $\Sigma$ or a sort generation constraint over $\Sigma$.

Given a signature morphism $\sigma : \Sigma \to \Sigma'$ and variable system X over $\Sigma$, we can obtain the variable system $\sigma(X)$ over $\Sigma'$ by

$$\sigma(X)_{s'} := \bigcup_{\sigma^S(s)=s'} X_s \ .$$

For an individual variable $x : s$ this translation yields $\sigma(x : s) = x : \sigma(s)$.

We extend this translation to terms by

- $\sigma(f_{w,s}(t_1, \ldots, t_n)) = \sigma^F_{w,s}(f_{w,s})_{\sigma^{S*}(w),\sigma^S(s)}(\sigma(t_1), \ldots, \sigma(t_n))$

and to formulae by

- $\sigma(p_w(t_1, \ldots t_n)) = \sigma^P_w(p_w)_{\sigma^{S*}(w)}(\sigma(t_1), \ldots \sigma(t_n))$,
- $\sigma(t_1 \stackrel{e}{=} t_2) = \sigma(t_1) \stackrel{e}{=} \sigma(t_2)$,
- $\sigma(t_1 = t_2) = \sigma(t_1) = \sigma(t_2)$,
- $\sigma(def\, t) = def\, \sigma(t)$,
- $\sigma(F) = F$,
- $\sigma(\varphi \wedge \psi) = \sigma(\varphi) \wedge \sigma(\psi)$,
- $\sigma(\varphi \Rightarrow \psi) = \sigma(\varphi) \Rightarrow \sigma(\psi)$,
- $\sigma(\forall x : s \bullet \varphi) = \forall x : \sigma^S(s) \bullet \sigma(\varphi)$.

The translation of a $\Sigma$-constraint $(\dot{S}, \dot{F}, \theta)$ along $\sigma$ is the $\Sigma'$-constraint $(\dot{S}, \dot{F}, \sigma \circ \theta)$. It is easy to see that the sentence translation along the identity signature morphism is the identity, and that the sentence translation along a composition of two signature morphisms is the composition of the sentence translations along the individual signature morphisms. Hence, sentence translation is functorial. This gives us the functor **sen**.

### 4.2.4 Satisfaction Relation

In order to define satisfaction of sentences we first must define term evaluation. Variable valuations are total, but the value of a term w.r.t. a variable valuation may be undefined, due to the application of a partial function during the evaluation of the term. Given a *total variable valuation* (i.e., a function assigning values to variables) $\nu : X \to M$, 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 $s \in S$.
- $\nu^\sharp_s(f_{w,s}(t_1, \ldots, t_n)) :=$
$$\begin{cases} (f_{w,s})_M(\nu^\sharp_{s_1}(t_1), \ldots, \nu^\sharp_{s_n}(t_n)) & \text{if } \nu^\sharp_{s_i}(t_i) \text{ is defined } (i = 1 \ldots n) \text{ and} \\ & \quad (f_{w,s})_M(\nu^\sharp_{s_1}(t_1), \ldots, \nu^\sharp_{s_n}(t_n)) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$
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 $w = \langle s_1, \ldots, s_n \rangle$.

The satisfiability 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 $\varphi$ :

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

$M \models \varphi$ holds for a many-sorted $\Sigma$-model $M$ and a many-sorted formula $\varphi$, iff $\nu \models \varphi$ for all variable valuations $\nu$ into $M$.

A $\Sigma$-constraint $(\dot{S}, \dot{F}, \theta)$ is satisfied in a $\Sigma$-model $M$, if the carriers of $M|_\theta$ of the sorts in $\dot{S}$ are generated by the function symbols in $\dot{F}$, that is, for every sort $s \in \dot{S}$ and every value $a \in (M|_\theta)_s$, there is a $\overline{\Sigma}$-term $t$ containing only function symbols from $\dot{F}$ and variables of sorts not in $\dot{S}$ such that $\nu^\sharp(t) = a$ for some assignment $\nu$ into $M|_\theta$.

For a sort generation constraint $(\dot{S}, \dot{F}, \theta)$ we can assume, without loss of generality, that all the result sorts of function symbols in $\dot{F}$ occur in $\dot{S}$. If not, we can just leave out from $\dot{F}$ those function symbols not satisfying this requirement. The satisfaction of the sort generation constraint in any model will not be affected by this: In the $\overline{\Sigma}$-term $t$ witnessing the satisfaction of the constraint, any application of a function symbol with result sort outside $\dot{S}$ can just be replaced by a variable of that sort, which is assigned the value resulting from the evaluation of the function application.

This concludes the presentation of *PCFOL*$^=$. For further details including the proof of the satisfaction condition for *PCFOL*$^=$ we refer the reader to [Mos02]. Next, we look at an extension which has sub-sorting.

## 4.3 The Institution *SubPCFOL*$^=$

We now introduce the institution *SubPCFOL*$^=$ (sub-sorted partial first order logic with sort generation constraints and equality), which extends *PCFOL*$^=$ with sub-sorting. It is this institution which underlies CASL.

### 4.3.1 Signatures

A *sub-sorted signature* $\Sigma = (S, TF, PF, P, \leq_S)$ consists of a many-sorted signature $(S, TF, PF, P)$ together with a reflexive and transitive *sub-sort relation* $\leq_S \subseteq S \times S$. The relation $\leq_S$ extends pointwise to sequences of sorts. We drop the subscript $S$ when it is obvious from the context. For a sub-sorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ 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 $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ iff there exist $w \in S^*, s \in S$ such that $w \leq_S w_1, w \leq_S w_2, s_1 \leq_S s$, and $s_2 \leq_S s$. Let $p : w_1, p : w_2 \in P$. Then $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ such that $w \leq_S w_1$ and $w \leq_S w_2$.

A *sub-sorted signature morphism* $\sigma : \Sigma \to \Sigma'$ is a many-sorted signature morphism that preserves the sub-sort relation and the overloading relations[2], that is, the following hold:

**p1** $s_1 \leq_S s_2$ implies $\sigma^S(s_1) \leq_{S'} \sigma^S(s_2)$ for all $s_1, s_2 \in S$ (preservation of the sub-sort relation),

**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 $f \in TF \cup PF$ (preservation of the overloading relation for functions), and

**p3** $p : w_1 \sim_P p : w_2$ implies $\sigma^P_{w_1}(p) = \sigma^P_{w_2}(p)$ for all $p \in P$ (preservation of the overloading relation for predicates).

With each sub-sorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ 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 $\mathtt{inj}_{s,s'} : s \to s'$ for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\mathtt{pr}_{s',s} : s' \to ?s$ for each pair of sorts $s \leq_S s'$, and
- an unary *membership* predicate symbol $\in^s_{s'}$: $s'$ for each pair of sorts $s \leq_S s'$.

We assume that the symbols used for injection, projection and membership are not used otherwise in $\Sigma$. We write $t \in s$ instead of $\in^s_{s'}(t)$ if $s'$ is clear from the context. We also drop the subscripts on the injection and projection functions when they are clear from the context.

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

### 4.3.2 Models

*Sub-sorted* $\Sigma$*-models* are many-sorted *PCFOL*$^=$ $\hat{\Sigma}$-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]Note that, thanks to preservation of sub-sorting, the preservation of the overloading relations can be simplified.

2. $\mathtt{inj}_{s,s'}(x) \stackrel{e}{=} \mathtt{inj}_{s,s'}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$.

3. $\mathtt{inj}_{s',s''}(\mathtt{inj}_{s,s'}(x)) \stackrel{e}{=} \mathtt{inj}_{s,s''}(x)$ for $s \leq_S s' \leq_S s''$.

4. $\mathtt{pr}_{s',s}(\mathtt{inj}_{s,s'}(x)) \stackrel{e}{=} x$ for $s \leq_S s'$.

5. $\mathtt{pr}_{s',s}(x) \stackrel{e}{=} \mathtt{pr}_{s',s}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$.

6. $\in^s_{s'}(x) \Leftrightarrow \mathit{def}\, \mathtt{pr}_{s',s}(x)$ for $s \leq_S s'$.

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 $w \leq_S w', w''$, $w = \langle s_1,\ldots,s_n\rangle$, $w' = \langle s'_1,\ldots,s'_n\rangle$, $w'' = \langle s''_1,\ldots,s''_n\rangle$, and $s', s'' \leq_S s$.

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 $w \leq_S w', w''$, $w = \langle s_1,\ldots,s_n\rangle$, $w' = \langle s'_1,\ldots,s'_n\rangle$, and $w'' = \langle s''_1,\ldots,s''_n\rangle$.

*Sub-sorted $\Sigma$-model morphisms* are many-sorted $\hat{\Sigma}$-model morphisms.

### 4.3.3 Sentences

The *Sub-sorted sentences over* $\Sigma$ are the many-sorted sentences over $\hat{\Sigma}$. Sentence translation along a sub-sorted signature morphism $\sigma$ is just sentence translation along the many-sorted signature morphism $\hat{\sigma}$.

### 4.3.4 Satisfaction

The satisfaction relation is inherited from *PCFOL*$^=$, as models and sentences are taken from there too. Therefore, the satisfaction condition is also inherited from *PCFOL*$^=$. For further details see [Mos02].

This concludes our presentation of *SubPCFOL*$^=$ which is the underlying institution of CASL. In order to use this institution for CSP-CASL we need to make some restrictions to the signature category. We discuss this next.

## 4.4 The Restricted *SubPCFOL*$^=$ Institution

To be able to define a well behaved equivalence relation on our alphabet construction, in Section 7.2.1, we need to restrict the signatures of *SubPCFOL*$^=$. Without this restriction we would later fail to prove transitivity of said relation. To this end, we form a sub-institution of *SubPCFOL*$^=$, which we call *ResSubPCFOL*$^=$ (restricted sub-sorted partial first order logic with sort generation constraints and equality). *ResSubPCFOL*$^=$ is the same as the *SubPCFOL*$^=$ institution but with a restricted signature category.

**Signatures** A *restricted sub-sorted signature* $\Sigma = (S, TF, PF, P, \leq)$ is a *SubPCFOL*$^{=}$ signature which satisfies the following additional properties [Rog06]:

- the set of sorts is finite, and
- the sub-sort relation has *local top sorts*, that is, if $s \leq u, u'$ then there exists $t \in S$ with $u, u' \leq t$.

These restrictions are not as harsh as they may seem at first sight. The first restriction is hardly restrictive at all, as all specifications will be of finite length and there is no construction to introduce an infinite number of sort symbols. Roggenbach [Rog06] states that the second restriction holds in a large number of the standard CASL libraries. We verify this claim using the following specification to test the standard CASL libraries.

**from** BASIC/ALGEBRA_I **get** MONOID, COMMUTATIVEMONOID, ...
...
**logic** CSPCASL

**spec** TEST_MONOID =
  **data** MONOID
  **process** $P$ : {};
**end**

**spec** TEST_COMMUTATIVEMONOID =
  **data** COMMUTATIVEMONOID
  **process** $P$ : {};
**end**
...

Here, we import each CASL specification from the standard libraries in turn and use them as the data part for a CSP-CASL specification. The tool HETS requires all CSP-CASL specifications to contain at least one process name, thus we specify a process name $P$ which has the empty communication set. The tool HETS throws errors when the data part of a CSP-CASL specification lacks local top sorts. The above text was successfully parsed and statically analysed. As no errors were thrown, we can conclude that all specifications in the standard CASL libraries have local top sorts. As these CASL libraries capture almost all common data types, we strongly believe that all reasonable specifications will have local top sorts.

A *restricted sub-sorted signature morphism* $\sigma : (S, TF, PF, P, \leq) \to (S', TF', PF', P', \leq')$ is a *SubPCFOL*$^{=}$ signature morphism which satisfies the additional properties:

**refl** $\sigma(s_1) \leq' \sigma(s_2)$ implies $s_1 \leq s_2$ for all $s_1, s_2 \in S$ (reflection of the sub-sort relation), and

**weak non-extension** $\sigma(s_1) \leq' u' \wedge \sigma(s_2) \leq' u'$ implies that there exists a sort $t \in S$ with $s_1 \leq t$, $s_2 \leq t$ and $\sigma(t) \leq' u'$ for all $s_1, s_2 \in S$ and $u' \in S'$.[3]

[3]Further note that for $s_1 = s_2$ , the condition trivially holds.

Composition of restricted sub-sorted signature morphisms and the definition of identity morphisms are inherited from *SubPCFOL*$^{=}$.

The construction in [Rog06] works with the condition *non-extension* in place of *weak non-extension*:

**non-extension** $\sigma(s_1) \leq' u' \wedge \sigma(s_2) \leq' u'$ implies that there exists a sort $u \in S$ with $\sigma(u) = u'$ for all $s_1, s_2 \in S$ and $u' \in S'$.

One can show however that the results of [Rog06] can also be achieved with the more liberal notion (*weak non-extension*) that we use here. The difference from the original version (*non-extension*) is the more liberal choice of sort $t$ (originally, we have required $\sigma(t) = u'$).

**Lemma 4.1** The properties *non-extension* and *refl* imply *weak non-extension*.

*Proof.* Assume that the property *non-extension* holds. Let $s_1, s_2 \in S$ and $u' \in S'$ be sorts, such that $\sigma(s_1) \leq' u'$, and $\sigma(s_2) \leq' u'$. We must show that there exists a sort $t \in S$ such that $s_1 \leq t$, $s_2 \leq t$, and $\sigma(t) \leq' u'$. By *non-extension* we know there exists a sort $t \in S$ such that $\sigma(t) = u'$. As we know $\sigma(s_1) \leq' u' = \sigma(t)$, by the property *refl* we know $s_1 \leq t$. Similarly we can conclude $s_2 \leq t$. Finally, as the sub-sort relation is reflexive we know $\sigma(t) \leq' u' = \sigma(t)$. □

As an example of a signature morphism that is allowed by *weak-non-extension*, but disallowed by the original *non-extension* property, consider the following. Let $\Sigma$ be a specification with sorts $s_1$, $s_2$, and $t$ such that $s_1$ and $s_2$ are sub-sorts of $t$. Let $\Sigma'$ be a specification with sorts $s_1'$, $s_2'$, $t'$ and $u'$ such that $s_1'$ and $s_2'$ are sub-sorts of $t'$ and $t'$ is a sub-sort of $u'$. Finally, let $\sigma : \Sigma \to \Sigma'$ be the signature morphism that maps: $s_1$ to $s_1'$, $s_2$ to $s_2'$, and $t$ to $t'$. Clearly $\sigma$ satisfies the property *weak-non-extension* but not *non-extension*. This is a simple signature morphism which we would not like to disallow.

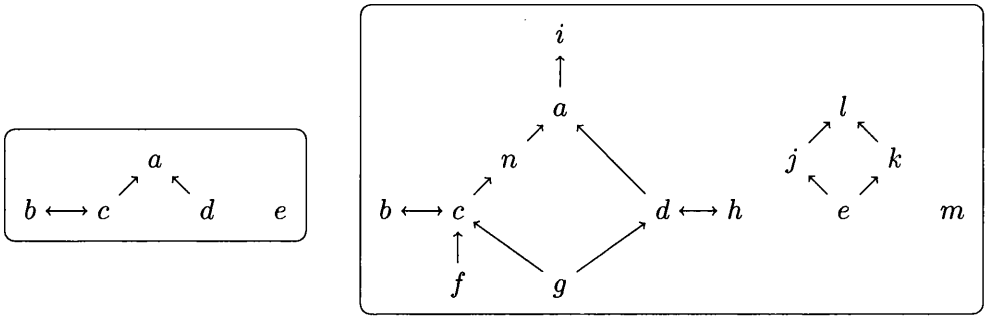
We now illustrate how new sorts can be introduced via restricted sub-sorted signature morphisms.

**Example 4.2** The following are two examples of valid *ResSubPCFOL*$^{=}$ specifications.

**spec** EXAMPLESPEC1 =
  **sorts** $c, d < a$;
    $b = c$;
    $e$
**end**

**spec** EXAMPLESPEC2 =
  **sorts** $a < i$;
    $n, d < a$;
    $c < n$;
    $b = c$; $d = h$;
    $f, g < c$; $g < d$;
    $j, k < l$;
    $e < j$; $e < k$;
    $m$
**end**

Let $\Sigma$ and $\Sigma'$ be the signatures of EXAMPLESPEC1 and EXAMPLESPEC2 respectively. Below are diagrams of the underlying sub-sort relations.




There is a signature morphism between the signatures $\Sigma$ and $\Sigma'$ where the morphism preserves the identity of the sorts. This signature morphism shows it is possible to:

- introduce new sorts on top of existing sub-sort relations (e.g., sort $i$ on top of the join (sort $a$) between sorts $c$ and $d$),
- introduce new sorts that are isomorphic to existing sorts (e.g., sort $h$),
- introduce new sorts that are sub-sorts of existing sorts (e.g., sort $f$),
- introduce new sorts that are inline with the existing sub-sort relation (e.g., sort $n$),
- join multiple existing sorts using new sub-sorts (e.g., sort $g$),
- introduce new super-sorts in different connected components (e.g., sort $j$), and
- introduce new independent sorts (e.g., sort $m$).

This shows that there are many situations in which new sort symbols can be introduced using signature morphisms. Therefore, we feel the restrictions are not too harsh and still allow for liberal use of signature morphisms.

The following is a more concrete example of how these signature morphisms are useful in practice.

**Example 4.3** Consider an abstract specification of an online shopping system. There are several distributed components: a customer, a warehouse, a payment system, etc. These entities communicate with one another with the goal of allowing the customer to purchase some goods. Here, we only consider the customer.

At the most abstract level, we may only wish to specify that the customer has a set of messages which it is able to send and receive. We may also wish to specify that there exist login and logout messages in the set of messages. This can be captured by the following specification (which has a restricted sub-sort signature).

**spec** ABSTRACT_CUSTOMER =
  **sorts** *LoginReq*, *LoginRes*, *LogoutReq*, *LogoutRes* < *Customer_Data*;
**end**

The customer has the following message types available for use: Login request messages, login response messages, logout request messages, and logout response messages. The idea is that the customer can send a request message and then wait for a response message (although we do not specify this behaviour here). Any other messages are abstract and are covered by the super sort $Customer\_Data$.

Later in the specification process, probably at a more concrete level of abstraction, the specifier may wish to define what further messages could additionally be communicated. The following concrete customer specification allows for this extension.

**spec** CONCRETE_CUSTOMER =
  **sorts** *LoginReq*, *LoginRes*, *LogoutReq*, *LogoutRes*,
    *ViewBasketReq*, *ViewBasketRes*,
    *AddItemReq*, *AddItemRes*, *RemoveItemReq*, *RemoveItemRes*,
    *CheckoutReq*, *CheckoutRes*, *CancelReq*, *CancelRes* < *Customer_Data*;
**end**

Here, we have added extra message types (sub-sorts of $Customer\_Data$). These allow the customer to view their basket, add and remove items from their basket, check out their basket and cancel their entire basket.

There is the obvious signature embedding from the signature of the abstract customer to the signature of the concrete customer. This signature embedding is a valid restricted sub-sorted signature morphism and shows how our notion of signature morphism can be useful in practice.

**Lemma 4.4** The restricted sub-sorted signatures and signature morphisms form a sub-category of the *SubPCFOL*$^{=}$ signature category.

*Proof.* Identity morphisms and composition are inherited from the *SubPCFOL*$^{=}$ signature category. It is straightforward to check that the identity morphisms satisfy the *refl* and *weak non-extension* properties. We now prove that composition of signature morphisms also preserves these proprieties.

Let $\sigma : \Sigma \to \Sigma'$ and $\sigma' : \Sigma' \to \Sigma''$ be *ResSubPCFOL*$^{=}$ signature morphisms with $S$, $S'$, $S''$ and $\leq$, $\leq'$ and $\leq''$ being the sets of sort symbols and sub-sort relations of $\Sigma$, $\Sigma'$ and $\Sigma''$, respectively.

**refl** Let $s_1, s_2 \in S$ such that $(\sigma' \circ \sigma)(s_1) \leq'' (\sigma' \circ \sigma)(s_2)$. We must prove $s_1 \leq s_2$. By *refl* of $\sigma'$, we know $\sigma(s_1) \leq' \sigma(s_2)$ and by *refl* of $\sigma$, we can conclude $s_1 \leq s_2$.

**weak non-extension** Let $s_1, s_2 \in S$ and $u'' \in S''$ such that $(\sigma' \circ \sigma)(s_1) \leq'' u''$, and $(\sigma' \circ \sigma)(s_2) \leq'' u''$. We must find a sort $t \in S$ such that $s_1 \leq t$, $s_2 \leq t$, and $(\sigma' \circ \sigma)(t) \leq u''$. By *weak non-extension* of $\sigma'$, we know there exists a sort $t' \in S'$ such that $\sigma(s_1) \leq' t'$, $\sigma(s_2) \leq' t'$, and $\sigma'(t') \leq'' u''$. By *weak non-extension* of $\sigma$, we know there exists a sort $t \in S$ such that $s_1 \leq t$, $s_2 \leq t$, and $\sigma(t) \leq' t'$. By the property **p1** of $\sigma$ (see Section 4.3.1) we also know that $\sigma'(\sigma(t)) \leq'' \sigma'(t')$. Thus, by transitivity of $\leq''$ we obtain $\sigma'(\sigma(t)) \leq'' u''$. □

The models, sentences and satisfaction relation of *ResSubPCFOL*$^{=}$ are the same as in the institution *SubPCFOL*$^{=}$.

**Theorem 4.5** *ResSubPCFOL*$^{=}$ is an institution.

*Proof.* As we have only restricted the signature category, the satisfaction condition is inherited. Therefore, *ResSubPCFOL*$^{=}$ forms an institution and is a sub-institution of *SubPCFOL*$^{=}$ (the underlying institution of CASL). □

## 4.5 Data-Logic

The *data-logic* (formulated in [Rog06]) is the logic used within the process part of CSP-CASL. The purpose of this institution is to provide a way of dealing with partiality in the models as well as allowing within sentences the ability to (1) test the equality of terms of different sorts, and (2) test whether a term is of a certain sort. Both of these 'features' are needed to be able to use CASL terms as communications and formulae as conditions in CSP processes. This data-logic can be thought of as a specialisation of *ResSubPCFOL*$^{=}$.

We briefly sketch this institution, for full details see [Rog06]. The signature category is the same as for *ResSubPCFOL*$^{=}$. However, sentences, models and satisfaction are different.

**Models** A *data-logic* $\Sigma$*-model* $M_\perp$ is the strict extension of a *ResSubPCFOL*$^{=}$ $\Sigma$-model $M$ [Rog06]. Let $\Sigma = (S, TF, PF, P, \leq)$ be a *ResSubPCFOL*$^{=}$ signature with the associated signature $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$, as defined in Section 4.3.1. Given a *ResSubPCFOL*$^{=}$ $\Sigma$-model $M$, its strict extension is defined by[4]:

- $(M_\perp)_s := M_s \cup \{\perp\}$ for all $s \in \hat{S}$ (we assume $\perp \notin M_s$),
- $(f_{w,s})_{M_\perp}(x_1, \ldots, x_n) := \begin{cases} (f_{w,s})_M(x_1, \ldots, x_n) & \text{if } x_i \in M_{s_i} (i = 1 \ldots n) \text{ and } (f_{w,s})_M(x_1, \ldots, x_n) \text{ is defined} \\ \perp & \text{otherwise} \end{cases}$

  for all $f$ in $\hat{TF}_{w,s} \cup \hat{PF}_{w,s}$, and
- $(p_w)_{M_\perp} := (p_w)_M$ for all $p \in \hat{P}_w$.

This construction leads to a one-one correspondence between ordinary many-sorted models in *ResSubPCFOL*$^{=}$ and their totalised counter parts in the data-logic. Given a *ResSubPCFOL*$^{=}$ model $M$, its extension $M_\perp$ is uniquely determined. Forgetting the strict extension results again in $M$.

*Data-logic* $\Sigma$*-morphisms* are many-sorted *ResSubPCFOL*$^{=}$ $\Sigma$-morphisms which are extended to $\perp$. Given a many-sorted morphism $h : M \to M'$ between two many-sorted *ResSubPCFOL*$^{=}$ $\Sigma$-models $M$ and $M'$, we can strictly extend $h$ to a totalised morphism $h_\perp : M_\perp \to M'_\perp$ by

$$(h_\perp)_s(x) := \begin{cases} h(x) & \text{if } x \in M_s \\ \perp & \text{if } x = \perp \end{cases}$$

[4] In the model definition we use the same $\perp$ symbol for all carrier sets. Later, the construction of the alphabet of communications (see Chapter 7) will differentiate some of them based upon the sub-sort relation.

As this extension is again uniquely determined, there is also a one-one correspondence between the many-sorted *ResSubPCFOL*$^{=}$ $\Sigma$-model morphisms and data-logic $\Sigma$-model morphisms. Given a *ResSubPCFOL*$^{=}$ model morphism $h : M \to M'$, its extension $h_\perp : M_\perp \to M'_\perp$ is uniquely determined. Forgetting the strict extension results again in $h : M \to M'$.

**Sentences** Sentences are similar to those in *ResSubPCFOL*$^{=}$, however we allow equality tests over different sorts and a membership test. To this end, Let $\Sigma = (S, TF, PF, P, \leq_S)$ be a signature and $X$ be a variable system over $\Sigma$. We inherit the set of $\Sigma$-terms $T_\Sigma(X)$ from *ResSubPCFOL*$^{=}$ and define the set $AF_\Sigma(X)$ of atomic $\Sigma$-formulae with variables in $X$ as the least set satisfying the following rules:

1. $p_w(t_1, \ldots t_n) \in AF_\Sigma(X)$, if $t_i \in T_\Sigma(X)_{s_i}$, $p_w \in P_w$, $w = \langle s_1, \ldots, s_n \rangle \in S^*$ (predicates),
2. $t_1 \stackrel{e}{=} t_2 \in AF_\Sigma(X)$, if $t_1, t_2 \in T_\Sigma(X)$ (existential equations),
3. $t_1 = t_2 \in AF_\Sigma(X)$, if $t_1, t_2 \in T_\Sigma(X)$ (strong equations),
4. $def\, t \in AF_\Sigma(X)$, if $t \in T_\Sigma(X)$ (definedness assertions),
5. $t\ in\ s' \in AF_\Sigma(X)$, if $t \in T_\Sigma(X)_s$, $s, s' \in S$ (element relation).

These are the same rules as for *PCFOL*$^{=}$, however, we have dropped the requirements on Points 2 and 3 that the terms $t_1$ and $t_2$ must be from the same sort. Furthermore, we have added a test for membership which allows a term of any sort to be tested for membership in any other sort. We then construct formulae and sentences as is done in *ResSubPCFOL*$^{=}$.

The translation of sentences across a signature morphism is the same as the translation of *ResSubPCFOL*$^{=}$ but extended with the rules:

- $\sigma(t_1 \stackrel{e}{=} t_2) := \sigma(t_1) \stackrel{e}{=} \sigma(t_2)$,
- $\sigma(t_1 = t_2) := \sigma(t_1) = \sigma(t_2)$, and
- $\sigma(t\ in\ s') := \sigma(t)\ in\ \sigma^S(s')$.

which translate our additional atomic formulae.

**Satisfaction** We inherit the evaluation of terms from *ResSubPCFOL*$^{=}$, however we totalise it, that is, the evaluation returns $\perp$ whenever the *ResSubPCFOL*$^{=}$ evaluation of a term yields undefined. Let $\Sigma = (S, TF, PF, P, \leq)$ be a signature. We define the evaluation of a formula $\varphi$, relative to a valuation $\nu : X \to M_\perp$, inductively over the structure of $\varphi$ as:

- $\nu \models p_w(t_1, \ldots, t_n)$ iff $(\nu^\sharp(t_1), \ldots, \nu^\sharp(t_n)) \in (p_w)_{M_\perp}$.
- $\nu \models t_1 \stackrel{e}{=} t_2$ iff
  - $\nu^\sharp_{s_1}(t_1) \neq \perp$, $\nu^\sharp_{s_2}(t_2) \neq \perp$,
  - there exists $u \in S$ such that $s_1 \leq u$ and $s_2 \leq u$, and

- for all $u \in S$ such that $s_1 \leq u$ and $s_2 \leq u$ holds:

$$\nu_u^\sharp(\mathtt{inj}_{s_1,u}(t_1)) = \nu_u^\sharp(\mathtt{inj}_{s_2,u}(t_2)).$$

- $\nu \models t_1 = t_2$ iff
either
  - $\nu_{s_1}^\sharp(t_1) = \bot$, $\nu_{s_2}^\sharp(t_2) = \bot$, and
  - there exists $u \in S$ such that $s_1 \leq u$ and $s_2 \leq u$,

  or
  - $\nu_{s_1}^\sharp(t_1) \neq \bot$, $\nu_{s_2}^\sharp(t_2) \neq \bot$,
  - there exists $u \in S$ such that $s_1 \leq u$ and $s_2 \leq u$, and
  - for all $u \in S$ such that $s_1 \leq u$ and $s_2 \leq u$ holds:

$$\nu_u^\sharp(\mathtt{inj}_{s_1,u}(t_1)) = \nu_u^\sharp(\mathtt{inj}_{s_2,u}(t_2)).$$

- $\nu \models \mathit{def}\, t$ iff $\nu^\sharp(t) \neq \bot$.

- $\nu \models t \mathit{\ in\ } s'$ iff there exists $a \in M_{\bot s'}$ such that
either
  - $\nu_s^\sharp(t) = a = \bot$, and
  - there exists $u \in S$ such that $s \leq u$ and $s' \leq u$,

  or
  - $\nu_s^\sharp(t) \neq \bot$, $a \neq \bot$,
  - there exists $u \in S$ such that $s \leq u$ and $s' \leq u$, and
  - for all $u \in S$ such that $s \leq u$ and $s' \leq u$ holds:

$$\nu_u^\sharp(\mathtt{inj}_{s,u}(t)) = (\mathtt{inj}_{s',u})_M(a).$$

- not $\nu \models F$.

- $\nu \models \varphi \wedge \psi$ iff $\nu \models \varphi$ and $\nu \models \psi$.

- $\nu \models \varphi \Rightarrow \psi$ iff $\nu \models \varphi$ implies $\nu \models \psi$.

- $\nu \models \forall x : s \bullet \varphi$ iff for all extended valuations $\zeta : X \cup \{x : s\} \to M_\bot$ (i.e., valuations where it holds that $\zeta(y) = \nu(y)$ for all $y \in X \setminus \{x : s\}$) such that $\zeta(x : s) \neq \bot$ we have $\zeta \models \varphi$.

$M_\perp \models \varphi$ holds for a $\Sigma$-model and a formula $\varphi$, iff $\nu \models \varphi$ for all variable valuations $\nu$ into $M_\perp$.

The proof that the data-logic forms an institution can be found in [Rog06]. An institution representation – see [Rog06] for details – allows us to map *ResSubPCFOL*$^=$ models to data-logic models.

This data-logic will form the basis for the description of data in CSP-CASL. We have extended the syntax of *ResSubPCFOL*$^=$ to include tests for equality between terms of different sorts and a sort membership predicate. These 'features' are necessary to use CASL terms as communications and formulae in CSP processes.

## 4.6 CSP Institutions

We now show that institutions are not only applicable to the area of algebraic specification, but also apply to other settings, for example, process algebra. CSP can also be formalised as various institutions. Here, we sketch the construction as it is presented in [MR07].

### 4.6.1 What is an Appropriate Notion of a Signature Morphism?

Before we define CSP signatures we illustrate with an example from [MR07] why we require CSP signature morphisms to be injective.

When analysing CSP specifications, it becomes clear that there are two types of symbols that change from specification to specification, namely, communications and process names. Pairs consisting of an alphabet $A$ of communication symbols and of process names $\dot{N}$ (together with some type information) will form CSP signatures, see Section 4.6.2. The notion of a signature morphism, however, is not as easy to determine. An institution captures how truth can be preserved under change of symbols. In this sense, we want to come up with a notion of a signature morphism that is as liberal as possible but still respects fundamental CSP properties. In this section, we discuss why this requires us to restrict alphabet translations to injective functions.

The process algebra CSP itself offers an operator that changes the communications of a process $P$, namely, *functional renaming* $f[P]$.[5] Here, $f : A \rightarrow ? A$ is a (partial) function such that $dom(f)$ includes all communications occurring in $P$ (see Chapter 2). The CSP literature (e.g., [Ros98]), classifies functional renaming as follows:

1. Functional renaming with an injective function $f$ preserves all process properties.

2. Functional renaming with a non-injective function $f$ is mainly used for process abstraction. Non-injective renaming can introduce unbounded non-determinism and thus change fundamental process properties.

[5] Note that the so-called relational renaming, which is included in our CSP dialect, subsumes functional renaming.

As a process algebra, CSP exhibits a number of fundamental algebraic laws (see Section 2.3 for a selection). One such law is the step law:

$$\begin{aligned} & (\Box x :: X \to P) \,\Box\, (\Box y :: Y \to Q) \\ = \;& \Box x :: X \cup Y \to \text{if } x \in X \cap Y \text{ then } (P \sqcap Q) \text{ else } (\text{if } x \in X \text{ then } P \text{ else } Q) \end{aligned}$$

The step laws hold in all the main CSP semantics and are essential for the definition of complete axiomatic semantics for CSP, see [Ros98, IR06]. The CSP step laws show that, for example, the behaviour of the external choice $\Box$, generalised parallel $[\![ X ]\!]$ and hiding $\setminus$ operators crucially depend on the equality relation in the alphabet of communications. We demonstrate this here for the external choice operator $\Box$ :

- Assume $a \neq b$. Then

$$\begin{aligned} & (\Box x :: \{a\} \to P) \,\Box\, (\Box y :: \{b\} \to Q) \\ = \;& \Box x :: \{a, b\} \to \text{if } x \in \{a\} \cap \{b\} \text{ then } (P \sqcap Q) \text{ else } (\text{if } x \in \{a\} \text{ then } P \text{ else } Q) \\ = \;& \Box x :: \{a, b\} \to \text{if } x \in \{a\} \text{ then } P \text{ else } Q \end{aligned}$$

- Mapping $a$ and $b$ with a non-injective function $f$ to the same element $c$ has the effect:

$$\begin{aligned} & f[(\Box x :: \{a\} \to P) \,\Box\, (\Box y :: \{b\} \to Q)] \\ = \;& ((\Box x :: \{c\} \to f[P]) \,\Box\, (\Box y :: \{c\} \to f[Q])) \\ = \;& \Box x :: \{c\} \to \text{if } x \in \{c\} \cap \{c\} \text{ then } (f[P] \sqcap f[Q]) \text{ else} \\ & \quad (\text{if } x \in \{c\} \text{ then } f[P] \text{ else } f[Q]) \\ = \;& \Box x :: \{c\} \to (f[P] \sqcap f[Q]) \end{aligned}$$

Here, we can see that the use of a non-injective renaming has changed the semantics of the process. Before the translation, the environment controlled which one of the two processes $P$ and $Q$ is executed – after the translation this control has been lost: the process makes an internal choice between $f[P]$ and $f[Q]$. Similar examples can be extracted from the step laws for the alphabetised parallel $[\![ X ]\!]$ and hiding $\setminus$ operators.

This example demonstrates that the use of non-injective renamings can change the semantics of a process. As institutions are concerned with preservation of truth, and in particular the satisfaction condition states that truth is preserved under change of notation, we are forced to conclude that such non-injective renamings are not compatible with institutions. Therefore, we require alphabet translations, which will be a constituent of CSP signature morphisms, to be injective functions.

### 4.6.2 CSP Signatures

A CSP signature is a pair $(A, N)$ where

- $A$ is an alphabet of communications, and
- $N = (\bar{N}, comms, param)$ collects information on process names; $\bar{N}$ is a set of process names, where each process name $n \in \bar{N}$ has

- a parameter type $param(n) = \langle X_1, \ldots, X_k \rangle$, $X_i \subseteq A$ for $1 \leq i \leq k$, $k \geq 0$. A process name without parameters has the empty sequence $\langle\rangle$ as its parameter type.
- a type $comms(n) \subseteq A$, which collects all possible communications in which the process name $n$ can engage in.

By abuse of notation, we will write $n \in N$ instead of $n \in \bar{N}$ and $(a_1, \ldots, a_k) \in param(n)$ instead of $(a_1, \ldots, a_k) \in X_1 \times \cdots \times X_k$, where $param(n) = \langle X_1, \ldots, X_k \rangle$.

A CSP *signature morphism* $\sigma = (\alpha, \nu) : (A, N) \to (A', N')$ consists of two maps:

- $\alpha : A \to A'$, an injective alphabet translation, and
- $\nu : N \to N'$, a translation of process names, which has the following two properties:
  - $param'(\nu(n)) = \alpha^*(param(n))$ : preservation of parameter types, where $\alpha^*$ denotes the extension of $\alpha$ to sequences of sets.
  - $comms'(\nu(n)) \subseteq (comms(n))$ : non-expansion of types, that is, the translated process $\nu(n)$ is restricted to those events which are obtained by translation of its type $comms(n)$.

The non-expansion of types principle is crucial for ensuring the satisfaction condition of the CSP institutions. It ensures that the semantics of a process is frozen when translated to a larger context, that is, even when moving to a larger alphabet, up to renaming, models for "old" names may only use "old" alphabet letters. This corresponds to a *black-box* view on processes that are imported from other specification modules.

### 4.6.3 CSP Sentences

Sentences in CSP are formed relative to a signature, a global variable system, a local variable systems, and a logic. Global variables are used as parameters for process names while local variables are introduced by bindings in some of the CSP operators (see Section 2.1). The logic is left open and only assumed to exhibit certain properties, for instance, that it comes with a substitution operator that works in the usual way. The logic defines the syntax of formulae that are used in the CSP conditional operator. Here, we sketch the construction of sentences, for full details see [MR07].

The set of *process terms* $T_{(A,N)}(G, L)$ is defined over a signature $(A, N)$ and relative to global and local variable systems $G$ and $L$, respectively. These process terms are similar to those in Section 2.1 where the variable systems control the availability and binding of variables.

A *process definition* over a signature $(A, N)$ is an equation

$$p(x_1, \ldots, x_k) = pt$$

where $p$ is a process name in $N$, the $x_i$ are variables with $x_i : X_i$, where $X_i$ is the $i\text{-}th$ component of $param(p)$, and $pt$ is a CSP process term. Finally, a process definition is a *sentence* if $pt \in T_{(comms(p),N)}(\{x_1 : X_1, \ldots, x_k : X_k\}, \emptyset)$, that is, $pt$ is a process term of the correct type which contains only appropriate global variables.

### 4.6.4 CSP Models and Reducts

Let $\mathcal{D}(A)$ be a CSP domain constructed relatively to a set of communications $A$, that is, an alphabet (see Chapter 2). A *model* $M$ over a signature $(A, N)$ assigns to each process name $n$ and for all $(a_1, \ldots, a_k) \in \mathit{param}(n)$ a type correct element of the semantic domain $\mathcal{D}(A)$, that is,

$$M(n(a_1, \ldots, a_k)) \in \mathcal{D}(\mathit{comms}(n)) \subseteq \mathcal{D}(A) \ .$$

We define model categories to be partial orders (i.e., CSP refinements), that is, there is a morphism between models $M_1$ and $M_2$, iff $M_1 \sqsubseteq_{\mathcal{D}} M_2$. Here $\sqsubseteq_{\mathcal{D}}$ is the pointwise extension of the partial order used in the denotational CSP semantics for the chosen domain $\mathcal{D}$.

Given an injective (total) alphabet translation $\alpha : A \rightarrow A'$ we define its partial inverse as

$$\begin{array}{rccl} \hat{\alpha}: & A' & \rightarrow? & A \\ & a' & \mapsto & \begin{cases} a & \text{if } a \in A \text{ is such that } \alpha(a) = a' \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

Let $\hat{\alpha}^{\mathcal{D}} : \mathcal{D}(A') \rightarrow? \mathcal{D}(A)$ be the extension of $\hat{\alpha}$ to semantic domains – to be defined for any domain individually.

The *reduct* of a model $M'$ along the signature morphism $\sigma$ is defined as

$$M'|_{\sigma}(n(a_1, \ldots, a_k)) = \hat{\alpha}^{\mathcal{D}}(M'(\nu(n)(\alpha(a_1), \ldots, \alpha(a_k)))) \ .$$

On the level of domains, we define the following *reduct condition* on $\alpha$ and $\hat{\alpha}^{\mathcal{D}}$ :

$$\forall X \subseteq A \bullet \ \hat{\alpha}_{\mathcal{D}}(\mathcal{D}(\alpha(X))) \subseteq \mathcal{D}(X) \ .$$

This condition controls the relationship between the underlying translation $\alpha$ and the domain translation $\hat{\alpha}^{\mathcal{D}}$. This condition is essential for proving the for the satisfaction condition to hold. The CSP-CASL institutions will impose a similar condition on model morphisms in Chapter 8.

### 4.6.5 Satisfaction

We now sketch the construction of the satisfaction relation as it is presented in [MR07].

Given a map $\mathit{denotation} : M \times T_{(A,N)}(\emptyset, \emptyset) \rightarrow \mathcal{D}(A)$, which – given a model $M$ – maps a closed process term $pt \in T_{(A,N)}(\emptyset, \emptyset)$ to its denotation in $\mathcal{D}(A)$, we define the satisfaction relation of our institution:

$$\begin{array}{c} M \models p(x_1, \ldots, x_k) = pt \\ \text{if and only if} \\ \forall (a_1, \ldots, a_k) \in \mathit{param}(p) \bullet \\ \mathit{denotation}_M(p(a_1, \ldots a_k)) = \mathit{denotation}_M(pt[a_1/x_1, \ldots, a_k/x_k]) \end{array}$$

The map *denotation* is similar to the CSP semantic functions $[\![\_]\!]_{\mathcal{D}}$ defined in Chapter 2 but takes into account the model $M$ for giving interpretations to the process names.

If the chosen CSP semantics has the reduct property and the extension of $\alpha$ and $\hat{\alpha}$ are inverse functions on $\mathcal{D}(A)$ and $\mathcal{D}(\alpha(A))$, then the satisfaction condition holds (see [MR07] for full details).

This concludes our sketch of the CSP institutions as presented in [MR07]. Each CSP semantics $\mathcal{D}$ can be catered for by plugging in the appropriate function for the *denotation* function above. This section has illustrated that institutions are not solely applicable to algebraic specification and also that CSP is well behaved, as it can be formalised as various institutions.

## 4.7 Institution Independent Structuring

We conclude this chapter by presenting a kernel structuring language which is institution independent, following Mossakowski [Mos02]. This structuring language provides a construction for basic specifications (also known as presentations) and several operations to build specifications in a composed manner. Such an approach has many advantages, for example, it allows code re-use and allows the specifier to focus on smaller contexts, see Section 1.2.1. We have already seen use of the structuring mechanisms throughout Chapter 3. However, we previously presented only informal ideas about their meanings. We now formally introduce such a structuring language.

For the remainder of this chapter let $I = (\mathbf{SIGN}, \mathbf{sen}, \mathbf{mod}, \models)$ be some fixed institution.

### 4.7.1 A Kernel Structuring Language

Structured specifications are formed from three structuring operators and basic presentations.

These operators and presentations are formulated in a model theoretic approach using the functions **Sig** for the signature of a specification and **Mod** for the models of a specification.[6] This means that only the specifications are structured and not the models.

**Presentations:** Presentations form the basis of specifications. A presentation consists of a signature coupled with a set of formulae (axioms).

For any signature $\Sigma \in |\mathbf{SIGN}|$ and finite set $\Gamma \subseteq \mathbf{sen}(\Sigma)$ of $\Sigma$-sentences, the presentation $\langle \Sigma, \Gamma \rangle$ is a specification with:

$$\begin{aligned} \mathbf{Sig}(\langle \Sigma, \Gamma \rangle) &:= \Sigma \\ \mathbf{Mod}(\langle \Sigma, \Gamma \rangle) &:= \{M \in |\mathbf{mod}(\Sigma)| \mid M \models \Gamma\} \end{aligned}$$

where $M \models \Gamma$ iff $M \models \varphi$ for all $\varphi \in \Gamma$.

**Union:** Specifications can be joined using the union operator, which takes two specifications with the same signature and restricts the models to only those models which satisfy both specifications.

For any two $\Sigma$-specifications $SP_1$ and $SP_2$, their union $SP_1$ **and** $SP_2$ is a specification with:

$$\begin{aligned} \mathbf{Sig}(SP_1 \textbf{ and } SP_2) &:= \Sigma \\ \mathbf{Mod}(SP_1 \textbf{ and } SP_2) &:= \mathbf{Mod}(SP_1) \cap \mathbf{Mod}(SP_2) \end{aligned}$$

Specifications with different signatures can be joined by first translating them such that they share the same signature.

[6] The function **Mod** for the models of a specification should not be confused with the functor **mod** from the underlying institution $I$ which produces the category of models for a signature.

**Renaming:** Specifications can be translated along signature morphisms. This allows, for instance, the expansion of a signature to include new symbols or even the collapsing of symbols (assuming such signature morphisms are included in signature category **SIGN**).

For any signature morphism $\sigma : \Sigma \to \Sigma'$ and $\Sigma$-specification $SP$, $SP$ **rename** $\sigma$ is a specification with:

$$\begin{array}{lcl} \mathbf{Sig}(SP \textbf{ rename } \sigma) & := & \Sigma' \\ \mathbf{Mod}(SP \textbf{ rename } \sigma) & := & \{M' \in |\mathbf{mod}(\Sigma')| \mid M'|_\sigma \in \mathbf{Mod}(SP)\} \end{array}$$

**Hiding:** Symbols of a specification can be hidden by "translating" them against signature morphisms using the hiding operator. This allows symbols to be hidden in the signature of a specification while they still have an effect on the models, that is, any axioms involving the hidden symbols still hold true in all models of the hidden specification.

For any signature morphism $\sigma : \Sigma \to \Sigma'$ and $\Sigma'$-specification $SP'$, $SP'$ **hide** $\sigma$ is a specification with:

$$\begin{array}{lcl} \mathbf{Sig}(SP' \textbf{ hide } \sigma) & := & \Sigma \\ \mathbf{Mod}(SP' \textbf{ hide } \sigma) & := & \{M'|_\sigma \in |\mathbf{mod}(\Sigma)| \mid M' \in Mod(SP')\} \end{array}$$

These building operators form a kernel structuring language for specifications. Although they are quite simple, they allow for composing specifications in a flexible manner [Mos02].

Structured specifications constructed of only presentations, unions and renamings can be *flattened* into presentations as follows:

$$\begin{array}{lcl} \mathit{flatten}(\langle \Sigma, \Gamma \rangle) & := & \langle \Sigma, \Gamma \rangle \\ \mathit{flatten}(SP_1 \textbf{ and } SP_2) & := & \langle \Sigma, \Gamma_1 \cup \Gamma_2 \rangle \\ & & \text{where } \mathit{flatten}(SP_i) = \langle \Sigma, \Gamma_i \rangle \ (i = 1, 2) \\ \mathit{flatten}(SP \textbf{ rename } \sigma) & := & \langle \Sigma', \sigma(\Gamma) \rangle \\ & & \text{where } \mathit{flatten}(SP) = \langle \Sigma, \Gamma \rangle \text{ and } \sigma : \Sigma \to \Sigma' \end{array}$$

The flattening operation preserves the signature and models of specifications [Mos02].

### 4.7.2 An Extended Structuring Language

Here, we present an extension of the kernel structuring language presented in Section 4.7.1, which CASL uses to offer more elaborate and complex operators. These extended operators can all be expressed in terms of the kernel specification language.

**Union of arbitrary specifications:** We define the union of specifications of different signatures following [Mos00]. The union operator is only allowed under the condition that the underlying institution allows for union of signatures. This means that there is an additional partial function which takes two signatures and produces their union. For any $\Sigma_1$-specification $SP_1$ and $\Sigma_2$-specification $SP_2$ the union

$$SP_1 \textbf{ and } SP_2$$

is a specification which is equivalent to

$$(SP_1 \textbf{ rename } \sigma_1) \textbf{ and } (SP_2 \textbf{ rename } \sigma_2)$$

where $\sigma_1 : \Sigma_1 \to \Sigma'$, $\sigma_2 : \Sigma_2 \to \Sigma'$, and $\Sigma'$ is the union of the signatures $\Sigma_1$ and $\Sigma_2$. This construction is only defined when $\Sigma'$ is defined.

**Extension of specifications** We allow a specification $SP$ to be extended with new symbols and axioms contained within a specification fragment $SP'$ following the intuitive explanations in [Mos04]. A specification fragment $SP'$ is not a complete specification, but a partial one that relies on symbols declared in another specification $SP$ (a complete specification must be obtained by combining $SP$ and $SP'$). For any $\Sigma$-specification $SP$ and specification fragment $SP'$ the specification

$$SP \textbf{ then } SP'$$

can be replaced by the specification

$$(SP \textbf{ rename } \sigma) \textbf{ and } SP''$$

where $SP''$ is the specification which is the same as that of $SP'$ but with all the symbols from $\Sigma$ added to its signature $\Sigma''$. The signature morphism $\sigma : \Sigma \to \Sigma''$ is the inclusion of $\Sigma$ into $\Sigma''$.

**Generic specifications** We now introduce further constructions, following [Mos04], to allow for named specifications possibly with parameters (i.e., generic specifications). Named specifications can be referenced by later specifications where all parameters, if any, must be instantiated.

We first illustrate the use of generic specifications and instantiations with the following example [EBK+02]:

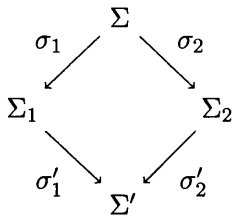
**spec** LIST[**sort** *Elem*] **given** NAT =
  **free type** *List* ::= *nil* | *cons*(*Elem*; *List*)
  **op** *length* : *List* → *Nat*
**end**

Here, we wish to specify lists with the typical constant *nil* and *cons* operation. We also specify a *length* operation that will return the length of a list as a natural number. The sort *Nat* is imported via the **given** construct. The motivation for using this construct instead of a standard import will be made clear shortly. We do not specify arbitrary lists, but heterogeneous lists. Here, we parametrise heterogeneous lists with the parameter specification **sort** *Elem*. Thus, we specify lists relatively to the sort *Elem* with the goal of instantiating *Elem* to give lists of specific types. For instance,

LIST[BOOLEAN **fit** *Elem* ↦ *Boolean*]

would be the instantiation of lists with the sort *Boolean* from the standard CASL library specification BOOLEAN. We provide a fitting morphism mapping the sort *Elem* to *Boolean*, this too will be explained shortly. This specification could now be extended where we can use our new sort of lists of type Boolean in further constructions.

Generic specifications and instantiations are supported in institutions with the amalgamation property. An institution has the so called *amalgamation property*, if the signature category has pushouts and if for any pushout of signatures




the following hold:

- for any two models $M_1 \in |\mathbf{mod}(\Sigma_1)|$ and $M_2 \in |\mathbf{mod}(\Sigma_2)|$ such that $M_1|_{\sigma_1} = M_2|_{\sigma_2}$, there exists a unique model $M' \in |\mathbf{mod}(\Sigma')|$ such that $M'|_{\sigma'_1} = M_1$ and $M'|_{\sigma'_2} = M_2$ (we call $M'$ the *amalgamation* of $M_1$ and $M_2$); and
- for any two model morphisms $h_1 : M_1^A \to M_1^B$ in $\mathbf{mod}(\Sigma_1)$ and $h_2 : M_2^A \to M_2^B$ in $\mathbf{mod}(\Sigma_2)$ such that $h_1|_{\sigma_1} = h_2|_{\sigma_2}$, there exists a unique model morphism $h' : M'^A \to M'^B$ in $\mathbf{mod}(\Sigma')$ such that $h'|_{\sigma'_1} = h_1$ and $h'|_{\sigma'_2} = h_2$ (we call $h'$ the *amalgamation* of $h_1$ and $h_2$).

A generic specification is of the form

$$SP[SP_1]\ldots[SP_n] \textbf{ given } SP''_1, \ldots, SP''_m = Body$$

where $SP$ is a fresh name for the specification, each $SP_i$ is a formal parameter specification, each $SP''_i$ is an import specification and $Body$ is the body of the specification. Each formal parameter $SP_i$ can either be a reference to an earlier named specification or a specification itself. Each import specification $SP''_i$ must be a reference to an earlier named specification. The parameter specifications are used to indicate which parts of the generic specification are intended to vary with each reference to it. The imports in contrast are fixed and are shared between all the parameters, the body and also any instantiated parameters. We write

$$SP[SP_1]\ldots[SP_n] = Body$$

when the list of imports is empty, and

$$SP = Body$$

when there are also no parameters. The above generic specification is equivalent to the specification

$$SP = (SP''_1 \textbf{ and } \ldots \textbf{ and } SP''_m) \textbf{ then } (SP_1 \textbf{ and } \ldots \textbf{ and } SP_n) \textbf{ then } Body \ .$$

**Instantiations** Now that we allow generic specifications, all that is left to allow is instantiated specifications. Instantiated specifications are not as simple to formulate and require several side conditions in order to be well-formed and defined.

An instantiation of the generic specification $SP$ (above) is a specification

$$SP[FA_1]\ldots[FA_n]$$

where each $FA_i$ is an appropriate fitting argument. A fitting argument

$$SP_i' \textbf{ fit } SM_i$$

is a specification $SP_i'$ (the actual parameter) along with a fitting morphism $SM_i$. The fitting argument $FA_i$ is well-formed only if it uniquely determines a signature morphism from the signature of the formal parameter $SP_i$ to the actual parameter specification $SP_i'$. Furthermore, when there is more than one parameter, the separate fitting argument morphisms have to be compatible, that is, their union has to yield a single morphism from the union of the formal parameters to the union of the actual parameters.

Each fitting argument $FA_i$ is regarded as an extension of the union of the imports. The fitting argument morphism has to be the identity on all symbols declared by the imports $SP_1''$, ..., $SP_m''$ of the generic specification $SP$. Any symbol declared explicitly in the parameter (and not only in the import) must be mapped to a symbol declared explicitly in the actual parameter specification. If these conditions are meet then the above instantiation is equivalent to the specification

$$((SP_1'' \textbf{ and } \ldots \textbf{ and } SP_m'') \textbf{ then } (SP_1 \textbf{ and } \ldots \textbf{ and } SP_n) \textbf{ then } Body \textbf{ rename } FM) \textbf{ and } SP_1' \textbf{ and } \ldots \textbf{ and } SP_n'$$

where $FM$ is the signature morphism yielded by the union of the fitting arguments $FA_1, \ldots, FA_n$ and each $SP_i'$ is the specification associated with the fitting argument $FA_i$.

The instantiation is only defined when the models of $SP_i'$ reduced by the signature morphism determined by $SM_i$ are models of the formal parameter specification $SP_i$. The instantiation is only well-formed when the result signature is a pushout of the body and formal parameter signatures. That is, if the translated body

$$((SP_1'' \textbf{ and } \ldots \textbf{ and } SP_m'') \textbf{ then } (SP_1 \textbf{ and } \ldots \textbf{ and } SP_n) \textbf{ then } Body \textbf{ rename } FM)$$

and the union of the actual parameter specifications

$$SP_1' \textbf{ and } \ldots \textbf{ and } SP_n'$$

share any symbols, then these symbols have to be translations (along $FM$) of symbols that share in the extension of the imports by the formal parameters

$$(SP_1'' \textbf{ and } \ldots \textbf{ and } SP_m'') \textbf{ then } (SP_1 \textbf{ and } \ldots \textbf{ and } SP_n) \ .$$

In CASL, two sorts share if they are identical, and two function or predicate symbols share if they are in the overloading relation. For further details see [Mos00].

The imports are necessary in order to share information between the body of the generic specification and the actual parameters used within instantiations. For example, consider again the generic specification of lists:

**spec** LIST[**sort** *Elem*] **given** NAT =
  **free type** *List* ::= *nil* | *cons*(*Elem*; *List*)
  **op** *length* : *List* → *Nat*
**end**

The sort *Nat* (provided by the specification NAT) is required in the body in order to specify the *length* operation. One would like to be able to instantiate this specification with the specification NAT to form lists of natural numbers, that is

LIST[NAT **fit** $Elem \mapsto Nat$] .

This instantiation is perfectly legal and well formed. However, had the generic specification referenced NAT within the body (i.e., imported it) instead of using the **given** construct then the instantiation would be ill-formed as it would violate the previous pushout condition. Thus, the **given** construct was introduced with the condition that the body and actual parameters can only share symbols which have originated from **given** specifications. This construction then makes instantiations such as the one above possible.

This concludes our presentation of the institution independent structuring mechanisms. In this chapter we have presented the formal notion of institutions and presented several examples. We have presented the institution $PCFOL^{=}$ and its extension $SubPCFOL^{=}$, which is the underlying institution of CASL. We then presented $ResSubPCFOL^{=}$ which is a sub-institution of $SubPCFOL^{=}$ where the signature category has been restricted. This restriction will be used in Chapter 7 to create alphabets out of many-sorted algebras. Following this, we defined the so called data-logic, which provides the ability to ask the right types of "questions" required by the CSP semantics. The models and models morphisms of the data logic are in one-one correspondence to the models and model morphisms of $ResSubPCFOL^{=}$. We use the data-logic in Chapter 8 when we formalise CSP-CASL as institutions. Following the presentation of the data-logic, we showed the process algebra CSP also forms institutions, thus demonstrating that institutions are not restricted to the setting of algebraic specification. Finally, we presented an institution independent kernel structuring language and some extensions which allow for basic specifications, unions of specifications, translations, hidings, generic specifications and instantiated specifications. This structuring language will be made available to CSP-CASL once we formalise CSP-CASL as institutions in Chapter 8 and prove that CSP-CASL has a suitable amalgamation property.

*Chapter 5*

# Original CSP-CASL (2006)

**Contents**



The aim of CSP-CASL [Rog06] was to develop a specification language, integrating the description of processes and data, which was suitable for the description of reactive systems. CSP-CASL would allow expressive data to be specified in CASL allowing the use of loose semantics, sub-sorting and partiality. One could then specify a CSP process which uses the specified data as communications.

The integration was to be deliberately lightweight so that tools could be reused. The tools HETS [MML07] and Isabelle/HOL [Pau94, NPW02] would be used to deal with the CASL aspects and the interactive theorem prover CSP-Prover [IR, IR05] for the CSP aspects.

In this chapter we describe CSP-CASL as it was originally designed, closely following [GKOR09]. We discuss tool support and present an example capturing part of the electronic payment standard EP2 (introduced in Chapter 1). We also show how CSP-CASL refinements can be used, not only to validate development steps, but also to verify properties such as deadlock freedom. Finally, we discuss the current limitations of CSP-CASL and how we intend to overcome them.

## 5.1 The Design of CSP-CASL

CSP-CASL was initially presented by Roggenbach in 2006 and was the starting point for the integration of the languages CSP and CASL. The general idea of CSP-CASL is to use CSP to describe the dynamics of a system as processes where the communications are values of data

types loosely specified in CASL. In this way it is possible to describe data in a rich setting which can then be used in an intuitive formalism for describing processes.

By using CASL, data can be easily described at all levels of abstraction, from high levels where loose semantics plays a role, to concrete levels where initial semantics are necessary. The full language of CASL can be used in the specification of data, namely sub-sorted partial first order logic with sort generation constraints and equality. CSP provides a convenient way to describe the behaviour of systems (see Chapter 2). Whilst reactive behaviour can be described in CASL alone, CSP provides a more elegant and well suited environment for the task. All standard CSP operators are available, such as multiple prefix choice, various parallel operators, operators for non-deterministic choice, hiding, renaming and communication over channels (see Chapter 2).

Syntactically, a CSP-CASL specification with name $N$ consists three distinct parts, namely, a data part, an optional channel part and a process part:

**ccspec** $SP =$ **data** $D$ **channel** $C$ **process** $P$ **end**

The data part $D$ is a structured CASL specification which is used to describe all the data that is necessary in the channel and process parts. The channel part $C$ allows the optional declaration of channels (see Section 2.2.3) typed according to the data part. Finally, the process part $P$ allows the description of an unnamed process (possibly with sub-processes using a let notation) written in CSP. Within the CSP process CASL terms are used as communications, CASL sorts denote sets of communications, relational renaming is described by binary CASL predicates and unary functions, and the CSP conditional construct uses CASL formulae as conditions.

Semantically CSP-CASL follows the traditional process algebra approach. Each specification defines a family of process denotations, where each model of the data part $D$ gives rise to exactly one process denotation (built relatively to the data model) for the process part $P$. CSP-CASL is generic in the CSP semantics that can be used.

The various CSP semantics are built relative to a fixed set of communications (see Chapter 2), where the semantic clauses involve various test functions over this set. To this end, CSP-CASL's semantical construction provides what we call a *data type of communications*, which, besides an alphabet of communications, defines the following functions:

- test on equality for arbitrary CASL terms
  (can two communications synchronise?),
- 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?), and
- satisfaction of a CASL first order formula
  (is the formula of the conditional construct true?).

These test functions, living on the alphabet, can be lifted to formulae in CASL. This data type of communications makes the language CSP-CASL generic in the choice of a specific CSP semantics. A CSP semantics can be used with CSP-CASL if the only operations it requires are

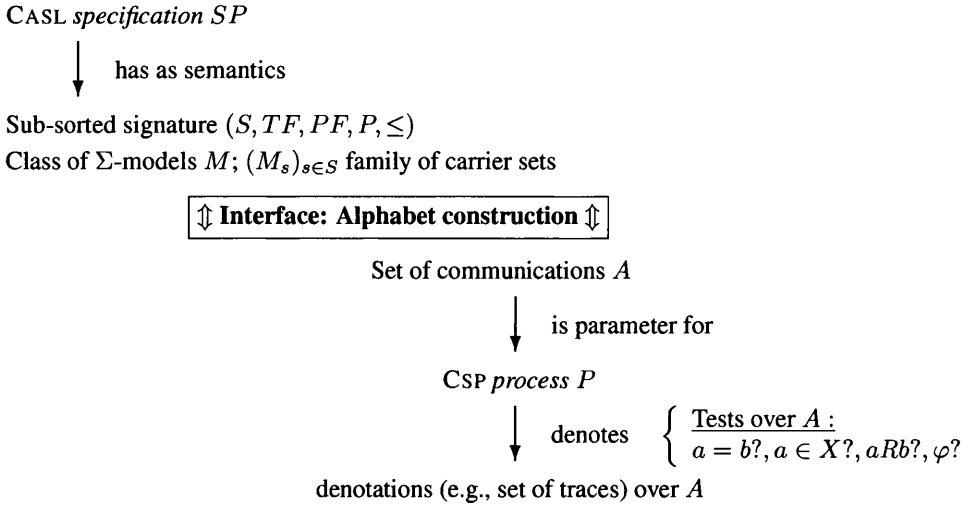



Figure 5.1: Translating data into an alphabet of communications [GKOR09].

covered by the above data type. The above listed, seemingly small set of test operations, allows for all denotational semantics described in [Ros05], namely the Traces semantics, Failures/Divergences semantics and Stable-Failures semantics.

The data types specified by algebraic specification consist of *many-sorted algebras*. The data type of communications required by the process algebraic semantics is a *one-sorted algebra*. Thus, in order to integrate data into processes, we need to turn a many-sorted algebra into one set of values such that the above described tests are closely connected with the original data type. Figure 5.1 depicts the construction. We illustrate our construction here for the example of many-sorted total algebras.

There are two natural ways to define the alphabet of communications in terms of the carrier sets of a CASL model: union and disjoint union of all carrier sets. To illustrate the effect of both possibilities, consider the following CSP-CASL specification (presented in [Rog06]):

**ccspec** SP =
  **data sorts** $S, T$
    **ops** $c : S; d : T$
  **process** $c \rightarrow$ SKIP $\|$ $d \rightarrow$ SKIP
**end**

Its data part, written in CASL, provides two constants $c$ and $d$ of sort $S$ and $T$, respectively. The process part, written in CSP with CASL terms denoting communications, combines two processes using the synchronous parallel operator, that is, they have to agree on all actions.

The question is, may $c$ and $d$ synchronise or not? In all the various CSP semantics, $c$ and $d$ synchronise if and only if they are equal. Now consider two isomorphic CASL models $M$ and

$N$ of the data part:

$$M_S = \{*\},\ M_T = \{+\},\ (c)_M = *,\ (d)_M = +$$
$$N_S = N_T = \{\sharp\},\ (c)_N = (d)_N = \sharp$$

Choosing the union of all carrier sets as the alphabet has the effect, that $c$ and $d$ do not synchronise for algebra $M$ while they synchronise for algebra $N$. Thus, isomorphic algebras give rise to different behaviour. Therefore, we define the alphabet to be the disjoint union – with the consequence that $c$ and d do not synchronise.

Similar 'experiments' for partiality, sub-sorting, and the combination of both – see [Rog06] – finally lead to an alphabet construction

$$Alph(M) = (\biguplus_{s \in S} (M_s \cup \{\bot\}))/_{\sim_M} .$$

Here, $M$ is a CASL model (actually a *ResSubPCFOL*$^=$ model) and $S$ is the set of all sorts declared in the data part. The special element $\bot$ is added to each carrier set and encodes partiality, while $\sim_M$ is an equivalence relation which – on the alphabet level – deals with sub-sorting. We do not discuss the alphabet construction or the equivalence relation further here, but simply note that the alphabet construction flattens a many-sorted model to a flat set of communications in a reasonable way. The alphabet construction and the equivalence relation will be presented in depth in Chapter 7.

## 5.2 CSP-CASL's Semantics and Refinement

The channel part of a CSP-CASL specification can be encoded in the data part, thus every CSP-CASL specification $(D, C, P)$ can be written in a form $(D', P')$ – see [Kah10] for details. From this point onwards we assume such a step has taken place and do not discuss channels directly.

Let $\mathcal{D}$ be a fixed CSP semantics, that is, the Traces semantics, the Failures/Divergences semantics or the Stable-Failures semantics. A CSP-CASL specification $(D, P)$ leads to a family of CSP denotations indexed by CASL models, that is,

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

where every data model $M \in \mathbf{Mod}(D)$ gives rise to a single CSP denotation in the chosen CSP semantics, that is $d_M \in \mathcal{D}(Alph(M))$.

Refinement [Rog06, KR09, Kah10] in CSP-CASL allows for the development of specifications and is naturally based upon refinement for CSP and CASL. During development of a specification one might extend the specification with new symbols and functionality (horizontal development) or reduce the level of abstraction (vertical development). Both of these types of development are handled by CSP-CASL's refinement notion. 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 means less non-deterministic choice.

Let $\sigma : \Sigma \to \Sigma'$ be a *ResSubPCFOL*$^=$ signature morphism. Let $(d_M)_{M \in I}$ and $(d'_{M'})_{M' \in I'}$ be families of process denotations (indexed by classes of $\Sigma$- and $\Sigma'$-models) over signatures $\Sigma$ and $\Sigma'$ respectively. Then,

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

where $I'|_\sigma = \{M'|_\sigma \mid M' \in I'\}$. We drop the superscript $\sigma$ when it is the identity signature morphism. The function $\hat{\alpha}^{\mathcal{D}}_{\sigma,M'} : \mathcal{D}(Alph(M')) \to \mathcal{D}(Alph(M'|_\sigma))$ takes denotations over the alphabet produced by $M'$ and translates them to appropriate denotations in the reduced alphabet produced by the model $M'|_\sigma$. This function will be described in detail in Chapter 7, but for now it is enough to know that it has a reasonable definition and suitable properties, such as preserving CSP refinements. Given CSP-CASL specifications $SP = (D, P)$ and $SP' = (D', P')$, by abuse of notation we also write

$$(D, P) \rightsquigarrow^{\sigma}_{\mathcal{D}} (D', P')$$

if the above refinement notion holds for the denotations of $SP$ and $SP'$, respectively. Again we drop the superscript $\sigma$ when it is the identity signature morphism.

Deadlock analysis can also be performed using refinement. First we describe what deadlock freedom means. A CSP-CASL specification is deadlock free, if all its possible models are deadlock free. On the semantical level, we capture this as follows.

Let $(d_M)_{M \in I}$ be a family of process denotations over the Stable-Failures semantics, that is, $d_M = (T_M, F_M) \in \mathcal{F}(Alph(M))$ for all $M \in I$.

- A denotation $d_M$ is deadlock free if $(s, X) \in F_M$ implies that $X \neq Alph(M)^{\checkmark}$. That is, at any point in the execution of a process, we may not refuse the entire alphabet.
- $(d_M)_{M \in I}$ is deadlock free if for all $M \in I$ it holds that $d_M$ is deadlock free.

The most abstract deadlock free CSP-CASL specification over a CASL signature $\Sigma$ with a set of sort symbols $S$ is defined as:

**ccspec** $\mathrm{DF}_\Sigma =$
  **data** ... declaration of $\Sigma$ ...
  **process** $DF = \sqcap_s :: S \, ( \sqcap x :: s \to DF) \sqcap \mathrm{SKIP}$
**end**

We observe that $\mathrm{DF}_\Sigma$ is deadlock free (see [Kah10] for details). This result on $\mathrm{DF}_\Sigma$ extends to a complete proof method for CSP-CASL deadlock analysis:

**Theorem 5.1** A CSP-CASL specification $(D, P)$ is deadlock free if and only if $\mathrm{DF}_\Sigma \rightsquigarrow_{\mathcal{F}} (D, P)$ where $\Sigma$ is the signature of $D$.

This result shows that CSP-CASL's refinement notion is not only useful in the context of specification development, but can also be used to analyse specifications and establish properties about them.

Shortly, we present an example of using CSP-CASL to model the electronic payment standard EP2. However, we first discuss tool support as our example uses tools in order to establish formal CSP-CASL refinements.

## 5.3 Tool Support

Here, we discuss the tool support available for CSP-CASL which allows one to establish refinement relations between CSP-CASL specifications. CSP-CASL-Prover [O'R08] is an interactive theorem prover based upon HETS [MML07] and CSP-Prover [IR, IR05] (which itself is based upon the interactive theorem prover Isabelle/HOL [Pau94, NPW02]). CSP-CASL-Prover is a tool dedicated to proving refinement statements between CSP-CASL specifications. Here, we briefly present how CSP-CASL refinements can be proven using HETS and CSP-CASL-Prover.

Kahsai presents the following decomposition rule [KR09] for CSP-CASL specifications.

$$\frac{\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D) \qquad (D', \sigma(P)) \rightsquigarrow_{\mathcal{D}} (D', P')}{(D, P) \rightsquigarrow^{\sigma}_{\mathcal{D}} (D', P')}$$

where $\sigma : \Sigma \to \Sigma'$ is a CSP-CASL signature morphism, $D$ and $D'$ are CASL specifications with signature $\Sigma$ and $\Sigma'$ respectively, and $\sigma(P)$ is the translation of the CSP-CASL process $P$ along the signature morphism $\sigma$.

This result shows that a CSP-CASL refinement can always be broken down into a data only refinement (i.e., $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$, also written $D \rightsquigarrow_\sigma D'$) and a separate process only refinement (i.e., $(D', \sigma(P)) \rightsquigarrow_{\mathcal{D}} (D', P')$). This is the basis for tool support that allows one to establish refinement relations on CSP-CASL specifications.

Data only proof obligations of the form $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$ (i.e., a CASL refinement) can be discharged with HETS, see Section 3.5 for details. Process refinements of the form $(D', \sigma(P)) \rightsquigarrow_{\mathcal{D}} (D', P')$ (where the data part remains constant) can be proven with CSP-CASL-Prover (see [O'R08] for details).

CSP-CASL-Prover takes CSP-CASL specifications and refinement statements as input and translates these into Isabelle theory files suitable for use with the interactive theorem prover CSP-Prover. One can then interactively prove within CSP-Prover whether the refinement statements hold or not. There are two main ways to use CSP-Prover, the first way is to rewrite processes using rules like those discussed in Section 2.3 until syntactic equality of processes is reached. The second way, is to apply the semantics and show set inclusion of the denotations. Usually, the first method is preferred as CSP-Prover offers various Isabelle tactics which automate this process to a large extent.

Together, the decomposition rule, HETS and CSP-CASL-Prover can be used to provide tool support for establishing CSP-CASL refinements between specifications. In order to establish a refinement, one can first break it down into a data only refinement and process only refinement. Then, prove each of these refinements separately using HETS and CSP-CASL-Prover.

Deadlock analysis can also be performed by CSP-CASL-Prover. Refinements of the form $\mathrm{DF}_\Sigma \rightsquigarrow_{\mathcal{F}} (D, P)$, as discussed in Section 5.2, can be decomposed into a data only refinement and a process only refinement. Thus, CSP-CASL-Prover can be used for deadlock analysis of CSP-CASL specifications. See [O'R08] for full details of CSP-CASL-Prover.

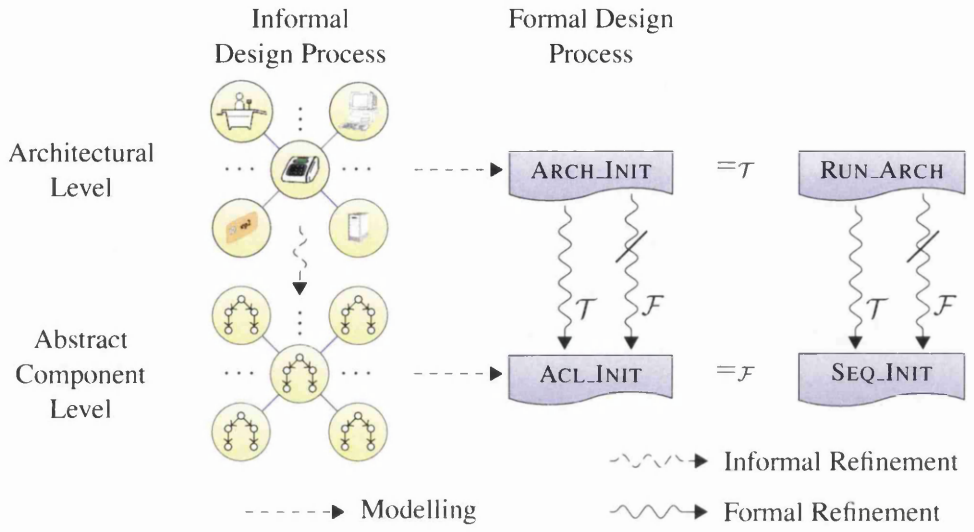



Figure 5.2: Refinement in EP2 [GKOR09].

## 5.4 Towards a Verification of EP2

We now present an example of specifying EP2 in CSP-CASL closely following [GKOR09]. We specify a dialogue of the EP2 system introduced in Chapter 1. We establish formal refinements between levels of abstractions and prove deadlock freedom of the dialogue.

We consider two levels of the EP2 specification, namely, the *architectural level* (ARCH) and the *abstract component level* (ACL). We choose a dialogue between the *Terminal* and the *Acquirer*. In this dialogue, the terminal and the acquirer are supposed to exchange initialisation information. For presentation purposes we study here only a nucleus of the full dialogue, which however, exhibits all technicalities present in the full version.

Our notion of CSP-CASL refinement mimics the informal refinement step present in the EP2 documents: there, the first system design sets up the interface between the components (architectural level), then these components are developed further (abstract component level). We now demonstrate how we can capture such an informal development in a formal way, see Figure 5.2.

We first specify the data involved using CASL only. The data specification of the architectural level (D_ARCH_GETINIT) requires only the existence of sets of values:

**spec** D_ARCH_GETINIT =
  **sort** *D_SI_Init*
**end**

In the EP2 system, these values are communicated over channels. Data of sort *D_SI_Init* is interchanged on a channel *C_SI_Init* linking the terminal and the acquirer. On the architectural level, both these processes just 'run', that is, they are always prepared to communicate an event from *D_SI_Init* or to terminate. We formalise this in CSP-CASL:

**ccspec** ARCH_INIT =
  **data** D_ARCH_GETINIT
  **channel** *C_SI_Init* : *D_SI_Init*
  **process let** *EP2Run* = (*C_SI_Init* ? *x* :: *D_SI_Init* → *EP2Run*) □ SKIP
    *Acquirer* = *EP2Run*
    *Terminal* = *EP2Run*
  **in** *Terminal* ||[ *C_SI_Init* ]|| *Acquirer*
**end**

The overall unnamed process in this specification is *Terminal* ||[ *C_SI_Init* ]|| *Acquirer* which is the parallel combination of *Acquirer* and *Terminal* where they must synchronise over the channel *C_SI_Init*.

On the abstract component level (D_ACL_GETINIT), data is refined by introducing a type system on messages. In CASL, this is realised by introducing sub-sorts of *D_SI_Init*. For our nucleus, we restrict ourselves to four sub-sorts, the original dialogue involves twelve of them.

**spec** D_ACL_GETINIT =
  **sorts** *SesStart*, *SesEnd*, *DataRequest*, *DataResponse* < *D_SI_Init*
  **ops** *r* : *DataRequest*; *e* : *SesEnd*
  • ∀ *x* : *DataRequest*; *y* : *SesEnd* • ¬ *x* = *y*
  • ∀ *x* : *DataRequest*; *y* : *SesStart* • ¬ *x* = *y*
  • ∀ *x* : *DataResponse*; *y* : *SesEnd* • ¬ *x* = *y*
  • ∀ *x* : *DataResponse*; *y* : *SesStart* • ¬ *x* = *y*
**end**

In the above specification, the axioms prevent confusion between the different sorts. The sorts represent messages for starting a session, ending a session, requesting data and responding to a request for data. Using this data, we can specify the EP2 system at the abstract component level in CSP-CASL. In the process part, the terminal (*TerInit*) initiates the dialogue by sending a message of type *SesStart*; on the other side, the acquirer (*AcqInit*) receives this message. In *AcqConf*, the acquirer takes the internal decision either to end the dialogue by sending the message *e* of type *SesEnd* or to send another type of message. The terminal (*TerConf*), waits for a message from the acquirer, and depending on the type of this message, the terminal engages in a data exchange. The system as a whole consists of the parallel composition of terminal and acquirer:

**ccspec** ACL_INIT =
  **data** D_ACL_GETINIT
  **channel** *C_SI_Init* : *D_SI_Init*
  **process let** *AcqInit* = *C_SI_Init* ? *session* :: *SesStart* → *AcqConf*
    *AcqConf* = *C_SI_Init* ! *e* → SKIP
      □ *C_SI_Init* ! *r* → *C_SI_Init* ? *resp* :: *DataResponse* →
        *AcqConf*
    *TerInit* = *C_SI_Init* ! *session* :: *SesStart* → *TerConf*
    *TerConf* = *C_SI_Init* ? *confMess* :: *D_SI_Init* →

```
                    if(confMess : DataRequest)
                    then C_SI_Init ! resp :: DataResponse → TerConf
                    else if(confMess : SesEnd) then SKIP else STOP
        in TerInit ||[C_SI_Init]|| AcqInit
end
```

Here, we have made a development step in the specification of the dialogue. We have decreased the level of abstraction and introduced more detail to the protocol. We can check that this is a reasonable development step in CSP-CASL by showing a formal refinement between the specifications.

**Theorem 5.2** ARCH_INIT $\rightsquigarrow^{\sigma}_{\mathcal{T}}$ ACL_INIT

*Proof.* Using tool support (described in Section 5.3), we establish this refinement by introducing two intermediate specifications RUN_ARCH and SEQ_INIT:

```
ccspec RUN_ARCH =
    data D_ARCH_GETINIT
    channel C_SI_Init : D_SI_Init
    process let EP2Run = (C_SI_Init ? x :: D_SI_Init → EP2Run) □ SKIP
        in EP2Run
end

ccspec SEQ_INIT =
    data D_ACL_GETINIT
    channel C_SI_Init : D_SI_Init
    process let SeqStart = C_SI_Init ! session :: SesStart → SeqConf
                SeqConf = C_SI_Init ! e → SKIP
                        □ C_SI_Init ! r → C_SI_Init ! resp :: DataResponse →
                          SeqConf
        in SeqStart
end
```

With CSP-CASL-Prover we proved: ARCH_INIT $=_{\mathcal{T}}$ RUN_ARCH. Now we want to prove that RUN_ARCH $\rightsquigarrow^{\sigma}_{\mathcal{T}}$ SEQ_INIT. Using HETS we showed the CASL refinement D_ARCH_GETINIT $\rightsquigarrow_{\sigma}$ D_ACL_GETINIT (i.e., **Mod**(D_ACL_GETINIT)$|_{\sigma} \subseteq$ **Mod**(D_ARCH_GETINIT)). Now, we formed the specification (D_ACL_GETINIT, $P_{\text{SEQ\_INIT}}$) and showed in CSP-CASL-Prover that, over the Traces semantics $\mathcal{T}$, this specification refines to SEQ_INIT. Here, $P_{\text{SEQ\_INIT}}$ denotes the process part of SEQ_INIT. O'Reilly et al. [ORI09] proves ACL_INIT $=_{\mathcal{F}}$ SEQ_INIT. As stable failure equivalence implies trace equivalence, we obtain ACL_INIT $=_{\mathcal{T}}$ SEQ_INIT. Figure 5.2 summarises this proof structure. □

As ACL_INIT involves parallel composition, it is possible for this system to deadlock. Furthermore, the process *TerConf* includes the CSP process STOP within one branch of its conditional operator. Should this branch of *TerConf* be reached, the whole system will be in deadlock. The dialogue between the terminal and the acquirer for the exchange of initialisation

messages has been proven to be deadlock free in [ORI09]. Specifically, it has been proven that the following refinement holds: SEQ_INIT $\rightsquigarrow_{\mathcal{F}}$ ACL_INIT, where SEQ_INIT is a sequential system. Sequential systems are regarded to be deadlock free. With our proof method from Section 5.2, we can strengthen this result by actually proving that SEQ_INIT is deadlock free. To this end, we proved with CSP-CASL-Prover that $DF_{\Sigma} \rightsquigarrow_{\mathcal{F}}$ SEQ_INIT, where $DF_{\Sigma}$ is the least refined deadlock free specification where $\Sigma$ is the signature of the data part of SEQ_INIT. As Stable-Failures refinement preserves deadlock freedom and as refinement is transitive, we conclude that ACL_INIT is deadlock free. Details of the various proofs can be found in [Kah10].

## 5.5 Current Limitations of CSP-CASL

CSP-CASL follows the traditional process algebra approach for its semantics. Typically in process algebra each process defines a single system. In CSP-CASL, a specification defines a single process denotation for each data model. One can use a *let* construction within a specification to give the process part a little structure. However, the semantics forgets how the process was constructed and simply represents it as a single denotation. This causes difficulties for ideas such as code reuse, component based design and structured specifications.

Structuring is only supported within the data part. Within the process part one must specify an unnamed system, possibly using a let construct to specify sub-systems. As there is no structuring it is impossible to specify processes in one specification and reuse or import them into another specification. Thus, separate systems cannot be developed in isolation and combined at a later stage. This makes code reuse impossible for processes and severely limits component based design to only data aspects. The specification ACL_INIT in Section 5.4 is a good example of this. We were forced to specify the behaviour of the terminal and the acquirer in the same specification even though these are two separate components in EP2 design documents.

Furthermore, the semantics does not support loose processes. Loose process semantics would assign to a specification a class of CSP denotations for each data model. As an example of the use of such loose processes, consider the specification ARCH_INIT in Section 5.4. In this specification we were forced to bind the sub-process names *Acquirer* and *Terminal* to CSP processes. In this case we indeed wanted to bind them to the *EP2Run* process, however, if instead we wanted to leave the behaviour of either process completely open, we could not. Not only does the syntax of original CSP-CASL not support this, there is no way to support this in the original semantics either.

Furthermore, the lack of loose process semantics forbids the use of generic specifications and instantiations (see Section 4.7). This is because when using generic specifications the formal parameters capture the class of models that can be used as the parameter. The actual parameters then select suitable models out of the classes specified by the formal parameters. The CSP-CASL semantics does not use such model classes and thus generic specifications and instantiations cannot be supported.

In this thesis we re-design the semantics of CSP-CASL, whilst building on the progress made so far. We define the semantics using a model class approach that reconciles the process algebra world of single denotations and the algebraic specification world of model classes. Furthermore, we define a new refinement notion, which works with loose process semantics

and is based upon model class inclusion. Finally, we allow full structuring in both the data and process parts of CSP-CASL specifications, thus allowing for generic and instantiated CSP-CASL specifications. This in turn lends itself to component based design and compositional reasoning.

In this chapter we have discussed the design and goals of CSP-CASL. We have sketched the original formalisation of CSP-CASL and discussed the available tool support. We have presented an example using CSP-CASL where we modelled the electronic payment standard EP2. We also established formal refinements and verified deadlock freedom of our example. Finally, we considered the limitations of CSP-CASL and how we intend to overcome these with developments within this thesis.

*Chapter 6*

# Related Work

**Contents**



The integration of processes and data has become an interesting and highly active research area. This research area has partly risen due to the limitations of process algebras:

> "Basic process algebras such as CCS, CSP, and ACP are well suited for the study of elementary behavioural properties of communicating systems. However, when it comes to the study of more realistic systems these languages turn out to be less adequate. One main problem is that process algebras tend to lack the ability to handle data." [Lis]

Process algebras, such as ACP [BK84], CCS [Mil89], and CSP [Hoa78, Hoa85, Ros98, Sch99, AJS05, Ros05, Hoa06] use algebraic specification as the meta level to describe communications and events. This algebraic specification is ad-hoc and usually invented as required. Various extensions and combinations of algebraic specification and process algebra have been studied which formalise such integration. In this chapter we will briefly describe several of these extensions as well as other approaches to the integration of processes and data. Here, we provide a survey of alternatives to CSP-CASL and compare them to our approach. An in depth survey can be found in [ABR99]. We also discuss structuring where it is present.

Like CSP-CASL other reactive extensions to CASL have been studied, for instance: CCS-CASL, CASL-CHARTS, CO-CASL, and CASL-LTL. CCS-CASL [SAA01, SAA02] combines

CASL with the process algebra CCS. It offers only initial semantics and thus reformulates ideas from LOTOS (discussed shortly). CASL-CHARTS [RR00] is another approach which combines CASL with state charts. CASL-LTL [RAC00] and CO-CASL [MRS03] extend CASL with reactive components, namely, CASL-LTL extends CASL with labelled transition systems, while CO-CASL adds co-types. Both of these can be seen as providing a meta language, for instance, CCS has been formalised in both CASL-LTL and CO-CASL [RAC00, MSRR06].

## 6.1 Approaches using Initial Semantics for Data

The following languages combine data and processes using initial semantics for data. Whilst this approach works, it forces data to be fixed the moment it is needed within the modelling. This makes it difficult (if not impossible) to capture high abstraction levels of systems where an overview of the system should be modelled and not the technical details.

### 6.1.1 LOTOS

LOTOS (the Language of Temporal Ordering Specifications) [ISO89, BSS87, BB88] was developed within ISO (International Standards Organisation) specifically for the formal description of the OSI (Open Systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. LOTOS combines the algebraic specification language ACT-ONE [EM85] with an extension of the process algebra CCS.

LOTOS has been used in various practical applications and has an extensive tool set associated with it. EUCALYPTUS [Gar96], which stands for *European/Canadian* LOTOS *Protocol Tool Set*, is one such tool set. The tool consists of: static analysers, a simulator, a model generator, a model verifier, a C-code generator, a model viewer, a trace analyser, a test case generator and several other components. The model verifier is capable of deadlock and livelock detection.

ACT-ONE, which is the algebraic specification language used in LOTOS, which allows for the specification of data types using initial semantics and equational logic. A translation from ACT-ONE to first order logic with equality (a sub-language of CASL) has been defined in [Mos02]. While ACT-ONE uses initial semantics, CASL allows for both initial and loose semantics and also supports partiality and sub-sorting.

### 6.1.2 ELOTOS

ELOTOS (Enhanced LOTOS) [JTC01, Ver99] is the successor of LOTOS, the most significant enhancements include the introduction of a notion of quantitative time, new data types (including Booleans, characters, bits and integers), composed types (including records, sets and arrays), and a new module system. The module systems allows both data and processes to be specified in separate modules and imported into further modules. Both Modules and interfaces can be specified. The interfaces allow parts of the implementations to be hidden. Generic modules are also supported. As ELOTOS is based upon equivalence, as opposed to a refinement notion, there is no direct support for stepwise development, unlike CSP, CASL and CSP-CASL.

We now briefly present a short example by Verdejo [Ver99] to illustrate the syntax and features of ELOTOS. We specify a register that takes two input values, stores them, and then outputs them. First, we specify the following interface:

```
interface Register_Interface is
  type data
  process Register [ in1:data, in2:data, out1:data, out2:data ]
endint
```

Here, we have specified the register interface. This creates the names available in modules. We have created a new type named *data* and a new process name *Register*. The process name *Register* takes four gates as parameters, namely *in1*, *in2*, *out1* and *out2*. The former are intended to be used as input gates while the latter as output gates. All these gates communicate values of type *data*. At this point these are simply names and have no meaning bound to them, this is done in the following module:

```
module Register_Mod: [ Register_Interface renaming
                       (proc Register := Register_Nat) ] is
  type data renames nat endtype
  process Register_Nat [ in1:data, in2:data, out1:data, out2:data ] is
    var x1:data, x2:data in
    in1(?x1:data);
      ( in2(?x2:data)
      |||
      out1(!x1) );
      out2(!x2)
    endvar
  endproc
endmod
```

This is a module named *Register_Mod* which imports the register interface. During this import the process *Register* is renamed to *Register_Nat* to indicate that we wish to build a register that is tailored to dealing with natural numbers. Next, we declare that our type *data* will be a type synonym for the existing predefined type *Nat*. Following this, we give a definition to the process register. The register will use two variables *x1* and *x2* both of type *data* (i.e., *Nat*). These variables will hold the incoming values. We first receive a value on gate *in1* and bind this to variable *x1*, we then behave as the inner process. This inner process uses the interleaving operator which in this case gives the choice between outputting the value stored in variable *x1* on gate *out1* or receiving a second input on gate *in2* and binding it to the variable *x2*. Once both of these events happen, in either order, we finally output the variable *x2* on gate *out2*.

This example demonstrates two structuring operators, namely importing and renaming, and also how to specify processes that have some state information. Recursion within processes can be used to specify infinite behaviour. Modules in ELOTOS are ultimately used in specification blocks which are the top level structuring constructs.

### 6.1.3 CSP$_M$

CSP$_M$ (Machine Readable CSP) [Sca98, Ros05] is the input language for various tools including FDR (Failures Divergences Refinement) [FDR06] and PROBE (Process Behaviour Explorer) [Pro03]. FDR is a model checker for checking various refinement properties of CSP processes and also deadlock and livelock freedom. PROBE is a tool which allows one to explorer the behaviour of CSP processes interactively. CSP$_M$ is describe in [FDR06] and presented in detail in [Sca98].

CSP$_M$ provides a functional language which allows concrete data to be specified. This data can then be used with channels to form communications used in CSP processes. FDR does not provide sub-typing, but does allow some degree of partiality. Partial functions (such as divide) are provided but the situation of using undefined results in CSP processes is not properly catered for. CSP-CASL allows both sub-sorting and partiality where undefined results can be communicated in CSP.

### 6.1.4 PSF

PSF (Process Specification Formalism) [MV90] combines the process algebra ACP (Algebra of Communicating Processes) [BK84] with algebraic specification of data using ASF (Algebraic Specification Formalism) [BHK89]. PSF specifications consists of data modules and process modules. Data modules allow for the formalisation of data using initial semantics and equational logic, while process modules allows for processes to be described using ACP where actions may range over data types from the data part. The PSF toolkit provides several tools for PSF [PSF97, MV92] including: a compiler, term rewriter, simulator and animator.

## 6.2 Approaches using Loose Semantics for Data

The following languages combine data and processes using loose semantics for data. This allows systems to be captured at higher levels of abstraction than formalisms using initial semantics for data.

### 6.2.1 $\mu$CRL

$\mu$CRL (micro CRL also written as MCRL) [GP95, AG09] – where CRL is an abbreviation for Common Representation Language – allows data to be specified with loose semantics via equational logic with total functions. The Booleans are provided with a fixed interpretation. Processes are specified in a traditional algebraic style with syntax closely following ACP [BK84]. The semantics of $\mu$CRL is defined in terms of labelled transition systems via structural operational semantics. The states of the labelled transition systems correspond to process expressions while the transitions are labelled with actions. Branching simulation [vGW96] is used to establish equivalence relations between states. Analysis of $\mu$CRL specifications is supported via the $\mu$CRL toolset [Lis].

$\mu$CRL2 extends $\mu$CRL by adding extra constructs including various higher order constructs, predefined data types (including numbers, lists, sets, bags and higher order function types) and $\lambda$-calculus expressions.

## 6.3 An Object Orientated Approach: CSP-OZ

CSP-OZ [Fis97] is a combination of the object orientated state based formalism OBJECT-Z [Smi00] for describing data with the process algebra CSP. The specification language Z [WD96, Spi92] uses a model-orientated approach where state is modelled via input and output observations, where as CASL uses a property-orientated approach where the algebras consist of real functions. OBJECT-Z then adds object orientation to Z. The formal semantics of CSP-OZ is based on CSP's Failures/Divergences semantics, where as CSP-CASL is generic in the underlying denotational semantics.

In CSP-OZ a collection of objects are specified that interact with each other over channels. Each object has its own structure and behaviour. The objects are specified via paragraphs that introduce classes, global variables, functions and types. The overall specified system is then the collection of the inter-communicating objects.

## 6.4 A Deep Integration: CIRCUS

CIRCUS [WC01, WC02] provides a deeper integration of processes and data than the other approaches seen so far and allows for development of state-rich reactive systems based on refinement. This comes at the cost that tools need to be developed from scratch. Systems can be captured at various levels of abstraction from abstract models and designs to concrete programs. CIRCUS combines the specification language Z with the process algebra CSP and also provides a refinement notion for this setting.

CIRCUS comes equipped with a rich tool set which includes: a parser, a static type checker, a model checker, and a refinement checker. These tools are implemented as extensions of the CZT toolkit [MU05] for Z.

In general terms, data is specified via Z schemas, and processes via a combination of Z, CSP, and Dijkstra's command language. Z is used to define most of the data aspects while CSP is used to describe behaviour.

CIRCUS specifications consists of various paragraphs in series. Paragraphs come in a various variations: Z paragraphs, channel and channel set declarations, and process declarations. A Process declaration consists of a process name and a process body which may be defined from basic operations or using operators that combine other processes. In CIRCUS, CSP actions are allowed to be the Z schemas themselves, guarded commands, invocation of another action or a combination of these constructs using CSP operators. This is what gives CIRCUS the deeper integration and differs from CSP-CASL where the actions are simply values of data types defined by CASL. This paragraph style of specification is what allows circus specification to be structured. Process can be specified in one paragraph and used within later paragraphs.

CIRCUS also includes a refinement calculus, which allows stepwise development of specifications and programs. The semantics of CIRCUS is based on the UTP (Unified Theories

of Programming) [HJ98], a relational model that unifies programming theories across many different paradigms.

We now present a brief example by Oliveira et al. [OGC08] which illustrates CIRCUS specifications. This example specifies a chronometer which counts the passing of seconds and minutes.

$RANGE == 0 \mathinner{..} 59$
**channel** *tick*, *time*
**channel** *out*: $RANGE \times RANGE$

**process** $Chrono \mathrel{\widehat=}$
  **begin state** $AState == [\, sec, min: RANGE \,]$
  $AInit == [\, AState' \mid sec' = min' \land min' = 0 \,]$
  $IncSec == [\, \Delta AState \mid sec' = (\, sec + 1 \,) \bmod 60$
    $\land\ min' = min \,]$
  $IncMin == [\, \Delta AState \mid min' = (\, min + 1 \,) \bmod 60$
    $\land\ sec' = sec \,]$
  $Run \mathrel{\widehat=} (\, tick \rightarrow IncSec;\ ((\, sec = 0 \,)\ \&\ IncMin\,)$
    $\Box\ ((\, sec \neq 0 \,)\ \&\ Skip\,)))$
    $\Box\ (\, time \rightarrow out!(min, sec) \rightarrow Skip \,)$
  $\bullet\ (\, AInit;\ (\, \mu X \bullet (\, Run; X \,)))$
**process** $Clock \mathrel{\widehat=}$ **begin** $\bullet\ \mu X \bullet tick \rightarrow X$ **end**
**process** $TChrono \mathrel{\widehat=} (\, Chrono \ |[\ \{|\, tick \,|\}\ ]|\ Clock \,) \setminus \{|\, tick \,|\}$

First, we specify some data, that is, the range of values that will be allowed for seconds and minutes. Next, we define the channels *tick* and *time*. These carry no additional data and will just be simple actions that occur in the behaviour of the system.

Following this, we define a process paragraph that defines the the process *Chrono*. This process has a state *AState* which holds two values of type *Range* which records how many seconds and minutes have passed. We define the initial state as *Ainit* where both the seconds and minutes are set to zero. Next, we define two actions that increment the seconds and minutes. When the seconds are incremented to 60 they reset to zero, the same for the minutes. We next define the heart of the system, the run process. This process can do two branches. The first branch is when a tick event occurs. Following this *tick* event the chronometer increments the seconds and upon completion there is a further choice. If the seconds happen to have been reset back to zero then the minutes will be incremented, and if not then the process terminates via the CSP *Skip* operator. The second branch occurs when a *time* event occurs. This indicates that the user wishes for the device to output the current count of seconds and minutes. Upon this action happening, the device indeed outputs on the channel *out* both the amount of seconds and minutes that have elapsed. We now define the unnamed main process for the process paragraph *Chrono* which first sets the initial state and the repeatedly calls the *Run* process via fixed point recursion.

The *Clock* process is then defined which continuously counts time by outputting an infinite sequence of *tick* events. Finally, the *TChrono* process is defined by putting the *Clock* and *Chrono* processes parallel with the *Tick* event being hidden. From a users perspective (or

the environment) we can only interact with this *TChrono* process via the *time* action (which represents the user asking for the time) and the channel *out* which the device uses to output its data.

This example illustrates the differences with CSP-CASL. In CSP-CASL we communicate data specified via CASL. In CIRCUS the actions are Z paragraphs which can alter the state of the processes in complex ways. For instance, the *IncSec* action is not a simple action but involves arithmetic and the *mod* operator which controls the range of values that can be recorded in the state.

## 6.5 A Structured Approach to CSP: Wright

The Wright architectural description language [AG97] allows reasoning on typed processes for a sub-language of CSP– the renaming and hiding operators are missing, to name just a few. Semantically, Wright is restricted to a single CSP semantics, namely an early variant of the Failures/Divergences semantics.

Wright prescribes a strict specification style: specifications are called "connectors". Such a connector is built from several CSP processes. The processes that shall interact which each other are qualified with the keyword "role"; there is one process coordinating everything which is the so-called "glue". Here, we present an example from [AG97] which specifies a client and a server process, which are connected via a glue process:

**connector** *C-S-Connector* =
**role** *Client* = $(request!x \rightarrow result?y \rightarrow Client) \sqcap \S$
**role** *Server* = $(invoke?x \rightarrow return!y \rightarrow Server) \Box \S$
**glue** = $(Client.request?x \rightarrow Server.invoke!x \rightarrow Server.return?y \rightarrow$
$Client.result!y \rightarrow Glue) \Box \S$

Here, we specify a "connector" named *C-S-Connector* which consists of a server and client. The *client* sends a *request* message indicating that they wish for some action to be invoked, they then wait to receive a *result* message indicating the invoked action has completed. Once the *result* message is received the *client* repeats its behaviour. Instead of sending a request the *client* can choose non-deterministically to terminate via the § process (i.e., SKIP). The *server* process, on the other hand, sends an *invoke* message to start some action and then sends a *result* message and repeats. The *server* also offers the ability to terminate instead of sending an *invoke* message.

The *glue* process coordinates the the other two processes so that the server only sends an *invoke* message when the *client* has requested so. The *server* and *client* processes do not actually communicate with each other and instead go through the *glue* process. The *glue* process receives a *request* message from the *client*, then instructs the *server* to send an *invoke* message using the data received from the *client*. The *glue* then waits for the *server* to send a *result* message and relays this on to the *client* process. The result – achieved by combining all three processes with the synchronous parallel operator – is a process which only invokes an action after a request has been made.

There are strict naming rules which concern which actions the *glue* process can communicate in relation to the actions of the processes of type *role*. The *glue* process coordinates the overall system behaviour. The *role* and *glue* processes are restricted in the CSP operators that they can use, they must be sequential processes and must not make use of any parallel operators. They are however, semantically, combined using the synchronous parallel operator. This rather restricted scheme allows to provide an elegant, compositional proof rule for deadlock freedom: provided that a connector is deadlock-free and has the property to be "conservative", then a newly formed connector in which the roles are refined, is deadlock free as well.

While this deadlock analysis is quite elegant, at least in the original paper, there is no notion of development between Wright connectors: for example, a property such as deadlock freedom is propagated, however, it is not clear in which semantic relation the original connector and the newly gained connector are. Moreover, Wright does not cover data refinement. CSP-CASL, while allowing for the development of processes and data, also has elegant compositional proof rules.

## 6.6 An Institutional Approach: Zawlocki

Zawlocki [Zaw04] provides a framework for describing processes and data in a structured way. Zawlocki provides an institution for processes and data, where data is described using CASL, while reactive systems are described using the temporal logic CTL*. The aim is to provide a means for describing system architecture by using the institution independent architecture mechanism of CASL [Mos04]. For reactive systems, the choice of CTL* enables reasoning about safety, liveness and fairness properties.

Zawlocki's signatures are pairs $\Theta = (\Gamma, \Sigma)$, where

- $\Gamma$ is a set of action symbols,
- $\Sigma$ is a many-sorted first-order data signature.

The models are transition systems, where – given a signature $\Theta = (\Gamma, \Sigma)$ – a $\Theta$-system is a triple $S = (W, D, T)$ with

- $W$ a non-empty set of system states,
- $D : W \to Str(\Sigma)$ is a mapping assigning a data structure to each state, and
- $T \subseteq W \times \Gamma \times W$ is a set of transitions from states to states.

Inspired by CommUnity, Zawlocki's signature morphisms use a contravariant partial mapping for action symbols. Let $\Theta = (\Gamma, \Sigma)$ and $\Theta' = (\Gamma', \Sigma')$ be signatures. A signature morphism $\vartheta : \Theta \to \Theta'$ is a pair $(\gamma, \sigma)$, where

- $\gamma : \Gamma' \to ?\Gamma$ is a partial mapping,
- $\sigma : \Sigma \to \Sigma'$ is a first-order many-sorted signature morphism.

This contravariant construction forces Zawlocki to change the modal operators in the formula translation, for example,

$$\bar{\vartheta}(\phi_1 \,\mathbf{U}\, \phi_2) = \bar{\vartheta}(\phi_1)\, \mathbf{U}_{dom(\gamma)}\, \bar{\vartheta}(\phi_2) \ .$$

The validity of a formula under translation is restricted to those actions, which are present in the source signature. This leads to a logic where the modal operators need to be qualified by the set of actions for which they can be considered.

Another observation is that Zawlocki's framework fails to provide a mechanism which describes actions in an easy way: the set of action symbols $\Gamma$ needs to be written down directly by the specifier. It is a flat set, without any algebraic properties.

CSP-CASL differs from Zawlocki's approach in both these points. Firstly, data and processes are translated in a covariant way, this is possible thanks to the typing information attached to each process name. Secondly, the actions that a system can perform are specified using CASL, that is, they usually carry a rich algebraic structure.

## 6.7 Meta-Formalisms

There are several approaches which can be perceived as kinds of "meta-formalisms" where it is possible to specify whole process algebras within them, including their syntax and semantics. Here, we present two of these approaches.

### 6.7.1 CO-CASL

CO-CASL [MRS03, MSRR06] extends CASL by providing so-called co-types – see Figure 6.1 for the idea of how to dualize algebraic types. The co-algebraic types represent process behaviour, while the algebraic types cover data. CO-CASL is presented as an institution, where all the CASL institution independent structuring mechanisms can be applied. Extending these, there is also a structured **co-free** construct.

CSP-CASL differs from CO-CASL in that it is intended to be used as a concrete modelling language. That is, to specify real systems at various level of abstraction, as opposed to CO-CASL which is more suited to the description of specification formalisms themselves. Mossakowski et al. [MSRR06] formalise the process algebras CCS and CSP within CO-CASL. This meta-formalism allows one to reason about such specification formalisms themselves.

### 6.7.2 Rewriting logic and Maude

Rewriting logic, and thus Maude, are able to capture various forms of concurrency. For example, Verdejo and Martí-Oliet [VMO00] capture the process algebra CCS in Maude.

Figure 6.2 shows how to capture the syntax of CCS. The module *ACTION* establishes the data for use in CCS. Sub-sorting is used to introduce the sorts *Label* and *Action*. They provide a complement operation $\_^{\sim}$ where they specify that double complement is identity. The module *PROCESS* specifies the symbols needed to create processes. The sort *ProcessId* represents process identifiers which is a sub-sort of *Process*. They then mimic the CCS operations of prefix, summations, parallel composition and restriction.

| Algebra | Coalgebra |
|---|---|
| **type** = (partial) algebra | **cotype** = coalgebra |
| constructor | selector |
| generation | observability |
| **generated type** | **cogenerated (co)type** |
| = no junk | = full abstractness |
| = induction principle | = coinduction principle |
| no confusion | all possible behaviours |
| **free type** | **cofree cotype** |
| = absolutely initial data type | = absolutely final process type |
| = no junk + no confusion | = full abstractness + all possible behaviours |
| **free** { ... } = initial data type | **cofree** { ... } = final process type |

Figure 6.1: Summary of dualities between CASL and CO-CASL [MSRR06].

The operational semantics of CCS is then captured by conditional rewrite rules. This results in an effective simulator for CCS, however, without considering bisimilarity between processes on the operational semantics. As rewriting logic is reflective, it is possible to analyse the defined semantic setting.

Similarly to CO-CASL, the rewriting approach provides a powerful meta-formalism for analysing processes and data.

```
(fmod ACTION is
  protecting QID .
  sorts Label Act .
  subsorts Qid < Label < Act .

  op tau : -> Act .                          *** silent action

  op ~_ : Label -> Label .
  var N : Label .
  eq ~ ~ N = N .
endfm)

(fmod PROCESS is
  protecting ACTION .
  sorts ProcessId Process .
  subsort Qid < ProcessId < Process .

  op 0 : -> Process .                          *** inaction
  op _._ : Act Process -> Process [prec 25] .   *** prefix
  op _+_ : Process Process -> Process [prec 35] .
                                                *** summation
  op _|_ : Process Process -> Process [prec 30] .
                                               *** composition
  op _'[_/_'] : Process Label Label -> Process [prec 20] .
                *** relabelling: [b/a] relabels "a" to "b"
  op _\_ : Process Label -> Process [prec 20] .
                                               *** restriction
endfm)
```

Figure 6.2: Maude module capturing the syntax of CCS [VMO00].

# Part II

# Contributions

*Chapter 7*

# CSP-CASL Alphabet Construction

**Contents**



As we have seen in Chapter 2, CSP's syntax and semantics are constructed relative to some alphabet. In order to create CSP domains from CASL models, the CASL models first have to be flattened to sets. This chapter describes this process and also how to transform CASL model morphisms and CASL model reducts into appropriate functions between CSP domains. This forms the foundation for the CSP-CASL institutions presented in Chapter 8.

Here, we present two constructions. The first construction, in Section 7.1, lifts alphabets and alphabet translations to CSP domains and domain translations. We do this by forming two functors, one covariant and one contravariant, from the category **SET** to itself. In the second construction, in Section 7.2, we flatten many-sorted algebras and homomorphisms to alphabets and alphabet translations. In addition to this we also lift the reduct functor on the model categories to alphabets and alphabet translations.

These two constructions can then be composed, allowing the construction of CSP domains from CASL models, and domain translations from CASL model morphisms and model reducts. These constructions will then be used within the semantics of CSP-CASL in Chapter 8.

## 7.1 Construction 1: Lifting Alphabet Translations to CSP Domains

In this section, we describe how to transform alphabets and alphabet translations to CSP domains and domain translations, respectively.

Ultimately, we construct two functors, one covariant and one contravariant, which map from the category where objects are alphabets (i.e., sets) and morphisms are alphabet translations (i.e., total functions) to the category where objects are CSP domains and morphisms are

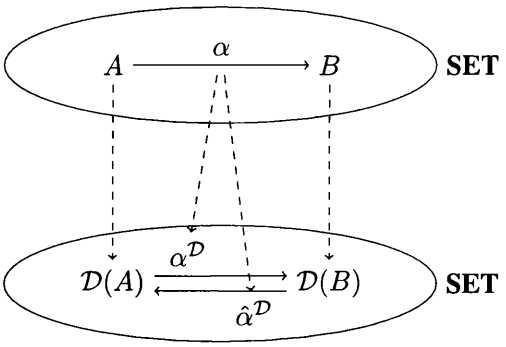



Figure 7.1: Lifting alphabets to CSP domains and alphabet translations to covariant and contravariant CSP domain translations.

CSP domain translations. Both of these categories are actually the category of **SET**, thus we create two functors from **SET** to **SET** where we lift alphabets to CSP domains built over such alphabets, and alphabet translations to translations between the respective CSP domains.

Figure 7.1 illustrates this construction. $A$ and $B$ are alphabets and $\alpha$ is an alphabet translation from $A$ to $B$. $\mathcal{D}(A)$ and $\mathcal{D}(B)$ are the CSP domains built relative to the alphabets $A$ and $B$ respectively. The alphabet translation *alpha* has been lifted to the covariant CSP domain translation $\alpha^{\mathcal{D}}$ and to the contravariant CSP domain translation $\hat{\alpha}^{\mathcal{D}}$. These functors are formed for each of the CSP semantics, namely the Traces semantics $\mathcal{T}$, the Failures/Divergences semantics $\mathcal{N}$ and the Stable-Failures semantics $\mathcal{F}$, that is, for each $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

These functors are built up using multiple sub-constructions. Firstly, in Section 7.1.1, we lift alphabets and alphabet translations to so called basic translations. Secondly, in Section 7.1.2, we lift basic translations to covariant CSP domain translations, which translate CSP domains along the underlying alphabet translations. Thirdly, in Section 7.1.3, we again lift basic translations, but this time, to form contravariant CSP domain translations, which translate CSP domains against the underlying alphabet translations. In Section 7.1.4 we look at how the covariant and contravariant CSP domain translations relate to each other. Section 7.1.5 defines the notion of the *top* element of a CSP domain and finally, Section 7.1.6 discusses what deadlock preserving properties the contravariant translation exhibits.

### 7.1.1 Lifting Alphabet Translations to Basic Translations

An alphabet translation is a function between two sets, which we call alphabets. It is simply the renaming of symbols from one alphabet to another, not necessarily injective nor surjective. In categorical terms this is the category **SET**. Here, we introduce some basic translation functions that will be used to define a lifting from alphabet translations to translations over the CSP semantics $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$. Technically, these are also functors from **SET** to **SET**.

Given an alphabet translation $\alpha : A \to B$ and constructions $\checkmark$, $*$, $*\checkmark$ and $\mathcal{P}\checkmark$ on sets (see Chapter 2), we extend the map $\alpha$ canonically to three maps as presented by Kahsai [Kah10]:

- To include the termination symbol $\checkmark$:

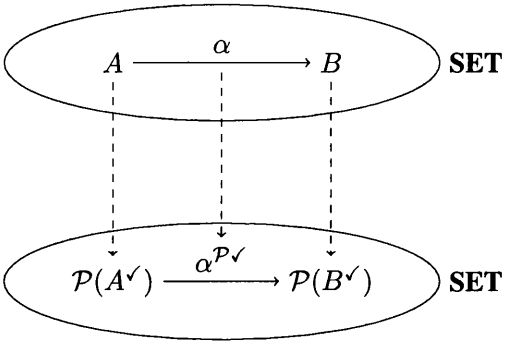



Figure 7.2: Illustration of the functor $\mathcal{P}\checkmark$.

$$\alpha^{\checkmark} : \begin{array}{lcl} A^{\checkmark} & \to & B^{\checkmark} \\ x & \mapsto & \begin{cases} \alpha(x) & \text{if } x \in A \\ \checkmark & \text{if } x = \checkmark \end{cases} \end{array}$$

- To extend it to strings with the termination symbol $\checkmark$:

$$\alpha^{*\checkmark} : \begin{array}{lcl} A^{*\checkmark} & \to & B^{*\checkmark} \\ s & \mapsto & \alpha^*(s) \\ s^\frown\langle\checkmark\rangle & \mapsto & \alpha^*(s)^\frown\langle\checkmark\rangle \end{array}$$

  where $s \in A^*$.

- To extend it to the power domain:

$$\alpha^{\mathcal{P}\checkmark} : \begin{array}{lcl} \mathcal{P}(A^{\checkmark}) & \to & \mathcal{P}(B^{\checkmark}) \\ X & \mapsto & \{\alpha^{\checkmark}(x) \mid x \in X\} \end{array}$$

These four basic translations (including $\alpha^*$ introduced as the component-wise extension of $\alpha$ to finite strings in Section 4.2.1) form functors from **SET** to **SET** (they can form more specific functors, but this is not of interest to us here), for instance:

$$\mathcal{P}\checkmark : \begin{array}{lcl} \textbf{SET} & \to & \textbf{SET} \\ A & \mapsto & \mathcal{P}(A^{\checkmark}) \\ \alpha : A \to B & \mapsto & \alpha^{\mathcal{P}\checkmark} : \mathcal{P}(A^{\checkmark}) \to \mathcal{P}(B^{\checkmark}) \end{array}$$

Figure 7.2 illustrates this construction for the functor $\alpha^{\mathcal{P}\checkmark}$.

**Lemma 7.1** The above basic translations (including $\alpha^*$) form valid functors from **SET** to **SET** and preserve monomorphisms (i.e., the resulting lifted functions are injective when the underlying translation is injective). In particular they are compatible with function composition:

- $\alpha_2^{\checkmark} \circ \alpha_1^{\checkmark} = (\alpha_2 \circ \alpha_1)^{\checkmark}$,
- $\alpha_2^* \circ \alpha_1^* = (\alpha_2 \circ \alpha_1)^*$,
- $\alpha_2^{*\checkmark} \circ \alpha_1^{*\checkmark} = (\alpha_2 \circ \alpha_1)^{*\checkmark}$,
- $\alpha_2^{\mathcal{P}\checkmark} \circ \alpha_1^{\mathcal{P}\checkmark} = (\alpha_2 \circ \alpha_1)^{\mathcal{P}\checkmark}$,

and $*\checkmark$ preserves prefixes:

- $s \leqslant t \implies \alpha^{*\checkmark}(s) \leqslant \alpha^{*\checkmark}(t)$.

where we use the notation $s \leqslant t$ for the string $s$ being a prefix of the string $t$.

*Proof.* It is straightforward to form the functors and prove that the functors are valid. The proof follows directly from the definitions of the functions above. Injectivity follows directly from the definitions of the functions as $\alpha$ is applied point wise. □

Here, we have formed four basic liftings (including $\alpha^*$) which will form the basis for the lifting of alphabet translations to the CSP domain translations presented in Section 7.1.2 and Section 7.1.3.

### 7.1.2 Covariant CSP Domain Translations

We now lift the basic translations from the previous sub-section to the CSP semantical domains: the Traces semantics $\mathcal{T}$, the Failures/Divergences semantics $\mathcal{N}$ and the Stable-Failures semantics $\mathcal{F}$. Kahsai presented a way to do this in [Kah10], however his definitions are not suitable for our institutional approach as they fail to preserve composition. Thus, we have adapted his definitions for our CSP-CASL institutions.

Here, we construct, what we call, the *covariant domain translations* which translate CSP domains in the same direction as the underlying alphabet translation. More precisely, we construct a covariant functor from the category where objects are alphabets and morphisms are alphabet translations to the category where objects are CSP domains and morphisms are CSP domain translations, that is, from **SET** to **SET** (see Figure 7.1).

We now define the lifting of alphabet translations to CSP domain translations for each of the CSP semantics.[1]

- For the CSP Traces semantics $\mathcal{T}$:

$$\begin{array}{lrcl} \alpha^{\mathcal{T}}: & \mathcal{T}(A) & \to & \mathcal{T}(B) \\ & T & \mapsto & \{\alpha^{*\checkmark}(s) \mid s \in T\} \end{array}$$

- For the CSP Failures/Divergences semantics $\mathcal{N}$:

$$\begin{array}{lrcl} \alpha^{\mathcal{N}}: & \mathcal{N}(A) & \to & \mathcal{N}(B) \\ & (F, D) & \mapsto & (\{(s', X') \in B^{*\checkmark} \times \mathcal{P}(B^{\checkmark}) \mid \exists (s, X) \in F \\ & & & \bullet\ \alpha^{*\checkmark}(s) = s' \wedge \forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha^{\checkmark})^{-}(x') \subseteq X\} \\ & & & \cup \{(s', X') \in B^{*\checkmark} \times \mathcal{P}(B^{\checkmark}) \mid s' \in \alpha^{\mathcal{ND}}(D)\}, \alpha^{\mathcal{ND}}(D)) \\ \alpha^{\mathcal{ND}}: & A^{*\checkmark} & \to & B^{*\checkmark} \\ & D & \mapsto & \{s' \in B^{*\checkmark} \mid \\ & & & \exists s \in D \bullet \exists t' \in B^{*\checkmark} \bullet \alpha^{*\checkmark}(s) \frown t' = s' \\ & & & \wedge \text{ if } s \text{ ends in } \checkmark \text{ then } t' = \langle\rangle\}) \end{array}$$

[1] Throughout use the letter $F$ as a variable to represent failures in both the Failures/Divergences semantics and the Stable-Failures semantics, however, they represent different failure sets. While $F$ in the Failures/Divergences semantics represents all failures, in the Stable-Failures semantics it represents only the stable failures.

- For the CSP Stable-Failures semantics $\mathcal{F}$:

$$\begin{array}{rcl} \alpha^{\mathcal{F}} : \mathcal{F}(A) & \to & \mathcal{F}(B) \\ (T, F) & \mapsto & (\alpha^{\mathcal{T}}(T), \{(s', X') \in B^{*\checkmark} \times \mathcal{P}(B^{\checkmark}) \mid \exists (s, X) \in F \\ & & \bullet\ \alpha^{*\checkmark}(s) = s' \wedge \forall x' \in B^{\checkmark} \bullet x' \in X' \Longrightarrow (\alpha^{\checkmark})^{-}(x') \subseteq X\}) \end{array}$$

where $(\alpha^{\checkmark})^{-}(x') := \{x \in A^{\checkmark} \mid \alpha^{\checkmark}(x) = x'\}$ (i.e., the inverse image of $x'$ under $(\alpha^{\checkmark})$) and $\mathcal{T}(\_), \mathcal{N}(\_)$ and $\mathcal{F}(\_)$ are the CSP semantical domains as defined in Section 2.4.

The Traces translation is straightforward where we just translate each trace in turn over $\alpha$. The Failures/Divergences and Stable-Failures translations require some extra elements to be added in order to preserve the healthiness conditions (see Section 2.4). In both cases failures need to be added that extend the refusal sets for each subset of elements outside of the image of $A$ in order to satisfy the CSP domain condition *F3*. In the case of divergences we must add all possible extensions of divergences that continue outside the image of $A$. This is to preserve condition *D1* which states if $s$ a divergence then all possible extensions over the full alphabet are also divergences. To satisfy condition *D2* of the Failures/Divergences semantics we must add all divergences as failures in the Failures/Divergences translation. It is for these reasons that the translations seem at first more complex than they need to be.

We now show via an example how the Stable-Failures covariant translation can increase the size of the refusal sets.

**Example 7.2** Let $A = \{a\}$ and $B = \{a', b'\}$ be alphabets. Let $\alpha : A \to B$ be an alphabet translation such that $\alpha(a) = a'$.

Let $(T, F) \in \mathcal{F}(A)$ such that

$$\begin{array}{rcl} T & = & \{\langle\rangle, \langle a\rangle\} \\ F & = & \{(\langle\rangle, \{\}), (\langle\rangle, \{\checkmark\}), (\langle a\rangle, \{\}), (\langle a\rangle, \{a\}), (\langle a\rangle, \{\checkmark\}), (\langle a\rangle, \{a, \checkmark\})\} \end{array}$$

This denotation represents the process $a \to$ STOP.

By applying the covariant domain translation $\alpha^{\mathcal{F}}$ we get $\alpha^{\mathcal{F}}(T, F) = (T', F')$ where

$$\begin{array}{rcl} T' & = & \{\langle\rangle, \langle a'\rangle\} \\ F' & = & \{(\langle\rangle, \{\}), (\langle\rangle, \{\checkmark\}), (\langle a'\rangle, \{\}), (\langle a'\rangle, \{a'\}), (\langle a'\rangle, \{\checkmark\}), (\langle a'\rangle, \{a', \checkmark\}), \\ & & (\langle\rangle, \{b'\}), (\langle\rangle, \{b', \checkmark\}), (\langle a'\rangle, \{b'\}), (\langle a'\rangle, \{a', b'\}), (\langle a'\rangle, \{b', \checkmark\}), (\langle a'\rangle, \\ & & \{a', b', \checkmark\})\} \end{array}$$

This denotation represents the process $a' \to$ STOP.

The interesting aspect here is the increase in the size of the refusal sets, for instance, the failure $(\langle a\rangle, \{a, \checkmark\})$ produces the following failures in the translation $(\langle a'\rangle, \{a', \checkmark\})$ and $(\langle a'\rangle, \{a', b', \checkmark\})$. In fact, in this example, each failure in $F$ produces two failures in the covariant translation $F'$.

Lemma 7.5 checks that our definitions are reasonable and do not violate the CSP domain conditions. However, in order to prove that lemma we first establish two additional lemmas. The first regards the preservation of prefix closure of $\alpha^{*\checkmark}$, while the second concerns the preservation of full refusal sets for the covariant domain translations.

**Lemma 7.3** ($\alpha^{*\checkmark}$ preserves prefix closure) Let $\alpha : A \to B$ be an alphabet translation and $t' \in B^{*\checkmark}$ be a trace such that $\alpha^{*\checkmark}(t) = t'$ for some trace $t \in A^{*\checkmark}$. If $s' \in B^{*\checkmark}$ is a trace such that $s' \leqslant t'$ then there exists trace $s \in A^{*\checkmark}$ such that $s \leqslant t$ and $\alpha^{*\checkmark}(s) = s'$.

*Proof.* This proof is by induction on the length of the trace $s'$ and is straightforward. □

**Lemma 7.4** (Covariant domain translations preserve full refusals) Let $\alpha : A \to B$ be an alphabet translation, then the following hold for the Failures/Divergences semantics $\mathcal{N}$:

- if $(s, X) \in F$ for all $X \subseteq A$ then $(\alpha^{*\checkmark}(s), X') \in F'$ for all $X' \subseteq B$, and
- if $(s, X) \in F$ for all $X \subseteq A^{\checkmark}$ then $(\alpha^{*\checkmark}(s), X') \in F'$ for all $X' \subseteq B^{\checkmark}$

where $(F, D) \in \mathcal{N}(A)$ and $\alpha^{\mathcal{N}}(F, D) = (F', D')$. Furthermore, the following hold for the Stable-Failures semantics $\mathcal{F}$:

- if $(s, X) \in F$ for all $X \subseteq A$ then $(\alpha^{*\checkmark}(s), X') \in F'$ for all $X' \subseteq B$, and
- if $(s, X) \in F$ for all $X \subseteq A^{\checkmark}$ then $(\alpha^{*\checkmark}(s), X') \in F'$ for all $X' \subseteq B^{\checkmark}$

where $(T, F) \in \mathcal{F}(A)$ and $\alpha^{\mathcal{F}}(T, F) = (T', F')$.

*Proof.* We prove only the first point for the Failures/Divergences semantics $\mathcal{N}$. The second point and the proof for the Stable-Failures semantics $\mathcal{F}$ are similar.

Assume $(s, X) \in F$ for all $X \subseteq A$. Let $X' \subseteq B$, we show $(s', X') \in F'$. Choose $X = \{x \in A \mid \alpha(x) \in X'\}$. Clearly, $X \subseteq A$. By construction we know $\forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha^{\checkmark})^{-}(x') \subseteq X$, thus $(s', X') \in F'$. □

We are now ready to prove that our covariant domain translations are reasonable and do not violate the CSP domain conditions from Section 2.4.

**Lemma 7.5** For all alphabet translations $\alpha : A \to B$, the covariant domain translation $\alpha^{\mathcal{D}}$ is healthy, that is, $\alpha^{\mathcal{D}} : \mathcal{D}(A) \to \mathcal{D}(B)$ for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

*Proof.* We prove that the covariant domain translations $\alpha^{\mathcal{T}}$, $\alpha^{\mathcal{N}}$, and $\alpha^{\mathcal{F}}$ preserve the healthiness conditions of the Traces semantics $\mathcal{T}$, the Failures/Divergences semantics $\mathcal{N}$, and the Stable-Failures semantics $\mathcal{F}$, respectively.

**Traces semantics $\mathcal{T}$** We show that $\alpha^{\mathcal{T}}$ preserves the condition *T1*. Let $T \in \mathcal{T}(A)$ and $T' = \alpha^{\mathcal{T}}(T)$.

**T1** We show that $T'$ is healthy, i.e., non-empty and prefix closed. As $\alpha^{*\checkmark}$ is total and $T$ is non-empty, it follows that $T'$ is also non-empty.

Let $t' \in T'$ and $s' \leqslant t'$. As $t' \in T'$ there exists $t \in T$ such that $\alpha^{*\checkmark}(t) = t'$. By Lemma 7.3 we know there exists trace $s \in A^{*\checkmark}$ such that $\alpha^{*\checkmark}(s) = s'$ and $s \leqslant t$. As $T$ is prefix closed we know $s \in T$, thus $s' \in T'$.

**Stable-Failures semantics $\mathcal{F}$** We show that $\alpha^{\mathcal{F}}$ preserves the conditions *T1*, *T2*, *T3*, *F2*, *F3*, and *F4*. Let $(T, F) \in \mathcal{F}(A)$ and $(T', F') = \alpha^{\mathcal{F}}(T, F)$.

**T1** Identical to the traces proof of condition *T1*.

**T2** Let $(s', X') \in F'$, we show $s' \in T'$. As $(s', X') \in F'$ then we know there exists $(s, X) \in F$ such that $\alpha^{*\checkmark}(s) = s'$. As $(T, F)$ is healthy (and satisfies condition *T2*), we know that $s \in T$ and thus $\alpha^{*\checkmark}(s) \in T'$, hence $s' \in T'$.

**T3** Let $s' \frown \langle\checkmark\rangle \in T'$, we show $(s' \frown \langle\checkmark\rangle, X') \in F'$ for all $X' \subseteq B^{\checkmark}$. We know there exists $s \frown \langle\checkmark\rangle \in T$ such that $\alpha^{*\checkmark}(s \frown \langle\checkmark\rangle) = s' \frown \langle\checkmark\rangle$. As $(T, F)$ is healthy (and satisfies condition *T3*), we know that $(s \frown \langle\checkmark\rangle, X) \in F$ for all $X \subseteq A^{\checkmark}$. By Lemma 7.4 we know $(s', X') \in F'$ for all $X' \subseteq B^{\checkmark}$.

**F2** Let $(s', X') \in F'$ and $Y' \subseteq X'$, then we must show $(s', Y') \in F'$. As $(s', X') \in F'$, there exists $(s, X) \in F$ such that $\alpha^{*\checkmark}(s) = s'$ and $\forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha^{\checkmark})^{-}(x') \subseteq X$. Define $Y = \{y \in A^{\checkmark} \mid \alpha^{\checkmark}(y) \in Y'\}$. Let $y \in Y$, then $\alpha^{\checkmark}(y) \in Y' \subseteq X'$, thus $(\alpha^{\checkmark})^{-}(\alpha^{\checkmark}(y)) \subseteq X$. As $y \in (\alpha^{\checkmark})^{-}(\alpha^{\checkmark}(y))$, we obtain $y \in X$, thus $Y \subset X$. As $(T, F)$ is healthy (and satisfies condition *F2*), we know that $(s, Y) \in F$. Thus, by construction we have $(s', Y') \in F'$.

**F3** Let $(s', X') \in F'$ and $Y' \subseteq B^{\checkmark}$ such that $\forall y' \in Y' \bullet s' \frown \langle y'\rangle \notin T'$, we show $(s', X' \cup Y') \in F'$. As $(s', X') \in F'$ there exists $(s, X) \in F$ such that $\alpha^{*\checkmark}(s) = s'$ and $\forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha^{\checkmark})^{-}(x') \subseteq X$. We can partition $Y'$ into $Y'_R$ containing the reachable elements from $A^{\checkmark}$ and $Y'_{NR}$ containing the non-reachable elements from $A^{\checkmark}$, i.e., let $Y'_R = \{y' \in Y' \mid \exists y \in A^{\checkmark} \bullet \alpha^{\checkmark}(y) = y'\}$ and $Y'_{NR} = \{y' \in Y' \mid \neg(\exists y \in A^{\checkmark} \bullet \alpha^{\checkmark}(y) = y')\}$. We can now construct the inverse image of $Y'_R$ over $\alpha^{\checkmark}$, i.e., let $Y_R := \{y \in A^{\checkmark} \mid \alpha^{\checkmark}(y) \in Y'_R\}$. As $Y'_R \subseteq Y'$, we also know $\forall y' \in Y'_R \bullet s' \frown \langle y'\rangle \notin T'$.

Now assume $\neg(\forall y \in Y_R \bullet s \frown \langle y\rangle \notin T)$. Then we know there exists $y \in Y_R$ such that $s \frown \langle y\rangle \in T$. By construction of $T'$ we have $\alpha^{*\checkmark}(s \frown \langle y\rangle) \in T'$, i.e., $s' \frown \langle\alpha^{\checkmark}(y)\rangle \in T'$. However, as $y \in Y_R$ we know that $\alpha^{\checkmark}(y) \in Y'_R$, and thus $s' \frown \langle\alpha^{\checkmark}(y)\rangle \notin T'$. Thus we have a contradiction and are forced to conclude $\forall y \in Y_R \bullet s \frown \langle y\rangle \notin T$. With this fact and the fact that $(T, F)$ is healthy (and satisfies condition *F3*), we know $(s, X \cup Y_R) \in F$. Finally, by construction of $(T', F')$ we can conclude $(s', X' \cup Y') \in F'$ by showing $\forall x' \in B^{\checkmark} \bullet x' \in X' \cup Y'_R \cup Y'_{NR} \implies (\alpha^{\checkmark})^{-}(x') \subseteq X \cup Y_R$. To this end, let $x' \in X' \cup Y'_R \cup Y'_{NR}$. We must show $(\alpha^{\checkmark})^{-}(x') \subseteq X \cup Y_R$. Let $x \in (\alpha^{\checkmark})^{-}(x')$, we show $x \in X \cup Y_R$. We know $\alpha^{\checkmark}(x) = x'$. We now make a case distinction on the source of $x'$.

**Case** $x' \in X'$ then $(\alpha^{\checkmark})^{-}(x') \subseteq X$ and thus $x \in X \cup Y_R$.

**Case** $x' \in Y'_R$ then $x \in Y_R$ by construction, thus $x \in X \cup Y_R$.

**Case** $x' \in Y'_{NR}$ then $\neg(\exists y \in A^{\checkmark} \bullet \alpha^{\checkmark}(y) = x')$. However, as we have $\alpha^{\checkmark}(x) = x'$, we have a contradiction and conclude that this case is impossible.

**F4** Let $s' \frown \langle\checkmark\rangle \in T'$, then we show $(s', B) \in F'$. We know there exists $s \frown \langle\checkmark\rangle \in T$ such that $\alpha^{*\checkmark}(s \frown \langle\checkmark\rangle) = s' \frown \langle\checkmark\rangle$. As $(T, F)$ is healthy (and satisfies conditions *F2* and *F4*), we know $(s, X) \in F$ for all $X \subseteq A$. Thus by Lemma 7.4 we know $(s', B) \in F'$.

**Failures/Divergences semantics $\mathcal{N}$** We show that $\alpha^{\mathcal{N}}$ preserves the conditions *F1*, *F2*, *F3*, *F4*, *D1*, *D2*, and *D3*. Let $(F, D) \in \mathcal{N}(A)$ and $(F', D') = \alpha^{\mathcal{N}}(F, D)$.

**F1** As $\alpha^{*\checkmark}$ is total and $tr_\perp(F, D)$ is non-empty, non-emptiness is preserved.

Let $t' \in tr_\perp(F', D')$, let $s' \leqslant t'$. We must show $s' \in tr_\perp(F', D')$. As $t'$ is a trace in $tr_\perp(F', D')$ we know $(t', \emptyset) \in F'$. This can happen in two ways:

**Case 1** The failure originates from $F$, that is, there exists $(t, Y) \in F$ such that $\alpha^{*\checkmark}(t) = t'$. As $(F, D)$ is healthy (by condition *F2*), we know $(t, \emptyset) \in F$, thus $t \in tr_\perp(F, D)$. By Lemma 7.3 we know there exists trace $s$ such that $\alpha^{*\checkmark}(s) = s'$ and $s \leqslant t$. As $tr_\perp(F, D)$ is prefix closed we have $s \in tr_\perp(F, D)$, thus $(s, \emptyset) \in F$. Applying $\alpha^{\mathcal{N}}$ we obtain $(s', \emptyset) \in F'$, hence $s' \in tr_\perp(F', D')$.

**Case 2** The trace $t'$ is a divergence, that is, $t' \in D'$. In this case there exists $t \in D$ and $u' \in B^{*\checkmark}$ such that $t' = \alpha^{*\checkmark}(t) \frown u'$. We consider two situations concerning how the traces $s'$, with $s' \leqslant t'$, and $\alpha^{*\checkmark}(t)$ are related.

**If** $s' \leqslant \alpha^{*\checkmark}(t)$ then by Lemma 7.3 we know there exists trace $s$ such that $\alpha^{*\checkmark}(s) = s'$ and $s \leqslant t$. As $t \in D$ we know $(t, \emptyset) \in F$ (by condition *D2*), thus $t \in tr_\perp(F, D)$. By prefix closure of $tr_\perp(F, D)$ we know $s \in tr_\perp(F, D)$, thus $(s, \emptyset) \in F$ and by definition of $\alpha^{\mathcal{N}}$ we know $(s', \emptyset) \in F'$. Hence $s' \in tr_\perp(F', D')$ by construction.

**Otherwise if** $s' \not\leqslant \alpha^{*\checkmark}(t)$, as we know $s' \leqslant \alpha^{*\checkmark}(t) \frown u'$, we can conclude there exists $v' \in B^{*\checkmark}$ such that $s' = \alpha^{*\checkmark}(t) \frown v'$. Thus $s' \in D'$ by construction, thus $(s', \emptyset) \in F'$, hence $s' \in tr_\perp(F', D')$.

**F2** Identical to the stable-failures proof of condition *F2* except for the extra failures we add from the divergences. As all possible refusal sets for each divergence are added as a failure, condition *F2* is preserved.

**F3** Identical to the stable-failures proof of condition *F3* except for the extra failures we add from the divergences. As all traces are possible after a divergence, there are no "refused" initials events.

**F4** Let $s' \frown \langle\checkmark\rangle$ in $tr_\perp(F', D')$, we show $(s', B) \in F'$. We know $(s' \frown \langle\checkmark\rangle, \emptyset) \in F'$. This can happen in two cases:

**Case 1** The failure originates from $F$, that is, there exists $(s \frown \langle\checkmark\rangle, X) \in F$ such that $\alpha^{*\checkmark}(s \frown \langle\checkmark\rangle) = s' \frown \langle\checkmark\rangle$. As $(F, D)$ is healthy (by condition *F2*), we know $(s \frown \langle\checkmark\rangle, \emptyset) \in F$, thus $s \frown \langle\checkmark\rangle \in tr_\perp(F, D)$. By conditions *F2* and *F4* we know $(s, Y) \in F$ for all $Y \subseteq A$. Finally by Lemma 7.4 we have $(s', B) \in F'$.

**Case 2** The trace is a divergence, that is, $s' \frown \langle\checkmark\rangle \in D'$. By the definition of $\alpha^{\mathcal{N}}$, we know there exists $s \in D$ and $t' \in B^{*\checkmark}$ such that $\alpha^{*\checkmark}(s) \frown t' = s' \frown \langle\checkmark\rangle$ and if $s$ ends in $\checkmark$ then $t' = \langle\rangle$. We now make a case distinction on $s$.

**If** $s$ ends in $\checkmark$ then $t' = \langle\rangle$ and $\alpha^{*\checkmark}(s) = s' \frown \langle\checkmark\rangle$. Let $u \frown \langle\checkmark\rangle = s$ (i.e., s without $\checkmark$). By condition *D3* we know $u \in D$. Thus, as $\alpha^{*\checkmark}(u) = s'$, we know $s' \in D'$ by construction. Hence $(s', B) \in F'$.

**Otherwise if** $s$ does not end in $\checkmark$ then $t'$ must end in $\checkmark$. Let $u' \frown \langle\checkmark\rangle = t'$ (i.e., $t'$ without $\checkmark$), then $\alpha^{*\checkmark}(s) \frown u' = s'$. Thus $s' \in D'$ by construction. Hence $(s', B) \in F'$.

**D1** Let $s' \in D' \cap B^*$ and $t' \in B^{*\checkmark}$, we show $s' \frown t' \in D'$. We know there exists $s \in D$ and $u' \in B^{*\checkmark}$ such that $s' = \alpha^{*\checkmark}(s) \frown u'$ (where $u'$ does not end in $\checkmark$ as $s'$ does not). Now $s' \frown t' = \alpha^{*\checkmark}(s) \frown u' \frown t'$ which is in $D'$ by construction.

**D2** Let $s' \in D'$ and $X' \subseteq B^{\checkmark}$. Then $(s', X') \in F'$ by definition of $\alpha^{\mathcal{N}}$.

**D3** Let $s' \frown \langle\checkmark\rangle \in D'$, we show $s' \in D'$. There exists $s \in D$ and $t' \in B^{*\checkmark}$ such that $\alpha^{*\checkmark}(s) \frown t' = s' \frown \langle\checkmark\rangle$ and if $s$ ends in $\checkmark$ then $t' = \langle\rangle$.

**Case 1** The trace $s$ ends in $\checkmark$. Then $t' = \langle\rangle$ and $\alpha^{*\checkmark}(s) = s' \frown \langle\checkmark\rangle$. Let $u \frown \langle\checkmark\rangle = s$ (i.e., let $u$ be $s$ without $\checkmark$). As $(F, D)$ is healthy (and satisfies condition *D3*) and $s \in D$, we know $u \in D$. As $\alpha^{*\checkmark}(u) = s'$, then $s' \in D'$ by construction.

**Case 2** The trace $s$ does not end in $\checkmark$, then the extension $t'$ must be non-empty and end in $\checkmark$. As we take all possible extensions $t'$, then one such extension would make true $\alpha^{*\checkmark}(s) \frown t' = s'$ (i.e, with no $\checkmark$). Therefore $s' \in D'$ by construction. □

Similar to how the basic translations in Section 7.1.1 form functors, the covariant domain translation functions above form the following functors:

- Traces model $\mathcal{T}$:

$$\begin{array}{lllll} \mathcal{T}: & \textbf{SET} & \to & \textbf{SET} \\ & A & \mapsto & \mathcal{T}(A) \\ & \alpha : A \to B & \mapsto & \alpha^{\mathcal{T}} : \mathcal{T}(A) \to \mathcal{T}(B) \end{array}$$

- Failures-divergences model $\mathcal{N}$:

$$\begin{array}{lllll} \mathcal{N}: & \textbf{SET} & \to & \textbf{SET} \\ & A & \mapsto & \mathcal{N}(A) \\ & \alpha : A \to B & \mapsto & \alpha^{\mathcal{N}} : \mathcal{N}(A) \to \mathcal{N}(B) \end{array}$$

- Stable-failures model $\mathcal{F}$:

$$\begin{array}{lllll} \mathcal{F}: & \textbf{SET} & \to & \textbf{SET} \\ & A & \mapsto & \mathcal{F}(A) \\ & \alpha : A \to B & \mapsto & \alpha^{\mathcal{F}} : \mathcal{F}(A) \to \mathcal{F}(B) \end{array}$$

We normally do not use the functor notation $\mathcal{D}(\alpha)$ for mapping morphisms and instead use the notation $\alpha^{\mathcal{D}}$ for the covariant CSP domain translation (for $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$) induced by the alphabet translation $\alpha$.

We now prove that these translations actually form valid functors. This is not obvious as the extra failures and divergences we add during the Failures/Divergences and Stable-Failures translations might disrupt functional composition.

**Lemma 7.6** $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ are valid functors, that is, they preserve identity morphisms and functional composition.

*Proof.* See Appendix A for proof. □

We now establish that the covariant domain translations preserve CSP refinements. This will be required when establishing the satisfaction conditions of the CSP-CASL institutions in Chapter 8.

**Lemma 7.7** The covariant domain translations $\alpha^{\mathcal{D}}$ are monotonic with respect to refinement for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$, that is, given an alphabet translation $\alpha : A \to B$ and semantic elements $d_1, d_2 \in \mathcal{D}(A)$, such that $d_1 \sqsubseteq_{\mathcal{D}} d_2$ then $\alpha^{\mathcal{D}}(d_1) \sqsubseteq_{\mathcal{D}} \alpha^{\mathcal{D}}(d_2)$.

*Proof.* This follows directly from the definitions of $\alpha^{\mathcal{D}}$. □

It is also the case that the covariant domain translations preserve injectivity. Again it is not obvious as the extra failures and divergences that we add during the translations might disrupt such a property.

**Lemma 7.8** If an alphabet translation $\alpha : A \to B$ is injective then the induced covariant domain translations $\alpha^{\mathcal{T}}$, $\alpha^{\mathcal{N}}$, and $\alpha^{\mathcal{F}}$ are also injective.

*Proof.* See Appendix A for proof. □

This concludes our construction of the covariant domain translations which are induced from alphabet translations. These allow us to lift alphabets translations to CSP domains. Next, we construct a similar lifting but in a contravariant style which maps against the underlying alphabet translation.

### 7.1.3 Contravariant CSP Domain Translations

Here, we present the lifting of *injective* alphabet translations to what we call *contravariant* CSP *domain translations*. This lifting was originally developed by Kahsai [Kah10]. We reformulate this lifting in a categorical setting which is required for our CSP-CASL institutional constructions (described in Chapter 8). These contravariant domain translations map in the opposite direction to the covariant domain translations, and therefore go against the underlying alphabet translation. More precisely, they are contravariant functors from **SET** to **SET**.

The contravariant translation will play a central role in CSP-CASL's model reducts in Section 8.1.3: when looking up an interpretation of a process name in a reduced CSP-CASL model, we take the process name's interpretation in the non-reduced model and translate the resulting denotation back to the correct alphabet using the contravariant translation.

We now discuss a short example which illustrates how such a contravariant translation will work with the CSP Traces semantics.

**Example 7.9** We illustrate the idea of translating CSP domains against an alphabet translation using the traditional example of a vending machine. Let $\alpha : Basic \to Enhanced$ be an injective alphabet translation from a basic alphabet that can only talk about *tea* (i.e., $Basic = \{tea\}$) to an enhanced alphabet that can talk about *tea* and *coffee* (i.e., $Enhanced = \{tea, coffee\}$) such that $\alpha(tea) := tea$. Both of these alphabets can be thought of as representing the actions of pressing various buttons on vending machines.

Let us focus on the CSP Traces semantics $\mathcal{T}$, and allow $T'$ to be some trace set over the enhanced alphabet. $T'$ can be seen as a set of use cases: each trace is a sequence of possible actions that can be carried out on the machine. The contravariant translations allow us to map this sequence back onto the basic machine, as far as we can.

For instance, a trace $\langle tea, tea, coffee, tea \rangle$ would be mapped back to the trace $\langle tea, tea \rangle$. We simply map the trace back against the alphabet translation as far as we can, until we find an action that cannot be represented in the basic system. When this happens we stop our translation of the trace. This indicates why we require injective alphabet translations: if the underlying alphabet translation was not injective, then we would not be able to map the action back across the translation in a unique way.

We stop the translation of a trace set once we encounter an action outside the basic alphabet because we are interpreting the trace set on the basic machine. While the traces remain within the basic alphabet, the behaviour remains the same. However, once the traces use actions not represented in the basic alphabet, then the behaviour remains the same only up to the occurrence of the first such action. After this action the traces of the basic machine cannot tell us anything about the behaviour of the exchanged machine.

Our definitions differ from the CSP hiding operator and both concepts of lazy abstraction and eager abstraction [Ros05]. The CSP hiding operator removes elements from traces. Lazy abstraction, which is defined in terms of hiding, and eager abstraction also change the trace and failure sets by hiding events. However, we do not want to modify the traces or failures via the translation as this would change the meaning of the process. Instead, we want to capture as much behaviour as possible with the smaller alphabet.

Kahsai et al. [KRS08] describe the same notion on the CSP Traces and Stable-Failures semantics. There it is used as a notion of enhancement of processes, guaranteeing preservation of behaviour up to the first communication that lies outside the original alphabet. Kahsai et al. use enhancement as a development notion, for the purpose of re-using test cases for CSP-CASL specifications.

We now define the contravariant domain translations, following the idea in the example above of stopping when we cannot translate backwards any further. This idea also works with failures and refusals: if we have an action that we can refuse in the larger alphabet and have no corresponding action in the smaller alphabet, then we simply forget about it. This makes sense as all actions outside the alphabet are not part of the behaviour of the process.

Given an injective alphabet translation $\alpha : A \rightarrow B$, we define the *contravariant domain translation* functions $\hat{\alpha}^{\mathcal{D}}$ to be:

- For the CSP Traces semantics $\mathcal{T}$:

$$\begin{array}{lrcl} \hat{\alpha}^{\mathcal{T}} : & \mathcal{F}(B) & \rightarrow & \mathcal{F}(A) \\ & T' & \mapsto & \{s \in A^{*\surd} \mid \alpha^{*\surd}(s) \in T'\} \end{array}$$

- For the CSP Failures/Divergences semantics $\mathcal{N}$:

$$\begin{array}{lrcl} \hat{\alpha}^{\mathcal{N}} : & \mathcal{N}(B) & \rightarrow & \mathcal{N}(A) \\ & (F', D') & \mapsto & (\{(s, X) \in A^{*\surd} \times \mathcal{P}(A^{\surd}) \mid \exists (s', X') \in F' \\ & & & \bullet\ \alpha^{*\surd}(s) = s' \wedge X' \cap \alpha^{\mathcal{P}\surd}(A^{\surd}) = \alpha^{\mathcal{P}\surd}(X)\}, \\ & & & \{s \in A^{*\surd} \mid \alpha^{*\surd}(s) \in D'\}) \end{array}$$

- For the CSP Stable-Failures semantics $\mathcal{F}$:

$$\begin{array}{lrl} \hat{\alpha}^{\mathcal{F}} : & \mathcal{F}(B) & \to \mathcal{F}(A) \\ & (T', F') & \mapsto (\hat{\alpha}^{\mathcal{T}}(T'), \{(s, X) \in A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}) \mid \exists (s', X') \in F' \\ & & \bullet\ \alpha^{*\checkmark}(s) = s' \wedge X' \cap \alpha^{\mathcal{P}\checkmark}(A^{\checkmark}) = \alpha^{\mathcal{P}\checkmark}(X)\}) \end{array}$$

Traces and divergences are translated in an obvious way, while the failures have to have their refusal sets restricted to elements from the alphabet $A$. This is achieved by selecting only those refusal sets which after forward translation match refusal sets $F'$ but only after the refusal sets in $F'$ have been restricted to the reachable elements in $A$.

**Discussion:** The condition $X' \cap \alpha^{\mathcal{P}\checkmark}(A^{\checkmark}) = \alpha^{\mathcal{P}\checkmark}(X)$ actually coincides with our earlier condition $\forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha^{\checkmark})^{-}(x') \subseteq X$ (used in the covariant domain translations) when the underlying function is injective and the domain healthiness conditions are considered. We could replace the condition above in the contravariant translations with the condition in the covariant translation without causing any further changes in our constructions other than proof details. However, we choose to use Kahsai's original (and somewhat simpler) definitions. It is the healthiness conditions and the injectivity constraint that allow for a simple contravariant translation compared to the more complex covariant translations (particularly in the Failures/Divergences semantics).

We now present an example illustrating the Stable-Failures contravariant translation.

**Example 7.10** Let $A = \{a\}$ and $B = \{a', b'\}$ be alphabets. Let $\alpha : A \to B$ be an alphabet translation such that $\alpha(a) = a'$. As $\alpha$ is injective, we can form the contravariant domain translation $\hat{\alpha}^{\mathcal{F}}$.

Let $(T', F') \in \mathcal{F}(B)$ such that

$$\begin{array}{rcl} T' & = & \{\langle\rangle, \langle a'\rangle, \langle a', b'\rangle\} \\ F' & = & \{(\langle\rangle, \{\}), (\langle\rangle, \{b'\}), (\langle\rangle, \{\checkmark\}), (\langle\rangle, \{b', \checkmark\}), \\ & & \{(\langle a'\rangle, \{\}), (\langle a'\rangle, \{a'\}), (\langle a'\rangle, \{\checkmark\}), (\langle a'\rangle, \{a', \checkmark\}), \\ & & \{(\langle a', b'\rangle, \{\}), (\langle a', b'\rangle, \{a'\}), (\langle a', b'\rangle, \{b'\}), (\langle a', b'\rangle, \{\checkmark\}), \\ & & (\langle a', b'\rangle, \{a', b'\}), (\langle a', b'\rangle, \{a', \checkmark\}), (\langle a', b'\rangle, \{b', \checkmark\}), (\langle a', b'\rangle, \{a', b', \checkmark\})\} \end{array}$$

This denotation represents the process $a' \to b' \to$ STOP.

By applying the contravariant domain translation $\alpha^{\mathcal{F}}$ we get $\hat{\alpha}^{\mathcal{F}}(T', F') = (T, F)$ where

$$\begin{array}{rcl} T & = & \{\langle\rangle, \langle a'\rangle\} \\ F & = & \{(\langle\rangle, \{\}), (\langle\rangle, \{\checkmark\}), (\langle a\rangle, \{\}), (\langle a\rangle, \{a\}), (\langle a\rangle, \{\checkmark\}), (\langle a\rangle, \{a, \checkmark\}) \end{array}$$

This denotation represents the process $a \to$ STOP.

The interesting aspect here is the shrinking effect, for instance, the failures $(\langle\rangle, \{\checkmark\})$ and $(\langle\rangle, \{b', \checkmark\})$ in $F'$ both yield only the single failure $(\langle\rangle, \{\checkmark\})$ in $F$. These two failures are "collapsed" during the contravariant translation.

This makes sense in that we are trying to convert the sequence of interactions of pressing the $a'$ button, followed by the $b'$ button back across the alphabet translation and we end up with the sequence of only pressing the $a$ button as there was no corresponding button for $b'$ in the smaller alphabet.

**Example 7.11** If we now consider Example 7.10 above along with that of Example 7.2, we see that the covariant and contravariant translations are not the inverse of each other. If we start with the process $a' \to b' \to$ STOP and translate backwards using the contravariant translation, we end up with the process $a \to$ STOP. If we now go forwards with the covariant translation we do not get back to where we started with the process $a' \to b' \to$ STOP, but instead we get the process $a' \to$ STOP.

The following lemma (presented in [Kah10]) checks that our definitions are reasonable and do not violate the CSP domain conditions.

**Lemma 7.12** For all injective alphabet translations $\alpha : A \to B$, the contravariant domain translation $\hat{\alpha}^{\mathcal{D}}$ is healthy, that is, $\hat{\alpha}^{\mathcal{D}} : \mathcal{D}(B) \to \mathcal{D}(A)$ for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

*Proof.* See Appendix A for proof. □

Just like the covariant domain translations form functors, the contravariant domain translations form the following contravariant functors:[2]

- Traces semantics $\mathcal{T}$:

$$\begin{array}{lllll} \mathcal{T}^{op} : & \textbf{SET\_INJ} & \to & \textbf{SET} \\ & A & \mapsto & \mathcal{T}(A) \\ & \alpha : A \to B & \mapsto & \hat{\alpha}^{\mathcal{T}} : \mathcal{T}(B) \to \mathcal{T}(A) \end{array}$$

- Failures/Divergences semantics $\mathcal{N}$:

$$\begin{array}{lllll} \mathcal{N}^{op} : & \textbf{SET\_INJ} & \to & \textbf{SET} \\ & A & \mapsto & \mathcal{N}(A) \\ & \alpha : A \to B & \mapsto & \hat{\alpha}^{\mathcal{N}} : \mathcal{N}(B) \to \mathcal{N}(A) \end{array}$$

- Stable-Failures semantics $\mathcal{F}$:

$$\begin{array}{lllll} \mathcal{F}^{op} : & \textbf{SET\_INJ} & \to & \textbf{SET} \\ & A & \mapsto & \mathcal{F}(A) \\ & \alpha : A \to B & \mapsto & \hat{\alpha}^{\mathcal{F}} : \mathcal{F}(B) \to \mathcal{F}(A) \end{array}$$

The fact that these actually form functors is not obvious from the definitions. The contravariant definitions use set intersections to select refusal sets from failures which may hinder the composition of the translations.

**Lemma 7.13** $\mathcal{T}^{op}$, $\mathcal{N}^{op}$, and $\mathcal{F}^{op}$ are valid functors, that is, they preserve identity morphisms and functional composition.

*Proof.* See Appendix A for proof. □

Finally, we guarantee that the construction of the contravariant domain translations above preserve CSP refinements.

[2] We use **SET_INJ** to denote the category **SET** where we restrict the morphisms to injective functions.

**Lemma 7.14** The functions $\hat{\alpha}^{\mathcal{D}}$ are monotonic with respect to refinement for all CSP semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$, that is, given an injective alphabet translation $\alpha : A \to B$ and semantic elements $d'_1, d'_2 \in \mathcal{D}(B)$, such that $d'_1 \sqsubseteq_{\mathcal{D}} d'_2$ then $\hat{\alpha}^{\mathcal{D}}(d'_1) \sqsubseteq_{\mathcal{D}} \hat{\alpha}^{\mathcal{D}}(d'_2)$.

*Proof.* This follows directly from the definitions of $\hat{\alpha}^{\mathcal{D}}$. □

**Corollary 7.15** Thanks to the monotonicity of the domain translations (lemmas 7.7 and 7.14), both the covariant and contravariant domain translations preserve CSP refinement for the CSP Traces semantics, Failures/Divergences semantics and the Stable-Failures semantics.

### 7.1.4 Relationship Between Covariant and Contravariant Domain Translations

We now study how the covariant and contravariant domain translations interact. It turns out that going forwards then backwards has no effect, while the converse is not true in general.

These results give us confidence that our construction makes sense, as one would expect such results to hold.

**Lemma 7.16** Given an injective alphabet translation $\alpha : A \to B$ then $\hat{\alpha}^{\mathcal{D}} \circ \alpha^{\mathcal{D}}$ is the identity function. Note: We are not claiming $\alpha^{\mathcal{D}} \circ \hat{\alpha}^{\mathcal{D}}$ is the identity function, this is not true in general, see Example 7.11.

*Proof.* We prove this for each semantics $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ separately.

**Traces semantics** $\mathcal{T}$ This proof follows directly from definitions.

**Stable-Failures semantics** $\mathcal{F}$ Let $(T, F) \in \mathcal{F}(A)$, we show $\hat{\alpha}^{\mathcal{F}}(\alpha^{\mathcal{F}}(T, F)) = (T, F)$ by showing equality between each component. Equality between the traces component is the same as for the Traces model above. We now show $snd(\hat{\alpha}^{\mathcal{F}}(\alpha^{\mathcal{F}}(T, F))) = F$ by subset inclusion in both directions.

- **'$\subseteq$' direction** Let $(s, X) \in snd(\hat{\alpha}^{\mathcal{F}}(\alpha^{\mathcal{F}}(T, F)))$ then we know by definition that there exists $(s', X') \in snd(\alpha^{\mathcal{F}}(T, F))$ such that $\alpha^{*\checkmark}(s) = s'$ and $X' \cap \alpha^{\mathcal{P}\checkmark}(A^{\checkmark}) = \alpha^{\mathcal{P}\checkmark}(X)$. Furthermore we know there exists $(t, Y) \in F$ such that $\alpha^{*\checkmark}(t) = s'$ and $\forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha^{\checkmark})^{-}(x') \subseteq Y$. By the injectivity of $\alpha$ we know $s = t$ and $X \subseteq Y$. By the healthiness conditions of $(T, F)$ we know $(s, X) \in F$.
- **'$\supseteq$' direction** Let $(s, X) \in F$, then by construction we know $(\alpha^{*\checkmark}(s), \alpha^{\mathcal{P}\checkmark}(X)) \in snd(\alpha^{\mathcal{F}}(T, F))$. As $\alpha^{\mathcal{P}\checkmark}(X) \cap \alpha^{\mathcal{P}\checkmark}(A^{\checkmark}) = \alpha^{\mathcal{P}\checkmark}(X)$ we know by construction that $(s, X) \in snd(\hat{\alpha}^{\mathcal{F}}(\alpha^{\mathcal{F}}(T, F)))$.

**Failures/Divergences semantics** $\mathcal{N}$ Let $(F, D) \in \mathcal{N}(A)$, we show $\hat{\alpha}^{\mathcal{N}}(\alpha^{\mathcal{N}}(F, D)) = (F, D)$ by showing equality between each component by subset inclusion in both directions.

- **Divergences '$\subseteq$' direction** Let $s \in snd(\hat{\alpha}^{\mathcal{N}}(\alpha^{\mathcal{N}}(F, D)))$, then we know by definition that $\alpha^{*\checkmark}(s) \in snd(\alpha^{\mathcal{N}}(F, D))$. Furthermore we know there exists $u \in D$ and $t' \in B^{*\checkmark}$ such that $\alpha^{*\checkmark}(u) \frown t = \alpha^{*\checkmark}(s)$. From this we can deduce there must exist $t \in A^{*\checkmark}$ with $\alpha^{*\checkmark}(t) = t'$, thus by the healthiness condition of $(F, D)$ we

know $u \frown t \in D$. By the injectivity of $\alpha$ and the fact that $\alpha^{*\checkmark}(u \frown t) = \alpha^{*\checkmark}(s)$, we know $u \frown t = s$, thus $s \in D$.

**Divergences '$\supseteq$' direction** This proof follows directly from definitions.

**Failures '$\subseteq$' direction** Let $(s, X) \in \mathit{fst}(\hat{\alpha}^{\mathcal{N}}(\alpha^{\mathcal{N}}(F, D)))$, then we know by definition that there exists $(s', X') \in \mathit{fst}(\alpha^{\mathcal{N}}(F, D))$ such that $\alpha^{*\checkmark}(s) = s'$ and $X' \cap \alpha^{\mathcal{P}\checkmark}(A^{\checkmark}) = \alpha^{\mathcal{P}\checkmark}(X)$. The failure $(s', X')$ can originate from either a failure in $F$ or from a divergence.

The proof for the case where $(s', X')$ originates from a failure in $F$ is the same as in the Stable-Failures semantics for the '$\subseteq$' direction on the failures component. We now consider the case where $(s', X')$ originates from from a divergence, that is, $s' \in \mathit{snd}(\alpha^{\mathcal{N}}(F, D))$. In this case we know, by definition of $\hat{\alpha}^{\mathcal{N}}$, that $s \in \mathit{snd}(\hat{\alpha}^{\mathcal{N}}(\alpha^{\mathcal{N}}(F, D)))$. By the above proof that the divergences are equal we know $s \in D$. Furthermore, as we know $(F, D)$ is healthy we can conclude $(s, X) \in F$.

**Failures '$\supseteq$' direction** The proof for this case is the same as in the Stable-Failures semantics for the '$\supseteq$' direction on the failures component. □

The above lemma will be critical in establishing the CSP-CASL institutions in Chapter 8. Next, we introduce the notion of controlled traces. If we restrict the domains of the covariant and contravariant translations appropriately using this notion of controlled traces, we find that they are actually inverse to each other. However, first we need the notion of *controlled traces.*

We now form "projection" functions from CSP denotations of $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ into the Traces semantics. These functions "project" out of a CSP denotations the controlled traces. The controlled traces of a process $P$ are the traces of $P$ which are under its control, that is, they are not divergent.

- For the Traces semantics $\mathcal{T}$:

$$\begin{array}{lccc} cTr^{\mathcal{T}} : & \mathcal{T}(A) & \to & \mathcal{T}(A) \\ & T & \mapsto & T \end{array}$$

- For the Failures/Divergences semantics $\mathcal{N}$:

$$\begin{array}{lccc} cTr^{\mathcal{N}} : & \mathcal{N}(A) & \to & \mathcal{T}(A) \\ & (F, D) & \mapsto & \{s \mid s \in tr_{\perp}(F, D) \wedge \forall t \in A^{*\checkmark} \bullet t < s \implies t \notin D\} \end{array}$$

Here, we take all traces of $(F, D)$ which are not extensions of a divergence. If a trace $s$ is in $D$ then all extensions of $s$ are also in $D$ by healthiness condition *D1*. Thanks to the healthiness conditions *F2* and *D2* all these divergences and their extensions are within the failures and thus in $tr_{\perp}(F, D)$. We only wish to take the traces which are not an extension of a divergence (we do not mind if the traces are the start of a divergence), hence we use the strict prefix operator $<$.

- For the Stable-Failures semantics $\mathcal{F}$:

$$\begin{array}{lccc} cTr^{\mathcal{F}} : & \mathcal{F}(A) & \to & \mathcal{T}(A) \\ & (T, F) & \mapsto & T \end{array}$$

We usually drop the superscript $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ when the particular CSP semantics is clear from the context.

The next lemma considers what happens to the controlled traces under contravariant translation.

**Lemma 7.17** Let $\alpha : A \to B$ be an injective alphabet translation. Let $A' \subseteq A$ and $B' \subseteq B$ such that $B' \subseteq \alpha^{\mathcal{P}\checkmark}(A')$. If $d' \in \mathcal{D}(B)$ with $cTr^{\mathcal{D}}(d') \in \mathcal{T}(B')$ then $cTr^{\mathcal{D}}(\hat{\alpha}^{\mathcal{D}}(d')) \in \mathcal{T}(A')$.

*Proof.* We prove this for each CSP semantics $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ individually.

**Traces semantics** $\mathcal{T}$ This follows directly from definitions.

**Failures-divergences model** $\mathcal{N}$ Let $(F', D') \in \mathcal{N}(B)$ such that $cTr^{\mathcal{N}}(F', D') \in \mathcal{T}(B')$. We must show $cTr^{\mathcal{N}}(\hat{\alpha}^{\mathcal{N}}(F', D')) \in \mathcal{T}(A')$, that is, for all traces $s \in cTr^{\mathcal{N}}(\hat{\alpha}^{\mathcal{N}}(F', D'))$ that $s \in A'^{*\checkmark}$. Let $s \in cTr^{\mathcal{N}}(\hat{\alpha}^{\mathcal{N}}(F', D'))$. Then we know $s \in tr_{\perp}(\hat{\alpha}^{\mathcal{N}}(F', D'))$ and $\forall t \in A^{*\checkmark} \bullet t < s \implies t \notin snd(\hat{\alpha}^{\mathcal{N}}(F', D'))$. Furthermore, we know $(s, \emptyset) \in fst(\hat{\alpha}^{\mathcal{N}}(F', D'))$, therefore there exists $(s', X') \in F'$ such that $\alpha^{*\checkmark}(s) = s'$. From these facts we can deduce $s' \in cTr^{\mathcal{N}}(F', D') \in \mathcal{T}(B')$, thus $s' \in (B')^{*\checkmark}$. As $B' \subseteq \alpha^{\mathcal{P}\checkmark}(A')$, we know $s' \in (\alpha^{\mathcal{P}\checkmark}(A'))^{*\checkmark}$, thus $s \in (A')^{*\checkmark}$.

**Stable-Failures semantics** $\mathcal{F}$ As the controlled traces is the projection on the traces component, the proof is the same as for the Traces semantics. □

If we consider only CSP denotations which have their controlled traces in the translation of the original alphabet, then we find out that going backwards over the contravariant domain translation and then forwards over the covariant domain translation yields the original denotation, that is, in this restricted setting the covariant and contravariant domain translations are bijections.

**Lemma 7.18** Given an injective alphabet translation $\alpha : A \to B$ and a denotation $d'$ in $\mathcal{D}(B)$ such that $cTr^{\mathcal{D}}(d') \in \mathcal{T}(\alpha^{\mathcal{P}\checkmark}(A))$ then $(\alpha^{\mathcal{D}}(\hat{\alpha}^{\mathcal{D}}(d'))) = d'$ for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

*Proof.* This proof is straightforward and follows from definitions. □

This concludes our study of the relationship between the covariant and contravariant domain translations. These results will be used in Chapter 8 to establish the CSP-CASL institutions.

### 7.1.5 Top Elements

We now define a notion, related to controlled traces, that will be used later in this thesis during various proofs. We define the notion of the "top" element of each semantical domain (restricted to a subset of the alphabet) which is the least refined element, relative to the subset of the alphabet. The "top" element will have its controlled traces constrained to the subset.

Given an alphabet $A$ and a subset of the alphabet $C \subseteq A$, we define the "top" element $Top^{\mathcal{D}}_C$ as:

- $Top^{\mathcal{T}}_C := C^{*\checkmark}$,

- $Top_C^{\mathcal{N}} := (C^{*\checkmark}, \{(s, X) \in A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}) \mid s \in C^{*\checkmark}\})$, and
- $Top_C^{\mathcal{F}} := (A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}), A^{*\checkmark})$, that is, the denotation for the process DIV

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

**Lemma 7.19** Given an alphabet $A$ and a subset of the alphabet $C \subseteq A$, $Top_C^{\mathcal{D}}$ exhibits the following properties:

1. $Top_C^{\mathcal{D}} \in \mathcal{D}(A)$,
2. $cTr^{\mathcal{D}}(Top_C^{\mathcal{D}}) \in \mathcal{T}(C)$, and
3. for all $d \in \mathcal{D}(A)$ with $cTr^{\mathcal{D}}(d) \in \mathcal{T}(C)$ we have that $Top_C^{\mathcal{D}} \sqsubseteq_{\mathcal{D}} d$

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

*Proof.* We prove each property in turn for each semantics.

**Property 1** It is straight forward to show $Top_C^{\mathcal{D}} \in \mathcal{D}(A)$ for each semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

**Property 2** As $cTr$ is a simple projection function for the Traces and Stable-Failures semantics, we have $cTr(Top_C^{\mathcal{T}}) = cTr(Top_C^{\mathcal{F}}) = C^{*\checkmark} \in \mathcal{T}(C)$.

We now show $cTr(Top_C^{\mathcal{N}}) \in \mathcal{T}(C)$, that is, we must show for all traces $s \in cTr(Top_C^{\mathcal{N}})$ that $s \in C^{*\checkmark}$. Let $Top_C^{\mathcal{D}} = (F, D)$ and $s \in cTr(Top_C^{\mathcal{N}})$. We know $(s, \emptyset) \in F$ and $\forall t \in A^{*\checkmark} \bullet t < s \implies t \notin D$. Assume $s \neq \langle\rangle$ then we have $\langle\rangle < s$ and $\langle\rangle \in D$, which is a contradiction. Thus we know $s = \langle\rangle \in C^{*\checkmark}$.

**Property 3** It is straight forward to show that $Top_C^{\mathcal{D}} \in \mathcal{D}(A)$ refines to all other elements which have their controlled traces contained within $C$. □

### 7.1.6 Deadlock Proprieties of the Contravariant Translation

In order to perform deadlock analysis in Section 9.4 it is required that the contravariant Stable-Failures domain translation interacts nicely with refusal sets. The following lemma states that deadlocks are preserved after translation and also that there is a strong link with failures in the translation.

**Lemma 7.20** Given an injective alphabet translation $\alpha : A \to B$, and domain elements $(T, F) \in \mathcal{F}(A)$ and $(T', F') \in \mathcal{F}(B)$ such that $\hat{\alpha}^{\mathcal{F}}(T', F') = (T, F)$ then the following hold:

- $(s, X) \in F \iff (\alpha^{*\checkmark}(s), \alpha^{\mathcal{P}\checkmark}(X)) \in F'$.
- $(\alpha^{*\checkmark}(s), B^{\checkmark}) \in F' \implies (s, A^{\checkmark}) \in F$, that is, deadlocks are preserved during contravariant translation,
- Let $T' \in \mathcal{T}(\alpha^{\mathcal{P}\checkmark}(A))$. Then $(s, A^{\checkmark}) \in F \implies (\alpha^{*\checkmark}(s), B^{\checkmark}) \in F'$. That is, under the condition that the traces are in the image of $A$, deadlocks only occur after a contravariant translation when there was a deadlock in the source.

*Proof.* The first two implications follow directly from definitions and the Stable-Failures domain condition *F2*. We now establish the third.

Let $(s, A^{\checkmark}) \in F$, then we know $(\alpha^{*\checkmark}(s), \alpha^{\mathcal{P}\checkmark}(A^{\checkmark})) \in F'$. Let $Y' = B^{\checkmark} - \alpha^{\mathcal{P}\checkmark}(A^{\checkmark})$. Thanks to the assumption on the traces component we know $\forall x' \in Y' \bullet \alpha^{*\checkmark}(s)^\frown\langle x'\rangle \notin T'$. Thus, by healthiness condition *F3* we know that $(\alpha^{*\checkmark}(s), \alpha^{\mathcal{P}\checkmark}(A^{\checkmark}) \cup Y') \in F'$, that is, $(\alpha^{*\checkmark}(s), B^{\checkmark}) \in F'$. □

This concludes our presentation and discussions of the CSP domain translation functions that we require for constructing the CSP-CASL institution. Given an alphabet translation (i.e., a function) $\alpha : A \to B$, we can create the covariant domain translation functions $\alpha^{\mathcal{D}} : \mathcal{D}(A) \to \mathcal{D}(B)$. Furthermore, if $\alpha$ is injective then we can also create the contravariant domain translation functions $\hat{\alpha}^{\mathcal{D}} : \mathcal{D}(A) \to \mathcal{D}(B)$ for each of the CSP semantics, that is, $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$. These covariant and contravariant domain translations behave in a reasonable and expected way and exhibit all the necessary properties for us to establish our CSP-CASL institutions.

Next, we look at flattening CASL models to alphabets and transforming CASL model morphisms and model reducts to alphabet translations. The results of these transformations can then be lifted, as we have done in this section, to form CSP domains, covariant domain translations and contravariant domain translations.

## 7.2 Construction 2: Lifting Reducts and Flattening Many-Sorted Algebras

The ultimate goal of this chapter is to lift many-sorted algebras, morphisms between them and reducts to the CSP context. Construction 1 in Section 7.1 described how to lift alphabets and alphabet translations to CSP. We now describe how to flatten CASL models to alphabets and how to lift model morphisms and model reducts to alphabet translations. The constructions described here can then be composed with those from Construction 1 to achieve our ultimate goal of lifting the notions of CASL models, model morphisms and reducts to the CSP context.

To this end, Section 7.2.1 and Section 7.2.2 construct a functor $Alph : \mathbf{mod}(\Sigma) \to \mathbf{SET}$ which maps many-sorted-$\Sigma$-algebras and morphisms to alphabets and alphabet translations, respectively. This is illustrated in Figure 7.3. Section 7.2.3 then lifts, in a similar manner, model reducts (which are induced by signature morphisms) to also form alphabet translations.

### 7.2.1 Alphabet Construction

The alphabet construction allows us to transform a CASL model, or more precisely, its carrier sets into a flat set whilst taking the sub-sort structure into account. We can then use this alphabet with CSP to create CSP domains over the alphabet.

Before we give the formal definition of this, we introduce some useful shorthand notation. Given a *ResSubPCFOL*$^{=}$ model $M$, we use the shorthand $M_\perp$ for the totalised version of $M$, that is, carrier sets include a bottom element $M_\perp(s) = M(s) \cup \{\perp_s\}$ and the interpretation of function and predicate symbols is strictly extended (see the data-logic, Section 4.5). We also sometimes write $M_{\perp_s}$ in place of $M_\perp(s)$ for the carrier set of sort $s$ in the totalised version

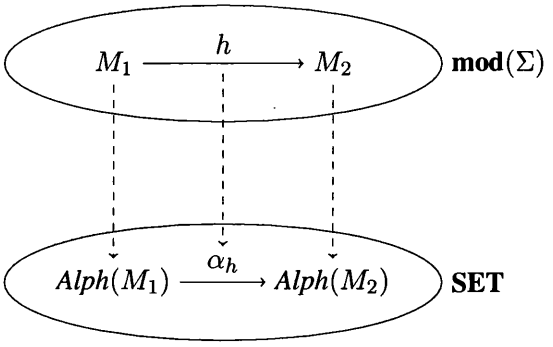



Figure 7.3: Lifting many-sorted-$\Sigma$-algebras and morphisms to alphabets and alphabet translations. Here $\alpha_h$ is shorthand for $Alph(h)$.

of the model $M$. Given a sort symbol $s$, a *ResSubPCFOL*$^=$ model $M$, and $x \in M_\perp(s)$ we write $\overline{x}^s_M$ to denote the alphabet element $[(s,x)]_{\sim_M}$ (presented shortly). Furthermore, we lift this notation to sorts, namely $\overline{s}_M = \{\overline{x}^s_M \mid x \in M_\perp(s)\} \subseteq Alph(M)$ for the set of communications that can arise from the sort $s$ in the model $M$. Finally, given a set of sorts $X$, we write $\overline{X}_M = \bigcup_{s \in X} \overline{s}_M$. We drop the subscripts $M$ and superscripts $s$ when clear from the context.

Given a *ResSubPCFOL*$^=$ model $M$ (see Section 4.4) for signature $\Sigma$ with sort set $S$ we define the *alphabet of communications* $Alph(M)$ (as is done in [Rog06]) as:

$$Alph(M) = (\biguplus_{s \in S}(M_s \cup \{\perp\}))/_{\sim_M}$$

where $(s,x) \sim_M (t,y)$ if and only if either

- $x = y = \perp$ and
- there exists $u \in S$ such that $s \leq u$ and $t \leq u$,

or

- $x \neq \perp, y \neq \perp$,
- there exists $u \in S$ such that $s \leq u$ and $t \leq u$, and
- for all $u \in S$ such that $s \leq u$ and $t \leq u$ the following holds:

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

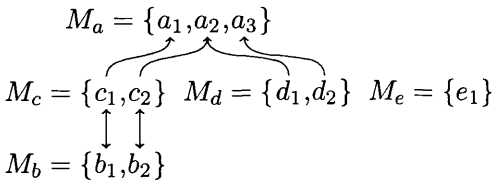
for $s, t \in S$, $x \in M_{\perp s}$, and $y \in M_{\perp t}$.

The *alphabet of communications* is constructed by disjointly uniting all the carrier sets extended by a bottom element $\perp$, but identifying carriers along sub-sort injections (this is captured by the equivalence relation $\sim_M$). Roggenbach [Rog06] proves that $\sim_M$ is an equivalence relation. This alphabet construction is used in the construction of CSP-CASL models in Section 8.1.2.

**Example 7.21** Let $\Sigma$ be the signature of the specification EXAMPLESPEC1 in Example 4.2 (on Page 60). Let $M$ be a $\Sigma$-model with:

- $M_a := \{a_1, a_2, a_3\}$,
- $M_b := \{b_1, b_2\}$,
- $M_c := \{c_1, c_2\}$,
- $M_d := \{d_1, d_2\}$,
- $M_e := \{e_1\}$,
- $(\mathtt{inj}_{c,a})_M(c_1) := a_1$,
- $(\mathtt{inj}_{c,a})_M(c_2) := a_2$,
- $(\mathtt{inj}_{d,a})_M(d_1) := a_2$,
- $(\mathtt{inj}_{d,a})_M(d_2) := a_3$,
- $(\mathtt{inj}_{b,c})_M(b_1) := c_1$,
- $(\mathtt{inj}_{b,c})_M(b_2) := c_2$,
- $(\mathtt{inj}_{c,b})_M(c_1) := b_1$, and
- $(\mathtt{inj}_{c,b})_M(c_2) := b_2$.

A diagram of these carrier sets follows where equal elements (according to the injections) have been connected:




The alphabet construction takes into account any carrier set elements of different sorts, that are united through injections into a common super sort. Such elements are collected together into the same equivalence classes. The *alphabet of communications* for this example turns out to be:

$$\begin{aligned} \mathit{Alph}(M) = \{\ & \{(a, a_1), (b, b_1), (c, c_1)\}, \\ & \{(a, a_2), (b, b_2), (c, c_2), (d, d_1)\}, \\ & \{(a, a_3), (d, d_2)\}, \\ & \{(e, e_1)\}, \\ & \{(a, \bot), (b, \bot), (c, \bot), (d, \bot)\}, \\ & \{(e, \bot)\}\ \} \end{aligned}$$

Now that we have flattened CASL models to alphabets we can construct the CSP domains from these alphabets (see Section 7.1), thus lifting CASL models to CSP domains.

Next, we look at lifting CASL model morphisms to alphabet translations, and following this, how to lift CASL model reducts to alphabet translations. These alphabet translations can then be further lifted to CSP domain translations.

### 7.2.2 Lifting of CASL Model Morphisms to Alphabet Translations

Now that we know how to construct alphabets from CASL models, the question arises of how to construct appropriate alphabet translations (between the flattened models) from CASL model morphisms. We answer this here.

We lift a *ResSubPCFOL*$^{=}$ model morphism (see Section 4.4) to form a translation on the level of the alphabet of communications. Given two restricted sub-sorted models $M_1$ and $M_2$ over a restricted sub-sorted signature $\Sigma$ and a restricted sub-sorted model morphism $h : M_1 \to M_2$, which is a mapping between carrier sets of $M_1$ and $M_2$, we define the alphabet translation $\alpha_h : \mathit{Alph}(M_1) \to \mathit{Alph}(M_2)$ as

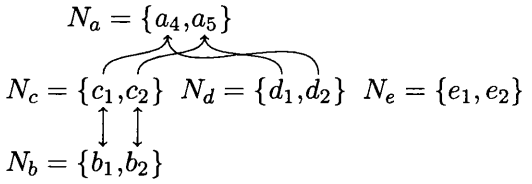
$$\alpha_h([(s, x)]_{\sim_{M_1}}) := [(s, h_\perp(x))]_{\sim_{M_2}} .$$

Here, we do not modify the sort and only map the underlying carrier set element across the underlying alphabet translation whilst also preserving the bottom elements. This make sense as the models are based on the same signature, thus they have the same sorts and only differ in the carrier set elements. The underlying model morphism maps carrier set elements from one model to another, thus we use this map , but additionally map bottom to bottom (see Section 4.5).

**Example 7.22** Let $\Sigma$ be the signature of the specification EXAMPLESPEC1 in Example 4.2 (on Page 60) and $M$ be the $\Sigma$-model in Example 7.21 (on Page 122). Let $N$ be the $\Sigma$-model which is the same as $M$ except for the carrier set of sorts $a$ and $e$ and the injections in to $a$ which are defined as:

- $N_a := \{a_4, a_5\}$,
- $N_e := \{e_1, e_2\}$,
- $(\mathtt{inj}_{c,a})_N(c_1) := a_4$,
- $(\mathtt{inj}_{c,a})_N(c_2) := a_5$,
- $(\mathtt{inj}_{d,a})_N(d_1) := a_5$, and
- $(\mathtt{inj}_{d,a})_N(d_2) := a_4$,

A diagram of these carrier sets follows where equal elements (according to the injections) have been connected:




The *alphabet of communications* of $N$ is:

$$\begin{aligned}\mathit{Alph}(N) = \{\ & \{(a, a_4), (b, b_1), (c, c_1), (d, d_2)\}, \\ & \{(a, a_5), (b, b_2), (c, c_2), (d, d_1)\}, \\ & \{(e, e_1)\}, \\ & \{(e, e_2)\}, \\ & \{(a, \perp), (b, \perp), (c, \perp), (d, \perp)\}, \\ & \{(e, \perp)\}\ \}\end{aligned}$$

Let $h : M \to N$ be a model morphism such that:

- $h_a(a_1) = a_4$,
- $h_a(a_2) = a_5$, and
- $h_a(a_3) = a_4$,

where all other mappings are the identity. Here we have chosen to collapse the elements $a_1$ and $a_3$ in $a_M$ onto the element $a_4$ in $a_N$. Thus, the alphabet translation turns out to be the map:

$$\begin{array}{lll}
\alpha_h : & \mathit{Alph}(M) & \to \mathit{Alph}(N) \\
& \{(a,a_1),(b,b_1),(c,c_1)\} & \mapsto \{(a,a_4),(b,b_1),(c,c_1),(d,d_2)\} \\
& \{(a,a_2),(b,b_2),(c,c_2),(d,d_1)\} & \mapsto \{(a,a_5),(b,b_2),(c,c_2),(d,d_1)\} \\
& \{(a,a_3),(d,d_2)\} & \mapsto \{(a,a_4),(b,b_1),(c,c_1),(d,d_2)\} \\
& \{(e,e_1)\} & \mapsto \{(e,e_1)\} \\
& \{(a,\bot),(b,\bot),(c,\bot),(d,\bot)\} & \mapsto \{(a,\bot),(b,\bot),(c,\bot),(d,\bot)\} \\
& \{(e,\bot)\} & \mapsto \{(e,\bot)\}
\end{array}$$

The interesting aspect here is that the equivalence classes $\{(a,a_1),(b,b_1),(c,c_1)\}$ and $\{(a,a_3),(d,d_2)\}$ from $\mathit{Alph}(M)$ are mapped onto the same equivalence class in $\mathit{Alph}(N)$. Hence, changes to super-sort carrier sets can have effects on sub-sort carrier sets. Also notice that this is an example of an alphabet translation which is neither injective nor surjective.

We now check that our lifting of *ResSubPCFOL*$^=$ model morphisms to alphabet translations is reasonable.

**Lemma 7.23** For all *ResSubPCFOL*$^=$ model morphisms $h : M_1 \to M_2$, the alphabet translation function $\alpha_h$ is well defined.

*Proof.* Let $h : M_1 \to M_2$ be a *ResSubPCFOL*$^=$ model morphism for a *ResSubPCFOL*$^=$ signature $\Sigma$ with sort set $S$ and sub-sort relation $\leq$. We show that if $(s,x) \sim_{M_1} (t,y)$ then $(s, h_\bot(x)) \sim_{M_2} (t, h_\bot(y))$ for all $s,t \in S$, $x \in M_{1\bot s}$, and $y \in M_{1\bot t}$.

Let $s,t \in S$, $x \in M_{1\bot s}$, and $y \in M_{1\bot t}$ such that $(s,x) \sim_{M_1} (t,y)$. By the definition of $\sim_{M_1}$ (defined in Section 7.2.1), there are two cases to consider:

**Case 1:** $x = y = \bot$. As $(s,x) \sim_{M_1} (t,y)$ holds, there exists $u \in S$ such that $s \leq u$ and $t \leq u$. We know that $h_\bot(x) = h_\bot(y) = \bot$ and as we have a top sort $u$ we know that $(s, h_\bot(x)) \sim_{M_2} (t, h_\bot(y))$.

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

1. $\exists u \in S$ such that $s \leq u$ and $t \leq u$.
2. $\forall u \in S$ such that $s \leq u$ and $t \leq u$ the following holds:

$$(\mathtt{inj}_{s,u})_{M_2}(h_\bot(x)) = (\mathtt{inj}_{t,u})_{M_2}(h_\bot(y))$$

For Condition 1, the proof is identical to the proof in Case 1. For Condition 2, let $u \in S$ such that $s \leq u$ and $t \leq u$. As $(s,x) \sim_{M_1} (t,y)$ holds, then for this $u$ we know that $(\mathtt{inj}_{s,u})_{M_1}(x) = (\mathtt{inj}_{t,u})_{M_1}(y)$. By applying $h_\bot$, we know that $h_\bot((\mathtt{inj}_{s,u})_{M_1}(x)) = h_\bot((\mathtt{inj}_{t,u})_{M_1}(y))$ holds.

By the homomorphism condition (which $h_\bot$ fulfils) we have:

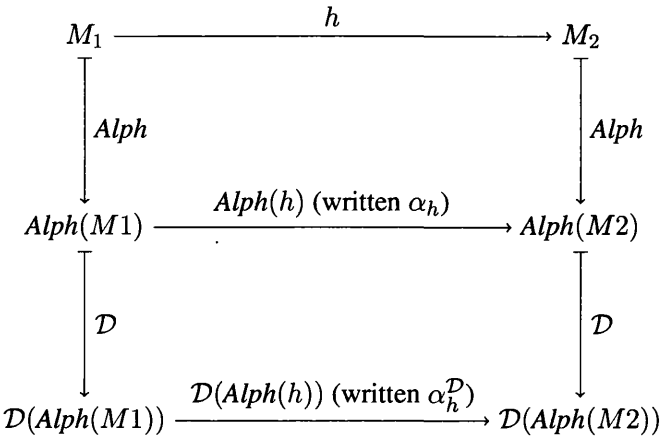



Figure 7.4: Illustration of lifting *ResSubPCFOL*$^=$ model morphisms to CSP domain translations.

- $h_\perp((\mathtt{inj}_{s,u})_{M_1}(x)) = (\mathtt{inj}_{s,u})_{M_2}(h_\perp(x))$ and
- $h_\perp((\mathtt{inj}_{t,u})_{M_1}(y)) = (\mathtt{inj}_{t,u})_{M_2}(h_\perp(y))$,

thus, we can conclude $(\mathtt{inj}_{s,u})_{M_2}(h_\perp(x)) = (\mathtt{inj}_{t,u})_{M_2}(h_\perp(y))$. □

We can now form the following functor from the model category (over signature $\Sigma$) to the category **SET**.

$$\begin{array}{lllll} Alph: & \mathbf{mod}(\Sigma) & \to & \mathbf{SET} \\ & M & \mapsto & Alph(M) \\ & h: M_1 \to M_2 & \mapsto & \alpha_h : Alph(M_1) \to Alph(M_2) \end{array}$$

We usually use the notation $\alpha_h$ in place of $Alph(h)$.

**Lemma 7.24** *Alph* is a valid functor, that is, *Alph* preserves identity morphisms and composition of functions.

*Proof.* This follows directly from the definition of *Alph* and the strict extension of morphisms (see Section 4.5). □

We can now form the CSP covariant domain translations from model morphisms via functor composition of *Alph* and $\mathcal{D}$ (see Section 7.1.2). For any model morphism $h : M_1 \to M_2$, we let $\alpha_h^{\mathcal{D}}$ be shorthand for $\mathcal{D}(Alph(h))$. This lifts the model morphism $h$ to the alphabet level and then further to the CSP context. This composition is illustrated in Figure 7.4.

As $\alpha_h$ is not necessarily injective for all model morphisms $h : M_1 \to M_2$, we can only form the covariant domain translations and not the contravariant domain translations as defined in Section 7.1.3. Formally, this restriction is realised by the functor *Alph* mapping into the category **SET** as opposed to **SET_INJ** where morphisms are injective functions between sets.

This concludes the lifting of CASL model morphisms to alphabet translations, and further to CSP covariant domain translations. These notions will be used when defining CSP-CASL model morphisms in Section 8.1.2.

### 7.2.3 Lifting of *ResSubPCFOL*$^{=}$ Reducts to Alphabet Translations

Another type of morphism that is interesting in the context of alphabet translations and CSP domain translations is that of CASL model reducts, which are induced by CASL signature morphisms. These can also be lifted to alphabet translations and thus, further lifted to CSP domain translations (both covariant and contravariant in this case). Such a lifting will be necessary, in Section 8.1.3, for defining an appropriate notion of model reduct for CSP-CASL models.

We now lift a *ResSubPCFOL*$^{=}$ reduct (see Section 4.4) to form an appropriate alphabet translation, which matches the alphabet construction of the model and its reduct.

Given two *ResSubPCFOL*$^{=}$ signatures $\Sigma$ and $\Sigma'$, a signature morphism $\sigma : \Sigma \to \Sigma'$, and a $\Sigma'$−model $M'$, we define the alphabet translation over the signature morphism $\sigma$ with respect to the $\Sigma'$-model $M'$ as $\alpha_{\sigma,M'} : \mathit{Alph}(M'|_\sigma) \to \mathit{Alph}(M')$ by:

$$\alpha_{\sigma,M'}([(s,x)]_{\sim_{M'|_\sigma}}) := [(\sigma(s),x)]_{\sim_{M'}} \ .$$

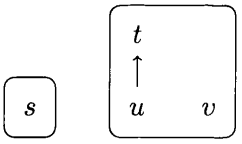
Here, we preserve the underlying carrier set element and only map the sort over the signature morphism, thus undoing the behaviour of the reduct. This gives us an appropriate alphabet translation from the alphabet of the reduced $\Sigma$-model to the alphabet of the original $\Sigma'$-model.

**Example 7.25** Let $\Sigma$ be the signature of EXAMPLESPEC3 and $\Sigma'$ be the signature of EXAMPLESPEC4 below.

**spec** EXAMPLESPEC3 =
  **sort** $s$
**end**

**spec** EXAMPLESPEC4 =
  **sorts** $u < t$;
    $v$
**end**

Below are diagrams of the underlying sub-sort relations.




Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism such that $\sigma(s) = u$. Let $M'$ be a $\Sigma'$-model such that

- $M'_u := \{u_1, u_2\}$,
- $M'_t := \{t_1, t_2, t_3\}$,
- $M'_v := \{v_1\}$,
- $(\mathtt{inj}_{u,t})_{M'}(u_1) := t_1$, and
- $(\mathtt{inj}_{u,t})_{M'}(u_2) := t_2$,

A diagram of these carrier sets follows where equal elements (according to the injections) have been connected:

$$\begin{array}{ccc} M'_t = \{t_1, & t_2, & t_3\} \\ \uparrow & \uparrow & \\ M'_u = \{u_1, & u_2\} & M'_v = \{v_1\} \end{array}$$

The *alphabet of communications* of $M'$ and $M'|_\sigma$ are:

$$\begin{aligned} \mathit{Alph}(M'|_\sigma) = \{\ & [(s,u_1)], \\ & [(s,u_2)], \\ & [(s,\bot)]\ \} \end{aligned} \qquad \begin{aligned} \mathit{Alph}(M') = \{\ & [(t,t_1),(u,u_1)], \\ & [(t,t_2),(u,u_2)], \\ & [(t,t_3)], \\ & [(v,v_1)], \\ & [(t,\bot),(u,\bot)], \\ & [(v,\bot)]\ \} \end{aligned}$$

The alphabet translation turns out to be the map:

$$\begin{array}{lccl} \alpha_{\sigma,M'}: & \mathit{Alph}(M'|_\sigma) & \to & \mathit{Alph}(M') \\ & [(s,u_1)] & \mapsto & [(t,t_1),(u,u_1)] \\ & [(s,u_2)] & \mapsto & [(t,t_2),(u,u_2)] \\ & [(s,\bot)] & \mapsto & [(t,\bot),(u,\bot)] \end{array}$$

This example illustrates the construction of alphabet translations from model reducts which are induced by signature morphisms.

We now check that our lifting of *ResSubPCFOL*$^=$ reducts to alphabet translations is reasonable and, furthermore, always yields injective alphabet translations.

**Lemma 7.26** For all *ResSubPCFOL*$^=$ signature morphisms $\sigma : \Sigma \to \Sigma'$, and all $\Sigma'$-models $M'$, $\alpha_{\sigma,M'}$ is well defined and injective. This lemma is presented from [Kah10].

*Proof.* See Appendix A for proof. □

As the alphabet translation $\alpha_{\sigma,M'}$ is injective we can form the partial inverse (denoted as ${\alpha_{\sigma,M'}}^{-1}$). This will be needed to transport process parameters to reduced signatures in certain proofs from Chapter 8 onwards.

This lifting also forms a functor, but to define this as a functor we first need to introduce a new category, which will be the source category of the functor. We define **Mod_Reduct** as the category where objects are pairs $(\Sigma, M)$ consisting of a *ResSubPCFOL*$^=$ signature $\Sigma$ and a $\Sigma$-model $M$. Morphisms are signature morphisms with the constraint that the source model is the reduct of the target model. More precisely, $\sigma : (\Sigma, M) \to (\Sigma', M')$ is a morphism from $(\Sigma, M)$ to $(\Sigma', M')$ if $\sigma : \Sigma \to \Sigma'$ is a *ResSubPCFOL*$^=$ signature morphism and $M = M'|_\sigma$.

**Lemma 7.27** **Mod_Reduct** is a category.

*Proof.* Identity morphisms are inherited from *ResSubPCFOL*$^=$ as the reduct along the identity signature morphism is the identity function. Composition of morphisms follows from the composition of reducts and associativity of composition of morphisms is also inherited. □

We can now form the following functor from **Mod_Reduct** to the category of sets with injective functions.

$$\begin{array}{lccl} \mathit{Alph\_Reduct}: & \textbf{Mod\_Reduct} & \to & \textbf{SET\_INJ} \\ & (\Sigma, M) & \mapsto & \mathit{Alph}(M) \\ & \sigma : (\Sigma, M'|_\sigma) \to (\Sigma', M') & \mapsto & \alpha_{\sigma,M'} : \mathit{Alph}(M'|_\sigma) \to \mathit{Alph}(M') \end{array}$$

Here, the application to objects coincides with the application of the *Alph* functor from Section 7.2.2. We usually use the notation $\alpha_{\sigma,M'}$ in place of $Alph\_Reduct(\sigma)$.

**Lemma 7.28** *Alph_Reduct* is a valid functor, that is, *Alph_Reduct* preserves identity morphisms and composition of functions.

*Proof.* This follows directly from definitions and the fact that *ResSubPCFOL*$^=$ model reducts preserve composition of signature morphisms (see Section 4.4). □

For each signature morphism $\sigma : \Sigma \to \Sigma'$ and $\Sigma'$-model $M'$, we now have an alphabet translation $\alpha_{\sigma,M'} : Alph(M'|_\sigma) \to Alph(M')$. We can lift this alphabet translation as is done in Section 7.1.2 to covariant CSP domain translations $\mathcal{D} \circ \alpha_{\sigma,M'}$ which we write as:

- $\alpha^{\mathcal{T}}_{\sigma,M'} : \mathcal{T}(Alph(M'|_\sigma)) \to \mathcal{T}(Alph(M'))$,
- $\alpha^{\mathcal{N}}_{\sigma,M'} : \mathcal{N}(Alph(M'|_\sigma)) \to \mathcal{N}(Alph(M'))$, and
- $\alpha^{\mathcal{F}}_{\sigma,M'} : \mathcal{F}(Alph(M'|_\sigma)) \to \mathcal{F}(Alph(M'))$.

Also as $\alpha_{\sigma,M'}$ is injective we can also lift the alphabet translation as is done in Section 7.1.3 to contravariant domain translations $\mathcal{D}^{op} \circ \alpha_{\sigma,M'}$ which we write as:

- $\hat{\alpha}^{\mathcal{T}}_{\sigma,M'} : \mathcal{T}(Alph(M')) \to \mathcal{T}(Alph(M'|_\sigma))$,
- $\hat{\alpha}^{\mathcal{N}}_{\sigma,M'} : \mathcal{N}(Alph(M')) \to \mathcal{N}(Alph(M'|_\sigma))$, and
- $\hat{\alpha}^{\mathcal{F}}_{\sigma,M'} : \mathcal{F}(Alph(M')) \to \mathcal{F}(Alph(M'|_\sigma))$.

These liftings are illustrated in Figure 7.5. We can lift a model reduct (induced by a signature morphism $\sigma$) to form an alphabet translation $\alpha_{\sigma,M'}$, which we can then lift further to both covariant and contravariant CSP domain translations $\alpha^{\mathcal{D}}_{\sigma,M'}$ and $\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}$, respectively. Finally, we can also form the partial inverse ${\alpha_{\sigma,M'}}^{-1}$ of the lifted alphabet translation $\alpha_{\sigma,M'}$, we cannot, and need not, lift this to the CSP context as this is not a total function.

This concludes all the necessary constructions required to build the CSP-CASL institutions in Chapter 8. We now present some commutativity properties of the lifted translations.

### 7.2.4 Commutativity Properties of Lifted Translations

The lifted model morphisms and lifted reducts to alphabet translations and domain translations interact well with each other, that is, they have reasonable commutativity properties. These properties are shown in the next two lemmas.

**Lemma 7.29** Let $\Sigma$ and $\Sigma'$ be *ResSubPCFOL*$^=$ signatures and $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Let $M'_1$ and $M'_2$ be $\Sigma'$-models and $h' : M'_1 \to M'_2$ be a *ResSubPCFOL*$^=$ model morphism. Then the following diagram commutes:

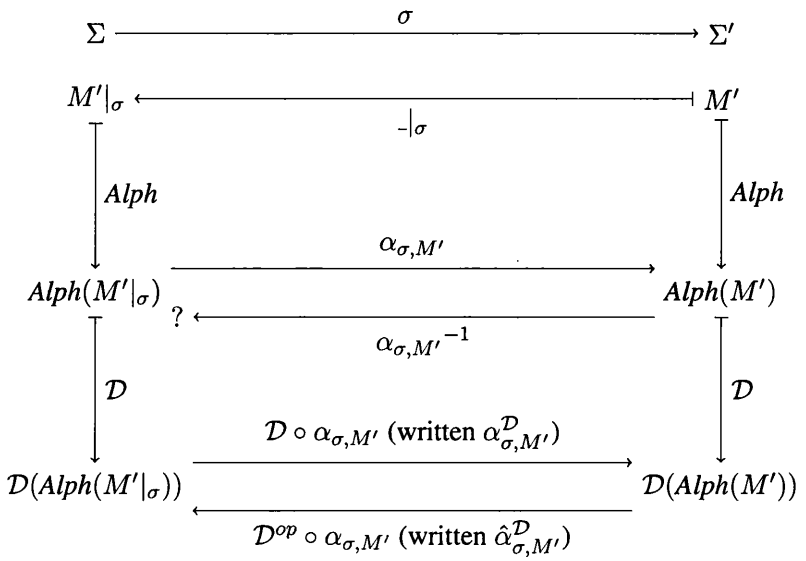
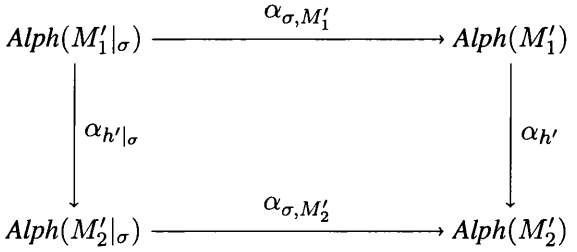



Figure 7.5: Illustration of lifting *ResSubPCFOL*$^=$ reducts to CSP domain translations.




that is, $\alpha_{h'} \circ \alpha_{\sigma,M_1'} = \alpha_{\sigma,M_2'} \circ \alpha_{h'|_\sigma}$.

*Proof.* Let $[(s,x)]_{\sim_{M_1'|_\sigma}} \in Alph(M_1'|_\sigma)$.

$$
\begin{aligned}
&\alpha_{h'}(\alpha_{\sigma,M_1'}([(s,x)]_{\sim_{M_1'|_\sigma}})) \\
&= \alpha_{h'}([(\sigma(s),x)]_{\sim_{M_1'}}) && \text{Definition of } \alpha_{\sigma,M_1'}. \\
&= [(\sigma(s),(h'_\perp)_{\sigma(s)}(x))]_{\sim_{M_2'}} && \text{Definition of } \alpha_{h'}. \\
&= [(\sigma(s),((h'|_\sigma)_\perp)_s(x))]_{\sim_{M_2'}} && \text{By definition of morphism reduct.} \\
&= \alpha_{\sigma,M_2'}([(s,((h'|_\sigma)_\perp)_s(x))]_{\sim_{M_2'|_\sigma}}) && \text{Definition of } \alpha_{\sigma,M_2'}. \\
&= \alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}([(s,x)]_{\sim_{M_1'|_\sigma}})) && \text{Definition of } \alpha_{h'|_\sigma}.
\end{aligned}
$$

□

If we consider only CSP denotations which have suitable controlled traces then we obtain a commutativity property on the CSP domain level.

**Lemma 7.30** Let $\Sigma$ and $\Sigma'$ be *ResSubPCFOL*$^=$ signatures and $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Let $M'_1$ and $M'_2$ be $\Sigma'$-models and $h' : M'_1 \to M'_2$ be a *ResSubPCFOL*$^=$ model morphism. Let $A \subseteq \mathit{Alph}(M'_1|_\sigma)$ and $B \subseteq \mathit{Alph}(M'_1)$ such that $\alpha^{\mathcal{P}\checkmark}_{\sigma,M'_1}(A) \subseteq B$. Finally, let $d \in \mathcal{D}(\mathit{Alph}(M'_1))$ with $cTr^{\mathcal{D}}(d') \in \mathcal{T}(B)$. Then

$$\alpha^{\mathcal{D}}_{h'|_\sigma}(\hat{\alpha}^{\mathcal{D}}_{\sigma,M'_1}(d)) = \hat{\alpha}^{\mathcal{D}}_{\sigma,M'_2}(\alpha^{\mathcal{D}}_{h'}(d)) \ .$$

*Proof.* and follows from Lemma 7.29. □

This concludes all the preliminary notations, functions and functors that we require to build the CSP-CASL institutions in Chapter 8. Ultimately, we have shown how to construct CSP domains from CASL models and how to lift CASL model morphisms and reducts to CSP domain translations. We have also established several properties which will be used in future chapters.

*Chapter 8*

# The CSP-CASL Institutions

**Contents**



This chapter aims to establish CSP-CASL in a standard framework which allows for structured CSP-CASL specifications (that is, CSP-CASL specifications which use the structuring language presented in Section 4.7), generic CSP-CASL specifications and instantiations of such specifications. To this end, we formalise CSP-CASL in the framework of institutions (see Chapter 4). Generic specifications and instantiations further require that the CSP-CASL institutions have an additional property, namely amalgamation. By establishing CSP-CASL as institutions, we not only place CSP-CASL in a well understood and developed framework, but also demonstrate the framework's generality.

In this chapter we present the formal syntax and semantics of CSP-CASL. We formalise CSP-CASL as three institutions, one for each of the main CSP semantics, namely the Traces semantics $\mathcal{T}$, the Failures/Divergences semantics $\mathcal{N}$ and the Stable-Failures semantics $\mathcal{F}$. The institutions for CSP-CASL are naturally based on institutions for CASL [Mos04] and for CSP [MR07], using the ideas for the original CSP-CASL semantics [Rog06] for the combination. These three institutions share the same notions of signatures and sentences and only vary in the models and satisfaction relations. However, their respective model categories and satisfaction relations are defined following a common scheme. Thus, CSP-CASL specifications can be interpreted in any of the main CSP semantics.

Our formalisation of CSP-CASL also allows for loose process semantics, which not only make sense in the methodological sense, but are also required for generic specifications (see Chapter 3). We prove that our CSP-CASL institutions exhibit a suitable amalgamation property, allowing us to form generic and instantiated CSP-CASL specifications. Following this, we then

discuss how CSP-CASL can be extended to support CSP's notion of channels. Finally, we discuss some attractive, possible extensions of CSP-CASL.

## 8.1 Construction of the CSP-CASL Institutions

Here, we first present CSP-CASL signatures, followed by models, model reducts and finally, the CSP-CASL satisfaction relation and the proof of the satisfaction condition.

### 8.1.1 Signatures

We now formally introduce the category of CSP-CASL signatures CSPCASLSIG. A signature in CSP-CASL captures the same elements as a CASL signature, that is, sort, operation and predicate symbols, but in addition also captures process names and their associated type information.

Formally, a CSP-CASL signature is a pair $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$ where:

- $\Sigma_{Data}$ is a restricted sub-sorted (i.e., *ResSubPCFOL*$^{=}$) signature (see Section 4.4), and
- $\Sigma_{Proc} = (N_{w,comms})_{w \in S^*, comms \in S_{\downarrow}}$ is a family of finite sets of process names. Such a process name $n$ is typed in the sort symbols $S$ of the data signature part ($\Sigma_{Data}$):
  - a string $w = \langle s_1, \ldots, s_k \rangle$, $s_i \in S$ for $1 \leq i \leq k$, $k \geq 0$, which is $n$'s parameter type. A process name without parameters has the empty sequence $\langle\rangle$ as its parameter type.
  - a set $comms \subseteq S$ which collects all types of events in which the process $n$ can possibly engage in (when not diverging). We require the set $comms$ to be downward closed under the sub-sort relation, that is, $comms \in S_{\downarrow} = \{X \subseteq S \mid X = {\downarrow}X\}$, where ${\downarrow}X = \{y \in S \mid \exists x \in X \bullet y \leq x\}$ for $X \subseteq S$.

Following CASL, process names are always fully qualified when used, for example, in sentences. We write $n_{w,comms}$ for a fully qualified process name $n \in N_{w,comms}$ when using mathematics, and $n : w, comms$ when writing fully qualified process names in specifications. We drop the subscripts when the parameters and communications of the process names are clear from the context.

An interesting point is why we chose to have the communication sorts $comms$ as part of the identity of a process name. Take the simple example below

**spec** SP =
  **data**
    **sorts** $s, t$
  **process**
    $P : s$;
    $P : t$
**end**

There are two sorts $s$ and $t$ and two processes both named $P$, one that communicates over $s$ and one that communicates over $t$. As the alphabet construction makes communications from different sorts which are not in a sub-sort relation disjoint, then these two process names $P$ can be distinguished using their communications (i.e., we can observe that they are different). It makes sense to distinguish these two process names but allow them to share the same underlying name. We therefore include the communication sorts as part of the identity of a process name.

Now that we have defined CSP-CASL signatures, we can define CSP-CASL signature morphisms. CSP-CASL signature morphisms work in a similar way to CASL signature morphisms, but in addition also map process names between the corresponding signatures.

Given CSP-CASL signatures $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$ and $\Sigma'_{CC} = (\Sigma'_{Data}, \Sigma'_{Proc})$, with $S$ being the sort set of $\Sigma_{Data}$, a CSP-CASL signature morphism is a pair $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ where:

- $\sigma : \Sigma_{Data} \to \Sigma'_{Data}$ is a restricted sub-sorted (i.e., *ResSubPCFOL*$^{=}$) signature morphism (see Section 4.4).
- $\nu = (\nu_{w,comms})_{w \in S^*, comms \in S_{\downarrow}}$ is a family of functions such that

  $$\nu_{w,comms} : N_{w,comms} \to N'_{\sigma(w),\downarrow(\sigma(comms))}$$

  is a mapping of process names. Another way to express this is that a process name $n \in N_{w,comms}$ is mapped to $\nu_{w,comms}(n) = n'$, where $n' \in N'_{\sigma(w),\downarrow(\sigma(comms))}$. We also write $\nu(n_{w,comms}) = n'_{\sigma(w),\downarrow(\sigma(comms))}$.

The type of process name translations ensure that both the parameter types, as well as the communication set, are translated with the signature morphism $\sigma$ of the data part. The translation of the type of process names ensures that the satisfaction condition holds for CSP-CASL, see Section 8.1.5.

**Example 8.1** As an example of a CSP-CASL signature morphism consider the following two specifications:

| | |
|---|---|
| **spec** SP1 = | **spec** SP2 = |
| **data** | **data** |
| **sorts** $s < u$ | **sorts** $s, t < u$ |
| **process** = | **process** = |
| $P : s;$ | $P : s;$ |
| $P : s, u;$ | $P : s, t, u;$ |
| $Q\ (s) : s, u;$ | $R\ (s) : s, t, u;$ |
| **end** | **end** |

Specification SP1 contains two sorts $s$ and $u$, with $s$ being a sub-sort of $u$; and three process names, namely, $P_{\langle\rangle,\{s\}}$, $P_{\langle\rangle,\{s,\,u\}}$ and $Q_{\langle s\rangle,\{s,\,u\}}$. Specification SP2 contains three sorts $s$, $t$ and $u$, with $s$ and $t$ being sub-sorts of $u$; and three process names, namely, $P_{\langle\rangle,\{s\}}$, $P_{\langle\rangle,\{s,\,t,\,u\}}$ and $Q_{\langle s\rangle,\{s,\,t,\,u\}}$.

There is a signature morphism between the signatures of SP1 and SP2, where $s \mapsto s$, $u \mapsto u$, $P_{\langle\rangle,\{s\}} \mapsto P_{\langle\rangle,\{s\}}$, $P_{\langle\rangle,\{s,\,u\}} \mapsto P_{\langle\rangle,\{s,\,t,\,u\}}$, and $Q_{\langle s\rangle,\{s,\,u\}} \mapsto R_{\langle s\rangle,\{s,\,t,\,u\}}$. As the parameter

sorts and communication set of a process name are translated in accordance with the data part of a signature morphism, once the data part of a signature morphism is fixed, there is only a choice in the mapping of process names and and not their types.

We now define composition of CSP-CASL signature morphisms. Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ and $\theta' = (\sigma', \nu') : \Sigma'_{CC} \to \Sigma''_{CC}$ be two CSP-CASL signature morphisms with $S$ being the sort set of $\Sigma_{CC}$ and $N$, $N'$, and $N''$ the families of process names of $\Sigma_{CC}$, $\Sigma'_{CC}$, and $\Sigma''_{CC}$ respectively. The composition of these two signature morphisms is defined component wise as:

$$\theta' \circ \theta = (\sigma', \nu') \circ (\sigma, \nu) := ((\sigma' \circ \sigma), (\nu' \circ \nu))$$

where $\nu' \circ \nu = ((\nu' \circ \nu)_{w,comms})_{w \in S^*, comms \in S_\downarrow}$ is a family of functions defined as

$$\begin{array}{rccl} (\nu' \circ \nu)_{w,comms} : & N_{w,comms} & \to & N''_{(\sigma' \circ \sigma)(w), \downarrow(\sigma'(\downarrow(\sigma(comms))))} \\ & n & \mapsto & \nu'_{\sigma(w), \downarrow(\sigma(comms))}(\nu_{w,comms}(n)) \end{array}$$

CSP-CASL's identity signatures morphisms are defined in the obvious way. Given a CSP-CASL signature $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$, with sort set $S$ and family of process names $\Sigma_{Proc} = (N_{w,comms})_{w \in S^*, comms \in S_\downarrow}$, let $Id_{\Sigma_{CC}} = (Id_{\Sigma_{Data}}, Id_{\Sigma_{Proc}}) : \Sigma_{CC} \to \Sigma_{CC}$ be the CSP-CASL signature morphism defined by

- $Id_{\Sigma_{Data}} : \Sigma_{Data} \to \Sigma_{Data}$ is the identity morphism of the restricted sub-sorted signature $\Sigma_{Data}$ from *ResSubPCFOL*$^=$, and
- $Id_{\Sigma_{Proc}} = ((Id_{\Sigma_{Proc}})_{w,comms})_{w \in S^*, comms \in S_\downarrow} : \Sigma_{Proc} \to \Sigma_{Proc}$ is a family of identity functions, that is, $(Id_{\Sigma_{Proc}})_{w,comms}(n_{w,conms}) := n_{w,comms}$ for any process name $n_{w,comms} \in N_{w,comms}$.

Note that the CSP-CASL identity signature morphism is type correct as it fits into the type $N_{w,comms} \to N_{Id_{\Sigma_{Data}}(w), \downarrow(Id_{\Sigma_{Data}}(comms))}$ as given any $comms \in S_\downarrow$, we know by construction that $\downarrow(Id_{\Sigma_{Data}}(comms)) = comms$.

**Theorem 8.2** CSP-CASL signatures and signature morphisms form a category.

*Proof.* The proofs that identity morphisms exist, signature morphisms compose, and that composition of signature morphisms is associative are straightforward and follow directly from definitions. □

This concludes the construction of the signature category for CSP-CASL. Next, we present CSP-CASL models and model morphisms.

### 8.1.2 Models

Each CSP-CASL signature $\Sigma_{CC}$ induces a category of models $\mathbf{mod}_{\mathcal{D}}(\Sigma_{CC})$ (for a particular CSP semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$). We define this category for a given signature $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$, with sort set $S$ and process part $\Sigma_{proc} = (N_{w,comms})_{w \in S^*, comms \in S_\downarrow}$, for a particular CSP semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

The models for the category $\mathbf{mod}_{\mathcal{D}}(\Sigma_{CC})$ are defined to be pairs $(M, I)$ (called $\Sigma_{CC}$-models) where

- $M$ is a restricted sub-sorted (i.e., *ResSubPCFOL*$^=$) model in $\mathbf{mod}(\Sigma_{Data})$.
- $I = (I_{w,comms})_{w \in S^*, comms \in S_\downarrow}$ is a family of process interpretation functions, such that
  1. each function $I_{w,comms} : N_{w,comms} \times \overline{s_1}_M \times \ldots \times \overline{s_k}_M \to \mathcal{D}(Alph(M))$, where $w = \langle s_1, \ldots, s_k \rangle$, is type correct, and
  2. for each function $I_{w,comms} : N_{w,comms} \times \overline{s_1}_M \times \ldots \times \overline{s_k}_M \to \mathcal{D}(Alph(M))$, where $w = \langle s_1, \ldots, s_k \rangle$, it holds that

$$cTr^{\mathcal{D}}(I_{w,comms}(n, a_1, \ldots, a_k)) \in \mathcal{T}(\overline{comms}_M)$$

  for all process names $n \in N_{w,comms}$ and all parameter values $a_1, \ldots, a_k \in \overline{s_1} \times \ldots \times \overline{s_k}$. Here, $\overline{comms}_M = \bigcup_{s \in comms} \overline{s}_M$ (see Section 7.2.1).

  We drop the subscripts $w$ and $comms$ when this type information is clear from the context.

  Each process interpretation map $I_{w,comms}$ is a function which takes a process name and suitable parameters, and yields a CSP denotation for the particular CSP semantics $\mathcal{D}$. We usually write $I(n(a_1, \ldots, a_k))$ in place of $I(n, a_1, \ldots, a_k)$. The second condition then constrains these denotations to ones where the controlled traces do not stray outside of the declared communications.

We also drop the subscript $\mathcal{D}$ when the CSP semantics is clear.

The controlled traces condition is used to restrict the process interpretations to only those which communicate in the declared set of communications. In the Traces and Stable-Failures semantics this is just the trace component of the denotations. Thus, we ensure all traces are contained within the declared set. The Failures/Divergences semantics however, requires the set of divergences to be extension closed over the entire alphabet (this includes extensions that stray outside the declared communications set). Thus, in the Failures/Divergences semantics we only require the traces which are not extensions of divergences to be in the declared set of communications. see the definition of controlled traces in Section 7.1.4.

We now define CSP-CASL model morphisms, which naturally use restricted sub-sorted model morphisms. Each restricted sub-sorted model morphism $h : M_1 \to M_2$ induces the alphabet translation function $\alpha_h : Alph(M_1) \to Alph(M_2)$ and the domain translation functions $\alpha_h^{\mathcal{D}} : \mathcal{D}(Alph(M_1)) \to \mathcal{D}(Alph(M_2))$ (defined in Section 7.2.2). A CSP-CASL model morphism $h : (M_1, I_1) \to (M_2, I_2)$ (also called a $\Sigma_{CC}$-model morphism) between two CSP-CASL models $(M_1, I_1)$ and $(M_2, I_2)$ is a restricted sub-sorted (i.e., *ResSubPCFOL*$^=$) model morphism $h : M_1 \to M_2$ such that

$$\alpha_h^{\mathcal{D}}(I_{1\,w,comms}(n(a_1, \ldots, a_k))) \bowtie I_{2w,comms}(n(\alpha_h(a_1), \ldots, \alpha_h(a_k)))$$

for all $w = \langle s_1, \ldots, s_k \rangle \in S^*$, $comms \in S_\downarrow$, process names $n \in N_{w,comms}$ and all parameters $a_1, \ldots, a_k \in \overline{s_1} \times \ldots \times \overline{s_k}$ where $\bowtie$ is $\sqsupseteq_{\mathcal{T}}$ for $\mathcal{D} = \mathcal{T}$, $\sqsubseteq_{\mathcal{N}}$ for $\mathcal{D} = \mathcal{N}$, and $\sqsupseteq_{\mathcal{F}}$ for $\mathcal{D} = \mathcal{F}$.

Here, we require that the forward (covariant) translations of the resulting denotations from the process interpretation map $I_1$ are in an appropriate refinement relation with the denotations

produced by $I_2$. Instead of using the covariant translation on $I_1$ we could have used the contravariant translation on $I_2$, but this would have added an unnecessary injectivity constraint on the model morphism $h : M_1 \to M_2$.

Here, we choose the direction of the refinements to match the fixed point theory in each of the CSP semantics. We conjecture that the limit in the model category will correspond to the least fixed point in terms of CSP. For there to be any hope of this working out, we must choose the refinement directions as we have. For the Traces and Stable-Failures semantics, we choose the $\sqsupseteq$ direction as, DIV is the most refined process, and thus can be the initial object. We choose the opposite direction for the Failures/Divergences semantics as there is no most refined process. However, there is a least refined process, that is, DIV. Thus, DIV becomes the initial object when using each of the semantics.

**Example 8.3** Consider the following specification

**logic** CSPCASL

**spec** INITIAL =
  **data free type** $s ::= a$
  **process** $n : s$;
**end**

The data part has a single model $M$ up to isomorphism. The process name $n$ is loose and can be interpreted in any legal way.

Let $(M, I)$ be a model in the Traces semantics such that $I$ interprets $n$ as DIV, that is, $I(n) = \{\langle\rangle\}$. Our choice of refinement direction causes $(M, I)$ to be the initial object in the model category. Similarly $(M, J)$ would be initial in the Stable-Failures semantics where $J$ interprets $n$ as DIV, that is, $J(n) = (\{\langle\rangle\}, \emptyset)$. Finally, $(M, K)$ would be initial with the Failures/Divergences semantics where $K$ interprets $n$ as DIV, that is, $K(n) = (Alph(M)^{*\surd} \times \mathcal{P}(Alph(M)^{\surd}), Alph(M)^{*\surd})$.

CSP-CASL inherits its notion of model morphism composition from the model category of *ResSubPCFOL*$^{=}$. Given a $\Sigma_{CC}$-model $(M, I)$, the identity morphism for the CSP-CASL model $(M, I)$ (written as $Id_{(M,I)}$) is defined to be the identity model morphism $Id_M : M \to M$ for the restricted sub-sorted model $M$ from the model category of *ResSubPCFOL*$^{=}$.

**Lemma 8.4** CSP-CASL models and model morphisms form a category, that is, $\mathbf{mod}_{\mathcal{D}}(\Sigma_{CC})$ is a category.

*Proof.* We prove that identity morphisms exist, model morphisms compose, and that composition of model morphisms is associative.

**Identity** Let $(M, I)$ be a $\Sigma_{CC}$-model. We show that $Id_M$ is the identity morphism for the model $(M, I)$. This is the case if

$$\alpha^{\mathcal{D}}_{Id_M}(I(n(a_1, \ldots, a_k))) \bowtie I(n(\alpha_{Id_M}(a_1), \ldots, \alpha_{Id_M}(a_k)))$$

holds for each process name $n$ and all appropriate parameters $a_1, \ldots, a_k$. As $Id_M$ is the identity function on carrier sets we know $\alpha_{Id_M}$ and $\alpha^{\mathcal{D}}_{Id_M}$ are also identity functions

(Lemmas 7.24 and 7.6). Furthermore, as process refinement is reflexive (see Section 2.5), we conclude that this equation holds, hence $Id_M$ is a valid CSP-CASL model morphism. The fact that $Id_M$ is both the left and right unit is inherited from *ResSubPCFOL*$^=$.

**Composition** Let $h_1 : (M_1, I_1) \to (M_2, I_2)$ and $h_2 : (M_2, I_2) \to (M_3, I_3)$ be $\Sigma_{CC}$-model morphisms. We must show that $h_2 \circ h_1$ is a $\Sigma_{CC}$-model morphism between $(M_1, I_1)$ and $(M_3, I_3)$. To this end, we must show

$$\alpha^{\mathcal{D}}_{(h_2 \circ h_1)}(I_1(n(a_1, \ldots, a_k))) \bowtie I_3(n(\alpha_{(h_2 \circ h_1)}(a_1), \ldots, \alpha_{(h_2 \circ h_1)}(a_k)))$$

for each process name $n$ and all appropriate parameters $a_1, \ldots, a_k$. This equation follows from the facts that both the alphabet liftings and the domain liftings preserve composition (Lemmas 7.24 and 7.6), domain translations preserve refinement (Corollary 7.15) and CSP refinements are transitive (see Section 2.5).

**Associativity** Associativity of composition is inherited from *ResSubPCFOL*$^=$ as CSP-CASL model morphisms are *ResSubPCFOL*$^=$ model morphisms. □

This concludes our construction of CSP-CASL models and model morphisms, that is, the category $\mathbf{mod}_{\mathcal{D}}(\Sigma_{CC})$ for each CSP-CASL signature $\Sigma_{CC}$ and each CSP semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

This completes the definition of the object map of CSP-CASL's $\mathbf{mod}_{\mathcal{D}}$ functor. Next, we define the morphism map, that is, reducts.

### 8.1.3 Model Reducts

We now define the reducts of CSP-CASL models and CSP-CASL model morphisms for a fixed CSP semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$. To this end, we make use of restricted sub-sorted model and model morphism reducts. Each restricted sub-sorted signature morphism $\sigma : \Sigma \to \Sigma'$ and restricted sub-sorted $\Sigma'$-model $M'$ induces the contravariant CSP domain translation function $\hat{\alpha}^{\mathcal{D}}_{\sigma,M'} : \mathcal{D}(Alph(M')) \to \mathcal{D}(Alph(M'|_\sigma))$ (defined in Section 7.2.3).

Given a CSP-CASL signature morphism $\theta = (\sigma, \nu) : \Sigma_{cc} \to \Sigma'_{cc}$ and a $\Sigma'_{cc}$-model $(M', I')$, we define the $\Sigma_{cc}$-model $\mathbf{mod}(\theta)(M', I')$ (usually written $(M', I')|_\theta$) as:

$$(M', I')|_\theta := (M'|_\sigma, I'|_\theta)$$

where

$$(I'|_\theta)_{w,comms}(n(a_1, \ldots, a_k)) := \hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(I'_{\sigma(w),\downarrow(\sigma(comms))}(n'(\alpha_{\sigma,M'}(a_1), \ldots, \alpha_{\sigma,M'}(a_k))))$$

and $\nu_{w,comms}(n_{w,comms}) = n'_{\sigma(w),\downarrow(\sigma(comms))}$ for each process name $n_{w,comms}$ and appropriate parameters $a_1, \ldots, a_k$.

On the data side we just define the reduct to be the *ResSubPCFOL*$^=$ model reduct. On the process side we define the notion of a reduct on the process interpretation map $I'$. A reduced process interpretation map works in a similar way to many-sorted algebra reducts. In order to interpret a process name with parameters in the "smaller" signature, we first translate the

process name and parameters to the "larger" signature and then take the interpretation using the original process interpretation map. However, unlike with reduced many-sorted algebras, we have to translate the resulting CSP denotation back to the reduced alphabet using the contravariant translation function $\hat{\alpha}^{D}_{\sigma,M'}$.

Now that we have defined CSP-CASL model reducts we turn our attention to reducts of model morphisms. Given a CSP-CASL signature morphism $\theta = (\sigma, \nu) : \Sigma_{cc} \to \Sigma'_{cc}$ and a $\Sigma'_{cc}$-model morphism $h' : (M'_1, I'_1) \to (M'_2, I'_2)$, we define the $\Sigma_{cc}$-model morphism $\mathbf{mod}(\theta)(h') : (M'_1, I'_1)|_\theta \to (M'_2, I'_2)|_\theta$ (usually written $h'|_\theta$) as:

$$h'|_\theta := h'|_\sigma$$

that is, $h'|_\theta$ is the *ResSubPCFOL*$^=$ model morphism reduct $h'|_\sigma$.

**Lemma 8.5** CSP-CASL model reducts and model morphism reducts form a functor, that is, given a CSP-CASL signature morphism $\theta : \Sigma_{cc} \to \Sigma'_{cc}$, $\mathbf{mod}(\theta) : \mathbf{mod}(\Sigma'_{CC}) \to \mathbf{mod}(\Sigma_{CC})$ is a functor.

*Proof.* We prove that model reducts as well as model morphism reducts are well defined, model morphism reducts preserve identity model morphisms, and model morphism reducts preserve composition of model morphisms. Let $\Sigma_{cc} = (\Sigma_{Data}, \Sigma_{Proc})$ be a CSP-CASL signature (with sort set $S$) and $\theta = (\sigma, \nu) : \Sigma_{cc} \to \Sigma'_{cc}$ be a CSP-CASL signature morphism.

**Model reducts are well defined** Let $(M', I')$ be a $\Sigma'_{cc}$-model. We must show that $(M', I')|_\theta$ is a $\Sigma_{cc}$-model. We have that $(M', I')|_\theta = (M'|_\sigma, I'|_\theta)$ by definition. As $\sigma$ is a *ResSubPCFOL*$^=$ signature morphism, it follows that $M'|_\sigma \in \mathbf{mod}(\Sigma_{Data})$. We also know that $I'_\theta$ is type correct thanks to its construction. Thus, we just have to show that the controlled traces are contained within the allowed set.

To this end, let $w = \langle s_1, \ldots, s_k \rangle \in S^*$, $comms \in S_!$, and $n_{w,comms}$ be a process name in $\Sigma_{Proc}$. Furthermore, let $\nu(n_{w,comms}) = n'_{\sigma(w),!(\sigma(comms))}$ and $a_1, \ldots, a_k \in \overline{s_1}_{M'|_\sigma} \times \ldots \times \overline{s_k}_{M'|_\sigma}$. We must show

$$cTr^{\mathcal{D}}((I'|_\theta)_{w,comms}(n(a_1, \ldots, a_k))) \in \mathcal{T}(\overline{comms}_{M'|_\sigma}).$$

Unfolding the reduct definition we must establish

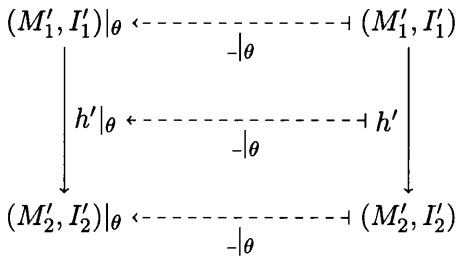
$$cTr^{\mathcal{D}}(\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(I'_{\sigma(w),!(\sigma(comms))}(n'(\alpha_{\sigma,M'}(a_1), \ldots, \alpha_{\sigma,M'}(a_k))))) \in \mathcal{T}(\overline{comms}_{M'|_\sigma}).$$

As $(M', I')$ is a $\Sigma'_{CC}$-model, we know

$$cTr^{\mathcal{D}}(I'_{\sigma(w),!(\sigma(comms))}(n'(\alpha_{\sigma,M'}(a_1), \ldots, \alpha_{\sigma,M'}(a_k)))) \in \mathcal{T}(\overline{!(\sigma(comms))}_{M'}).$$

From this combined with the facts that controlled traces are preserved after translation (Lemma 7.17) and that $\overline{!(\sigma(comms))}_{M'} = \alpha^{\mathcal{P}\surd}_{\sigma,M'}(\overline{comms}_{M'|_\sigma})$ we obtain our goal.

**Model morphism reducts are well defined** We prove that reducing a $\Sigma'_{CC}$-model morphism yields a valid $\Sigma_{CC}$-model morphism. To this end, let $h' : (M'_1, I'_1) \to (M'_2, I'_2)$ be a $\Sigma'_{cc}$-model morphism. We show $h'|_\theta$ is $\Sigma_{cc}$-model morphism from $(M'_1, I'_1)|_\theta$ to $(M'_2, I'_2)|_\theta$. This is illustrated below:




As $h'|_\theta = h'|_\sigma$ by definition, we know $h'|_\sigma : M_1'|_\sigma \to M_2'|_\sigma$ is a *ResSubPCFOL*$^=$ model morphism. Thus, we just need to show that $h'|_\sigma$ respects the refinement condition of model morphisms.

To this end, let $w = \langle s_1, \ldots, s_k \rangle \in S^*$, $comms \in S_\downarrow$, $n_{w,comms}$ be a process name in $\Sigma_{Proc}$. Furthermore, let $a_1, \ldots, a_k \in \overline{s_1}_{M'|_\sigma} \times \ldots \times \overline{s_k}_{M'|_\sigma}$ be alphabet elements. We must show

$$\alpha^{\mathcal{D}}_{h'|_\sigma}(I_1'|_\theta(n(a_1, \ldots, a_k))) \bowtie I_2'|_\theta(n(\alpha_{h'|_\sigma}(a_1), \ldots, \alpha_{h'|_\sigma}(a_k)))$$

where $\bowtie$ is $\sqsupseteq_{\mathcal{T}}$ for $\mathcal{D} = \mathcal{T}$, $\sqsubseteq_{\mathcal{N}}$ for $\mathcal{D} = \mathcal{N}$, and $\sqsupseteq_{\mathcal{F}}$ for $\mathcal{D} = \mathcal{F}$. If we expand the definition of model reduct in the above refinement equation, we must show:

$$\alpha^{\mathcal{D}}_{h'|_\sigma}(\hat{\alpha}^{\mathcal{D}}_{\sigma,M_1'}(I_1'(\nu(n)(\alpha_{\sigma,M_1'}(a_1), \ldots, \alpha_{\sigma,M_1'}(a_k))))) \bowtie \hat{\alpha}^{\mathcal{D}}_{\sigma,M_2'}(I_2'(\nu(n)(\alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}(a_1)), \ldots, \alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}(a_k))))).$$

As $h'$ is a valid $\Sigma'_{CC}$-model morphism we know specifically that

$$\alpha^{\mathcal{D}}_{h'}(I_1'(\nu(n)(\alpha_{\sigma,M_1'}(a_1), \ldots, \alpha_{\sigma,M_1'}(a_k)))) \bowtie I_2'(\nu(n)(\alpha_{h'}(\alpha_{\sigma,M_1'}(a_1)), \ldots, \alpha_{h'}(\alpha_{\sigma,M_1'}(a_k))))$$

holds. As $\alpha_{h'} \circ \alpha_{\sigma,M_1'} = \alpha_{\sigma,M_2'} \circ \alpha_{h'|_\sigma}$ (Lemma 7.29) we therefore know

$$\alpha^{\mathcal{D}}_{h'}(I_1'(\nu(n)(\alpha_{\sigma,M_1'}(a_1), \ldots, \alpha_{\sigma,M_1'}(a_k)))) \bowtie I_2'(\nu(n)(\alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}(a_1)), \ldots, \alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}(a_k)))).$$

By applying $\hat{\alpha}^{\mathcal{D}}_{\sigma,M_2'}$ to both sides of the above refinement equation and thanks to preservation of refinement (Corollary 7.15), we obtain

$$\hat{\alpha}^{\mathcal{D}}_{\sigma,M_2'}(\alpha^{\mathcal{D}}_{h'}(I_1'(\nu(n)(\alpha_{\sigma,M_1'}(a_1), \ldots, \alpha_{\sigma,M_1'}(a_k))))) \bowtie \hat{\alpha}^{\mathcal{D}}_{\sigma,M_2'}(I_2'(\nu(n)(\alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}(a_1)), \ldots, \alpha_{\sigma,M_2'}(\alpha_{h'|_\sigma}(a_k))))).$$

Finally, as $\hat{\alpha}^{\mathcal{D}}_{\sigma,M_2'} \circ \alpha^{\mathcal{D}}_{h'} = \alpha^{\mathcal{D}}_{h'|_\sigma} \circ \hat{\alpha}^{\mathcal{D}}_{\sigma,M_1'}$ (Lemma 7.30), we arrive at our goal.

**Model morphism reducts preserve identity model morphisms** Given CSP-CASL signature morphism $\theta = (\sigma, \nu) : \Sigma_{cc} \to \Sigma'_{cc}$ and a $\Sigma'_{cc}$-model $(M', I')$, we show $(Id_{(M',I')})|_\theta = Id_{((M',I')|_\theta)}$.

$$
\begin{array}{rcll}
(Id_{(M',I')})|_\theta & = & (Id_{M'})|_\theta & \text{By definition of } Id_{(M',I')}. \\
& = & (Id_{M'})|_\sigma & \text{By definition of model morphism reduct.} \\
& = & Id_{(M'|_\sigma)} & \text{As } (Id_{M'})|_\sigma \text{ is an identity model} \\
& & & \text{morphism in } \mathit{ResSubPCFOL}^=. \\
& = & Id_{(M'|_\sigma, I'|_\theta)} & \text{By definition of } Id_{(M'|_\sigma, I'|_\theta)}. \\
& = & Id_{((M',I')|_\theta)} & \text{By definition of model reduct.}
\end{array}
$$

**Model morphism reducts preserve composition of model morphisms** This fact follows directly from the fact that *ResSubPCFOL*$^=$ reducts preserve composition of model morphisms. □

We have just proven that $\mathbf{mod}(\theta)$ is a functor for each CSP-CASL signature morphism $\theta$. We still have to prove that **mod** is itself a functor, as required by the institutional framework.

**Theorem 8.6** **mod** is a contravariant functor from CSPCASLSIG $^{op}$ to **CAT**.

*Proof.* We prove that **mod** preserves identity morphisms and composition of signature morphisms.

**Identity morphisms** Let $Id_{\Sigma_{CC}} = (Id_{\Sigma_{Data}}, Id_{\Sigma_{Proc}})$ be the identity signature morphism for the signature $\Sigma_{CC}$. We must show $\mathbf{mod}(Id_{\Sigma_{CC}}) = Id_{\mathbf{mod}(\Sigma_{CC})}$. This has two sub-proofs, one for models and one for model morphisms:

1. $(M, I)|_{Id_{\Sigma_{CC}}} = (M, I)$ for all $\Sigma_{CC}$-models $(M, I)$, and
2. $h|_{Id_{\Sigma_{CC}}} = h$ for all $\Sigma_{CC}$-model morphisms $h : (M_1, I_1) \to (M_2, I_2)$.

The first point follows from definitions and the facts that our contravariant domain translations and the lifting of reducts to alphabet translations preserve identities (Lemmas 7.13 and 7.28). The second point follows directly from definitions and the fact that *ResSubPCFOL*$^=$ model morphisms preserves identities (Theorem 4.5).

**Composition of signature morphisms** Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ and $\theta' = (\sigma', \nu') : \Sigma'_{CC} \to \Sigma''_{CC}$ be CSP-CASL signature morphisms. We must show $\mathbf{mod}(\theta' \circ \theta) = \mathbf{mod}(\theta) \circ \mathbf{mod}(\theta')$. This has two sub-proofs, one for models and one for model morphisms:

1. $(M'', I'')|_{(\theta' \circ \theta)} = ((M'', I'')|_{\theta'})|_\theta$ for all $\Sigma''_{CC}$-models $(M'', I'')$, and
2. $h''|_{(\theta' \circ \theta)} = (h''|_{\theta'})|_\theta$ for all $\Sigma''_{CC}$-model morphisms $h'' : (M''_1, I''_1) \to (M''_2, I''_2)$.

With regards to the first point, we know from the fact that *ResSubPCFOL*$^=$ is an institution (Theorem 4.5) that $M''|_{(\sigma' \circ \sigma)} = M''|_{\sigma'}|_\sigma$, thus we just have to show $I''|_{(\theta' \circ \theta)} = (I''|_{\theta'})|_\theta$. This follows from our definitions and the facts that our contravariant domain translations and the lifting of reducts to alphabet translations preserve composition of

signature morphisms (Lemmas 7.13 and 7.28). The second point follows directly from definitions and the fact that the *ResSubPCFOL*$^=$ model functor preserves composition of signature morphisms (Theorem 4.5). □

This concludes the construction of the models and various reducts for each of the three CSP-CASL institutions. Next, we focus on sentences followed by the satisfaction relation.

### 8.1.4 Sentences

We now define the set of sentences $\mathbf{sen}(\Sigma_{CC})$ for each CSP-CASL signature $\Sigma_{CC}$ and the sentence translation function $\mathbf{sen}(\theta) : \mathbf{sen}(\Sigma_{CC}) \to \mathbf{sen}(\Sigma'_{CC})$ induced by the CSP-CASL signature morphism $\theta : \Sigma_{CC} \to \Sigma'_{CC}$, that is, the functor **sen**. Throughout this section, let $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$ be a CSP-CASL signature with sort set $S$ and process part $\Sigma_{Proc} = (N_{w,comms})_{w \in S^*, comms \in S_\downarrow}$. Finally, let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ be a CSP-CASL signature morphism.

In order to define CSP-CASL sentences over the signature $\Sigma_{CC}$, we first define the notion of CSP-CASL process terms. CSP-CASL Process terms are CSP terms which are built relative to a CSP-CASL signature and two variable systems (see Section 4.2.3): global variables (represented as $X_G$) and local variables (represented as $X_L$). Global variables are used as parameters to process names, whereas local variables are formed within a process term via certain CSP operators. For example, CSP's internal prefix choice operator introduces a new local variable.

The set of process terms $ProcTerms_{\Sigma_{CC}(X_G,X_L)}$ of the CSP-CASL signature $\Sigma_{CC}$ over variable systems $X_G$ and $X_L$ is defined first using a grammar, then static semantics to deal with bindings, finally, natural semantics [Kah87] to ensure that all variables are bound appropriately. This construction is detailed in [Gim08]. In short, process terms are normal CSP processes (see Section 2.1) with recursion where communications are valid *ResSubPCFOL*$^=$ terms over signature $\Sigma_{Data}$ with variables $X_G \cup X_L$. Formulae (occurring in a conditional process expression) are data-logic formulae over signature $\Sigma_{Data}$ with variables $X_G \cup X_L$.[1]

Here, we sketch the construction of process terms. In the following we extend the original notion of process terms (from CSP-CASL in 2006) by additional constructions allowing communications to range over only the defined values of sorts. We use the syntax $s_{def}$ as a variation on certain constructions which when evaluated with the semantics will yield only the defined elements of sort $s$ (after embedding into the alphabet). $s_{def}$ represents the restriction of sort $s$ to defined elements only, and is only allowed when $s$ is a sort in the signature. $s_{def}$ is not a sort itself. The set of process terms $ProcTerms_{\Sigma_{CC}(X_G,X_L)}$ is inductively defined using only the following rules:

- SKIP, STOP and DIV are process terms.
- $t \to pt$ is a process term if $t$ is a *ResSubPCFOL*$^=$ term over signature $\Sigma_{Data}$ using variables $X_G \cup X_L$ and $pt$ is a process term in $ProcTerms_{\Sigma_{CC}(X_G,X_L)}$.

[1]Note: Where $X_G$ and $X_L$ both contain the same variable name, the union $X_G \cup X_L$ takes the local version in $X_L$. This allows us to locally overwrite global variables.

- $\sqcap x :: s \to pt$, $\sqcap x :: s_{def} \to pt$, $\Box x :: s \to pt$ and $\Box x :: s_{def} \to pt$ are process terms if $s$ is a sort in $\Sigma_{CC}$ and $pt$ is a process term in $ProcTerms_{\Sigma_{CC}(X_G, X_L \cup \{x:s\})}$.

- $pt_1 \fatsemi pt_2$, $pt_1 \sqcap pt_2$, $pt_1 \Box pt_2$, $pt_1 \parallel pt_2$, and $pt_1 \mid\mid\mid pt_2$ are process terms if $pt_1$ and $pt_2$ are process terms in $ProcTerms_{\Sigma_{CC}(X_G, X_L)}$.

- $pt_1 \,|[\, s_1, \ldots, s_k \,]|\, pt_2$ is a process term if for each $s_i$ either $s_i = t$ for some sort $t \in S$ or $s_i = t_{def}$ for some sort $t \in S$ $(i = 1 \ldots k)$ and $pt \in ProcTerms_{\Sigma_{CC}(X_G, X_L)}$ is a process term.

- $pt_1 \,|[\, s_1, \ldots, s_k \mid t_1, \ldots, t_l \,]|\, pt_2$ is a process term if for each $s_i$ either $s_i = u$ for some sort $u \in S$ or $s_i = u_{def}$ for some sort $u \in S$ $(i = 1 \ldots k)$, and for each $t_j$ either $t_j = u$ for some sort $u \in S$ or $t_j = u_{def}$ for some sort $u \in S$ $(j = 1 \ldots l)$, and $pt_i$ $(i = 1, 2)$ is a process term in $ProcTerms_{\Sigma_{CC}(X_G, X_L)}$.

- $pt \setminus s_1, \ldots, s_k$ is a process term if for each $s_i$ either $s_i = t$ for some sort $t \in S$ or $s_i = t_{def}$ for some sort $t \in S$ $(i = 1 \ldots k)$, and $pt \in ProcTerms_{\Sigma_{CC}(X_G, X_L)}$ is a process term.

- $pt[R]$ is a process term if $R$ is a renaming (that is, a list of freely mixed binary predicate symbols and unary function symbols from $\Sigma_{Data}$) and $pt$ is a process term in $ProcTerms_{\Sigma_{CC}(X_G, X_L)}$.

- if $\varphi$ then $pt_1$ else $pt_2$ is a process term if $\varphi$ is a formula in the data-logic over signature $\Sigma_{Data}$ using variable system $X_G \cup X_L$ and $pt_1$ and $pt_2$ are process terms in $ProcTerms_{\Sigma_{CC}(X_G, X_L)}$.

- $n_{w,comms}(t_1, \ldots, t_k)$ is a process term if $n$ is a process name in $N_{w,comms}$ for some $w = \langle s_1, \ldots, s_k \rangle \in S^*$ and $comms \in S_{\downarrow}$ and where $t_i$ is a ***ResSubPCFOL***$^{=}$ term of sort $s_i$ over signature $\Sigma_{Data}$ using variables $X_G \cup X_L$ $(i = 1 \ldots k)$.

Above we only sketched the construction and omitted some required restrictions. For example, one such restriction is for the process term $pt_1 \,|[\, s_1, \ldots, s_k \mid t_1, \ldots, t_l \,]|\, pt_2$ to be well formed we require $\{s_1, \ldots, s_k\} \subseteq {\downarrow} sorts(pt_1)$ and $\{t_1, \ldots, t_k\} \subseteq {\downarrow} sorts(pt_2)$, where $sorts(pt)$ is the constituent alphabet of the process term $pt$, that is, the set consisting of all the possible communication sorts of $pt$. This is defined recursively of the structure of the process term $pt$ and simply collects all the sort symbols that are used. Full details can be found in [Gim08].

We now define process sentences by building upon the notion of process terms. A process sentence over $\Sigma_{CC}$ is an equation of the form

$$n_{w,comms}(x_1, \ldots, x_k) = pt$$

where $n_{w,comms}$ is a process name in $N_{w,comms}$, $pt \in ProcTerms_{\Sigma_{CC}(X_G, \emptyset)}$ is a process term, $w = \langle s_1, \ldots, s_k \rangle$, and $X_G = \{x_1 : s_1, \ldots, x_k : s_k\}$ under the conditions that the constituent alphabet of $pt$ is contained within the set $comms$ (i.e., $sorts(pt) \subseteq comms$), and $x_i \neq x_j$ for $1 \leqslant i, j \leqslant k$ such that $i \neq j$.

Finally, we define the set of CSP-CASL sentences over signature $\Sigma_{CC}$ to be the set containing all process sentences over $\Sigma_{CC}$ and all *ResSubPCFOL*$^{=}$ formulae over $\Sigma_{Data}$ (data sentences). In terms of the institution framework this is the set $\mathbf{sen}(\Sigma_{CC})$.

Data sentence translation is inherited from *ResSubPCFOL*$^{=}$. In order to define the translation of process sentences along a signature morphism, we first define translation of process terms as presented by Kahsai [Kah10]. Given a process term over the CSP-CASL signature $\Sigma_{CC}$ we inductively define the translated process term along the signature morphism $\theta$ as:

$$
\begin{array}{lcl}
\theta(\text{SKIP}) & := & \text{SKIP} \\
\theta(\text{STOP}) & := & \text{STOP} \\
\theta(\text{DIV}) & := & \text{DIV} \\
\theta(t \rightarrow pt) & := & \sigma(t) \rightarrow \theta(pt) \\
\theta(\sqcap x :: s \rightarrow pt) & := & \sqcap x :: \sigma(s) \rightarrow \theta(pt) \\
\theta(\Box x :: s \rightarrow pt) & := & \Box x :: \sigma(s) \rightarrow \theta(pt) \\
\theta(pt_1 \fatsemi pt_2) & := & \theta(pt_1) \fatsemi \theta(pt_2) \\
\theta(pt_1 \sqcap pt_2) & := & \theta(pt_1) \sqcap \theta(pt_2) \\
\theta(pt_1 \Box pt_2) & := & \theta(pt_1) \Box \theta(pt_2) \\
\theta(pt_1 \parallel pt_2) & := & \theta(pt_1) \parallel \theta(pt_2) \\
\theta(pt_1 \mid\mid\mid pt_2) & := & \theta(pt_1) \mid\mid\mid \theta(pt_2) \\
\theta(pt_1 \,|[\, s_1, \ldots, s_k \,]|\, pt_2) & := & \theta(pt_1) \,|[\, \sigma(s_1), \ldots, \sigma(s_k) \,]|\, \theta(pt_2) \\
\theta(pt_1 \,|[\, s_1, \ldots, s_k \mid t_1, \ldots, t_l \,]|\, pt_2) & := & \theta(pt_1) \,|[\, \sigma(s_1), \ldots, \sigma(s_k) \mid \\
 & & \qquad \sigma(t_1), \ldots, \sigma(t_l) \,]|\, \theta(pt_2) \\
\theta(pt \setminus s_1, \ldots, s_k) & := & \theta(pt) \setminus \sigma(s_1), \ldots, \sigma(s_k) \\
\theta(pt[r_1, \ldots, r_k]) & := & \theta(pt)[\sigma(r_1), \ldots, \sigma(r_k)] \\
\theta(\text{if } \varphi \text{ then } pt_1 \text{ else } pt_2) & := & \text{if } \sigma(\varphi) \text{ then } \theta(pt_1) \text{ else } \theta(pt_2) \\
\theta(n_{w,comms}(t_1, \ldots, t_k)) & := & \nu_{w,comms}(n_{w,comms})(\sigma(t_1), \ldots, \sigma(t_k))
\end{array}
$$

where $\sigma(s_{def}) := \sigma(s)_{def}$.

Finally, we define CSP-CASL sentence translation along the CSP-CASL signature morphism $\theta$ as

$$
\mathbf{sen}(\theta)(\varphi) := \begin{cases} \sigma(\varphi) & \text{if } \varphi \text{ is a } \textit{ResSubPCFOL}^{=} \text{ sentence.} \\ n'(\sigma(x_1), \ldots, \sigma(x_k)) = \theta(pt) & \text{if } \varphi \text{ is a process sentence of the form} \\ & n_{w,comms}(x_1, \ldots, x_k) = pt \text{ and} \\ & \nu_{w,comms}(n_{w,comms}) = n'. \end{cases}
$$

As is common when working with institutions we write $\mathbf{sen}(\theta)(\varphi)$ simply as $\theta(\varphi)$.

**Theorem 8.7** CSP-CASL sentences and sentence translation forms a functor, that is, $\mathbf{sen}(\theta) : \mathbf{sen}(\Sigma_{CC}) \rightarrow \mathbf{sen}(\Sigma'_{CC})$ is a functor.

*Proof.* This follows from definitions and the fact that the CASL sentence translation is a functor. □

This concludes the construction of CSP-CASL sentences and sentence translations. Next, we define the CSP-CASL satisfaction relation.

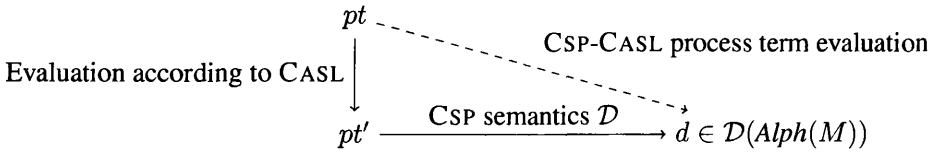



Figure 8.1: Overview of the two phase evaluation of CSP-CASL process terms.

### 8.1.5 Satisfaction

We now define what it means for CSP-CASL models to satisfy CSP-CASL sentences. Throughout this section let $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$ be a CSP-CASL signature with data part $\Sigma_{Data} = (S, TF, PF, P, \leq)$ and process part $\Sigma_{Proc} = (N_{w,comms})_{w \in S^*, comms \in S_\downarrow}$. Furthermore, let $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$ be a fixed CSP semantics, and let $(M, I)$ be $\Sigma_{CC}$- model.

For data sentences we will inherit satisfaction from the *ResSubPCFOL*$^=$ institution (this will be described in Section 8.1.5.3). For process sentences we define satisfaction via a two phase evaluation. Figure 8.1 illustrates the phases. We first evaluate process terms in process sentences according to the CASL semantics where we slightly modify the standard definition of "concatenating a substitution to a valuation". This evaluation effectively replaces all CASL elements in process terms by their evaluations, that is, elements of the alphabet. The resulting process terms ($pt'$ in Figure 8.1) are normal CSP process terms over the alphabet $Alph(M)$. Once this first phase is completed, we can apply the normal CSP semantics.

We now describe these phases in detail and give the formal definition of satisfaction.

#### 8.1.5.1 Phase 1: Evaluation According to CASL

The goal of this phase (originally described by Roggenbach [Rog06]) is to replace CASL elements in the process term of process sentences with elements of the alphabet $Alph(M)$. This evaluation replaces:

- sort symbols in internal and external prefix choice operators, various parallel operators and hiding operators with the embeddings of their carrier sets into the alphabet.
- unary function and binary predicate symbols in renamings with sets of pairs of alphabet elements that are mapped to each other via the function and predicate symbols.
- CASL terms in action prefix operators and process names with their evaluation after embedding into the alphabet.
- CASL formulae in conditional statements with their evaluated truth values.

Let $X_G$ and $X_L$ be global and local variable systems respectively. Furthermore, let $\mu_G : X_G \to M_\perp$ and $\mu_L : X_L \to M_\perp$ be global and local variable valuations, respectively, for the *ResSubPCFOL*$^=$ model $M$. The evaluation of CASL elements occurring in process terms in $ProcTerms_{\Sigma_{CC}(X_G, X_L)}$, with respect to *ResSubPCFOL*$^=$ model $M$ and valuations $\mu_G$ and $\mu_L$, is defined as :

- $[\![s]\!]_{M,\mu_G,\mu_L} := \overline{s}_M$ for all sort symbols $s \in S$.
- $[\![s_{def}]\!]_{M,\mu_G,\mu_L} := \overline{s}_M \setminus \{\overline{\perp^s_M}\}$ for all sort symbols $s \in S$.
- $[\![p_{\langle s_1,s_2\rangle}]\!]_{M,\mu_G,\mu_L} := \{(\overline{x}^{s_1}_M, \overline{y}^{s_2}_M) \mid (x,y) \in (p_{\langle s_1,s_2\rangle})_{M_\perp}\}$ for all binary predicate symbols $p_{\langle s_1,s_2\rangle} \in P$, $s_1, s_2 \in S$.
- $[\![f_{\langle s_1\rangle,s_2}]\!]_{M,\mu_G,\mu_L} := \{(\overline{x}^{s_1}_M, \overline{y}^{s_2}_M) \mid (f_{\langle s_1\rangle,s_2})_{M_\perp}(x) = y\}$ for all unary function symbols $f_{\langle s_1\rangle,s_2} \in TF \cup PF$, $s_1, s_2 \in S$.
- $[\![t]\!]_{M,\mu_G,\mu_L} := \overline{(\mu_G \cup \mu_L)^\#_s(t)_M}^s$ for all *ResSubPCFOL*$^=$ terms $t \in T_{\Sigma_{Data}}(X_G \cup X_L)_s$.
- $[\![\varphi]\!]_{M,\mu_G,\mu_L} := \begin{cases} true & \text{if } \mu_G \cup \mu_L \models \varphi \\ false & \text{if not } \mu_G \cup \mu_L \models \varphi \end{cases}$
  for all formulae $\varphi$ over $\Sigma_{Data}(X_G \cup X_L)$.

Here, the variable valuation $\mu_L$ deals with local variable bindings generated by the CSP evaluation in Phase 2. $\mu_G \cup \mu_L$ is the variable valuation from $X_G \cup X_L \to M_\perp$, where we take priority to local variables both in the union of the valuations and the union of the variable systems. $(\mu_G \cup \mu_L)^\#$ is the extension of $\mu_G \cup \mu_L$ to terms. As variable valuations map into the carrier sets of the totalised CASL models, this evaluation yields either carrier set elements or $\perp$ for undefined elements. In either case, the result can be embedded into the alphabet $Alph(M)$.

Figure 8.2 defines the CASL evaluation of process terms according to *ResSubPCFOL*$^=$ model $M$ and valuations $\mu_G$ and $\mu_L$. Substitutions are the way in which the various CSP semantics model the binding concept within the CSP prefix operators. Therefore we require a local variable valuation to model the CSP binding concept, which is used in Phase 2 (Section 8.1.5.2). The CSP semantics will bind alphabet elements to variable names. This is done by appending a substitution to a variable valuation. However, our valuations map into the totalised carrier sets and not the alphabet. Thus, when the CSP semantics tries to bind a variable to an alphabet element, we first need to unpack the alphabet element to produce a totalised carrier set element, and then bind this totalised carrier set element to the variable instead. This binding of local variables occurs only in the CSP internal and external prefix choice operators. Thus, the clauses (in Figure 8.2) for these operators take a substitution as the argument:

$$[\![\_]\!]_{M,\mu_G,(\lambda z.\mu_L)}[a/x] := [\![\_]\!]_{M,\mu_G,(\mu_L[unpack_s(a)/x])}$$

where $x$ is a variable of sort $s$. Here, $\mu_L[a'/x](y) := \mu_L(y)$ for $y \neq x$ and $\mu_L[a'/x](y) := a'$ for $y = x$. The *unpack* function reverses the alphabet construction and produces a carrier set element (or $\perp$) of the correct sort. For instance, let $\Sigma_{CC}$ be the signature with two sorts $s$ and $t$ such that $s \leq t$, let $x : s$ be a variable and let $a = [(t, y_t)]_{\sim M} = \{(s, y_s), (t, y_t)\}$ be an alphabet element. If we bind $a$ to the variable $x : s$, then $unpack_s(a)$ would yield the carrier set element $y_s$. It is this $y_s$ that is bound to the variable $x$.

### 8.1.5.2 Phase 2: Evaluation according to the CSP Semantics

The process term $pt'$ (see Figure 8.1) is produced from the process term $pt$ by applying the evaluation according to the CASL semantics. This process term $pt'$ is an ordinary CSP process

$$
\begin{array}{lcl}
[\![\text{SKIP}]\!]_{M,\mu_G,\mu_L} & := & \text{SKIP} \\
[\![\text{STOP}]\!]_{M,\mu_G,\mu_L} & := & \text{STOP} \\
[\![\text{DIV}]\!]_{M,\mu_G,\mu_L} & := & \text{DIV} \\
[\![t \rightarrow pt]\!]_{M,\mu_G,\mu_L} & := & [\![t]\!]_{M,\mu_G,\mu_L} \rightarrow [\![pt]\!]_{M,\mu_G,\mu_L} \\
[\![\sqcap x :: s \rightarrow pt]\!]_{M,\mu_G,\mu_L} & := & \sqcap x :: [\![s]\!]_{M,\mu_G,\mu_L} \rightarrow [\![pt]\!]_{M,\mu_G,(\lambda z.\mu_L)} \\
[\![\Box x :: s \rightarrow pt]\!]_{M,\mu_G,\mu_L} & := & \Box x :: [\![s]\!]_{M,\mu_G,\mu_L} \rightarrow [\![pt]\!]_{M,\mu_G,(\lambda z.\mu_L)} \\
[\![pt_1 \fatsemi pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \fatsemi [\![pt_2]\!]_{M,\mu_G,\emptyset} \\
[\![pt_1 \sqcap pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \sqcap [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![pt_1 \Box pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \Box [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![pt_1 \parallel pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \parallel [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![pt_1 \mid\mid\mid pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \mid\mid\mid [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![pt_1 \,|[\, s_1, \ldots, s_k \,]|\, pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \,|[\, [\![s_1]\!]_{M,\mu_G,\mu_L} \cup \ldots \cup \\
 & & [\![s_k]\!]_{M,\mu_G,\mu_L} \,]|\, [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![pt_1 \,|[\, s_1, \ldots, s_k \mid t_1, \ldots, t_l \,]|\, pt_2]\!]_{M,\mu_G,\mu_L} & := & [\![pt_1]\!]_{M,\mu_G,\mu_L} \,|[\, [\![s_1]\!]_{M,\mu_G,\mu_L} \cup \ldots \cup \\
 & & [\![s_k]\!]_{M,\mu_G,\mu_L} \mid [\![t_1]\!]_{M,\mu_G,\mu_L} \cup \ldots \cup \\
 & & [\![t_l]\!]_{M,\mu_G,\mu_L} \,]|\, [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![pt \setminus s_1, \ldots, s_k]\!]_{M,\mu_G,\mu_L} & := & [\![pt]\!]_{M,\mu_G,\mu_L} \setminus [\![s_1]\!]_{M,\mu_G,\mu_L} \cup \ldots \cup \\
 & & [\![s_k]\!]_{M,\mu_G,\mu_L} \\
[\![pt[r_1, \ldots, r_k]]\!]_{M,\mu_G,\mu_L} & := & [\![pt]\!]_{M,\mu_G,\mu_L} [[\![r_1]\!]_{M,\mu_G,\mu_L} \cup \ldots \cup \\
 & & [\![r_k]\!]_{M,\mu_G,\mu_L}] \\
[\![\text{if } \varphi \text{ then } pt_1 \text{ else } pt_2]\!]_{M,\mu_G,\mu_L} & := & \text{if } [\![\varphi]\!]_{M,\mu_G,\mu_L} \text{ then } [\![pt_1]\!]_{M,\mu_G,\mu_L} \\
 & & \text{else } [\![pt_2]\!]_{M,\mu_G,\mu_L} \\
[\![n_{w,comms}(t_1, \ldots, t_k)]\!]_{M,\mu_G,\mu_L} & := & n_{w,comms}([\![t_1]\!]_{M,\mu_G,\mu_L}, \ldots, \\
 & & [\![t_k]\!]_{M,\mu_G,\mu_L})
\end{array}
$$

Figure 8.2: Evaluation according to CASL. $M$ is a *ResSubPCFOL*$^=$ model, $\mu_G : X_G \rightarrow M_\perp$ and $\mu_L : X_L \rightarrow M_\perp$ are global and local variable valuations, respectively. The result of this evaluation is a standard CSP process over $Alph(M)$.

over the alphabet of communications $Alph(M)$. We can now apply our chosen CSP semantics $\mathcal{D}$ to it, that is, we apply the the function

$$[\![\text{-}]\!]_{\mathcal{D},I}$$

to the process term $pt'$. The function $[\![\text{-}]\!]_{\mathcal{D},I}$ is defined as:

$$
\begin{array}{lcl}
[\![pt]\!]_{\mathcal{T},I} & := & traces_I(pt) \\
[\![pt]\!]_{\mathcal{N},I} & := & (failures_{\perp,I}(pt), divergences_I(pt)) \\
[\![pt]\!]_{\mathcal{F},I} & := & (traces_I(pt), failures_I(pt))
\end{array}
$$

where $pt$ is a process term and $traces_I$, $failures_{\perp,I}$, $failures_I$, and $divergences_I$ are the normal CSP clauses (see Appendix B) augmented with the process interpretation map $I$. The augmentation with the process interpretation map $I$ is required to give interpretations to process names

occurring within process expressions. We sketch this construction for the $traces_I$ function.

$$
\begin{array}{lcl}
traces_I(\text{SKIP}) & := & \{\langle\rangle, \langle\checkmark\rangle\} \\
traces_I(\text{STOP}) & := & \{\langle\rangle\} \\
traces_I(\text{DIV}) & := & \{\langle\rangle\} \\
traces_I(a \rightarrow P) & := & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces_I(P)\} \\
traces_I(\sqcap x :: X \rightarrow P) & := & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces_I(P[a/x]), a \in X\} \\
traces_I(\Box x :: X \rightarrow P) & := & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces_I(P[a/x]), a \in X\} \\
\vdots & \vdots & \vdots \\
traces_I(\text{if } \varphi \text{ then } P \text{ else } Q) & := & \begin{cases} traces_I(P) & \text{if } \varphi \text{ evaluates to } true \\ traces_I(Q) & \text{if } \varphi \text{ evaluates to } false \end{cases} \\
traces_I(n_{w,comms}) & := & I_{w,comms}(n_{w,comms})
\end{array}
$$

The additional clauses for the other functions are:

$$
\begin{array}{lcl}
failures_{\perp,I}(n_{w,comms}) & := & fst(I(n_{w,comms})) \\
failures_I(n_{w,comms}) & := & snd(I(n_{w,comms})) \\
divergences_I(n_{w,comms}) & := & snd(I(n_{w,comms}))
\end{array}
$$

Here, we need the projection functions *fst* and *snd*, as the process interpretation map $I$ produces process denotations (i.e., pairs for the Failures/Divergences and Stable-Failures semantics).

The result of this second phase is a denotation within the CSP domain $\mathcal{D}$ over the alphabet of the *ResSubPCFOL*$^=$ model $M$, that is, a denotation $d \in \mathcal{D}(Alph(M))$.

#### 8.1.5.3 Satisfaction of CSP-CASL Sentences

We are now ready to define satisfaction of CSP-CASL sentences with respect to CSP-CASL models. Satisfaction of data sentences with respect to $\Sigma_{CC}$-models is inherited from the *ResSubPCFOL*$^=$ institution (see Section 4.4), that is,

$$(M, I) \models_{\Sigma_{CC}} \varphi \text{ iff } M \models_{\Sigma_{Data}} \varphi$$

for all $\Sigma_{CC}$-models $(M, I)$ and *ResSubPCFOL*$^=$ sentences $\varphi$, where $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$.

Satisfaction of a process sentence $n(x_1, \ldots, x_k) = pt$ over signature $\Sigma_{CC}$ (where $n : w, comms$, and $w = \langle s_1, \ldots, s_k\rangle$) and global variable system $X_G$ with respect to $\Sigma_{CC}$-model $(M, I)$ is defined as follows:

$$
\begin{array}{c}
(M, I) \models_{\Sigma_{CC}} n(x_1, \ldots, x_k) = pt \\
\text{if and only if} \\
\text{for all variable valuations } \mu_G : X_G \rightarrow M_\perp \text{ it holds that} \\
I_{w,comms}(n(\overline{\mu_G(x_1)}^{s_1}_M, \ldots, \overline{\mu_G(x_k)}^{s_k}_M)) = [\![[\![pt]\!]_{M,\mu_G,\emptyset}]\!]_{\mathcal{D},I} \ .
\end{array}
$$

The interpretation of the process name $n$, after variables have been evaluated and embedded into the alphabet, must equal the evaluation of the process term $pt$ with respect to the model and variable valuations.

This concludes the definition of the satisfaction relation for CSP-CASL. We will now establish the *reduct property* that will be the key fact in order to prove the *satisfaction condition*.

To this end, we first establish the relationship between variable valuations and model reducts (presented from [Rog06]).

Let $(M', I')$ be a $\Sigma'_{CC}$-model and $\mu : \sigma(X) \to M'_\perp$ be a valuation, where $\sigma(X)$ is the translation of the variable system $X$ as presented in Section 4.2.3. We define the valuation

$$\begin{array}{rccl} \mu|_\sigma : & X & \to & (M'|_\sigma)_\perp \\ & x : s & \mapsto & \mu_{\sigma^S(s)}(x : \sigma^S(s)) \end{array}$$

**Lemma 8.8** Let $(M', I')$ be a $\Sigma'_{CC}$-model and $\mu : \sigma(X) \to M'_\perp$ be a valuation. Then $\mu$ and $\mu|_\sigma$ are in a one-one correspondence.

*Proof.* See Lemma 5 in [Rog06]. The proof is straightforward and involves induction over the structure of terms. □

We now prove two lemmas which will shortly be used to extend Kahsai's proof [Kah10] of the reduct property to cover our construction for CSP-CASL.

The following lemma states that the translation of defined sort symbols interacts well with the CASL evaluation phase and model reducts.

**Lemma 8.9** Let $\sigma : \Sigma \to \Sigma'$ be a *ResSubPCFOL*$^=$ signature morphism, $M'$ be a $\Sigma'$-model and $X_G$ and $X_L$ be global and local variable systems respectively. Let $\mu_G : \sigma(X_G) \to M'_\perp$ and $\mu_L : \sigma(X_L) \to M'_\perp$ be valuations. Then the following holds

$$\alpha^{\mathcal{P}}_{\sigma,M'}([\![s_{def}]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}) = [\![\sigma(s)_{def}]\!]_{M',\mu_G,\mu_L}$$

where $\alpha^{\mathcal{P}}_{\sigma,M'}$ is the direct image of $\alpha_{\sigma,M'}$ (i.e., applied pointwise to sets).

*Proof.* We show subset inclusion in both directions. First, we show the '$\subseteq$' direction. Let $a' \in \alpha^{\mathcal{P}}_{\sigma,M'}([\![s_{def}]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma})$, then there exists $a \in [\![s_{def}]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}$ such that $\alpha_{\sigma,M'}(a) = a'$. Therefore, there exists $x \in ((M'|_\sigma)_\perp)_s$ such that $x \neq \perp$ and $a = [(s,x)]_{\sim_{M'|_\sigma}}$. As $a' = \alpha_{\sigma,M'}([(s,x)]_{\sim_{M'|_\sigma}}) = [(\sigma(s),x)]_{\sim_{M'}}$ and $x \neq \perp$ we know $a' \in [\![\sigma(s)_{def}]\!]_{M',\mu_G,\mu_L}$. The other direction is similar. □

**Lemma 8.10** (Reduct Property for process names) Let $X_G$ and $X_L$ be global and local variable systems respectively. Let $(M', I')$ be a $\Sigma'_{CC}$-model and $\mu_G : \sigma(X_G) \to M'_\perp$ and $\mu_L : \sigma(X_L) \to M'_\perp$ be valuations. Finally, let $n(t_1, \ldots, t_k) \in ProcTerms_{\Sigma_{CC}(X_G,X_L)}$ be a process term where $n$ is a process name and $t_1, \ldots, t_k$ are appropriate *ResSubPCFOL*$^=$ terms. Then

$$[\![[\![n(t_1, \ldots, t_k)]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}]\!]_{\mathcal{D},I'|_\theta} = \hat{\alpha}^{\mathcal{D}}_{\sigma,M'}([\![[\![\theta(n(t_1, \ldots, t_k))]\!]_{M',\mu_G,\mu_L}]\!]_{\mathcal{D},I'})$$

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

*Proof.* We show the equality by unfolding definitions. We start with the left hand side. By unfolding CSP-CASL's semantics this is equal to

$$I'|_\theta(n([\![t_1]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}, \ldots, [\![t_k]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma})) \ .$$

By the definition of CSP-CASL model reducts this is then equal to

$$\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(I'(\nu(n)(\alpha_{\sigma,M'}([\![t_1]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}),\ldots,\alpha_{\sigma,M'}([\![t_k]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma})))) \ .$$

As $\alpha_{\sigma,M'}([\![t_i]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}) = [\![\sigma(t_i)]\!]_{M',\mu_G,\mu_L}$, by substitution this is equal to

$$\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(I'(\nu(n)([\![\sigma(t_1)]\!]_{M',\mu_G,\mu_L},\ldots,[\![\sigma(t_k)]\!]_{M',\mu_G,\mu_L}))) \ .$$

By the definition of CSP-CASL's semantics this is equal to

$$\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}([\![[\![\nu(n)(\sigma(t_1),\ldots,\sigma(t_k))]\!]_{M',\mu_G,\mu_L}]\!]_{\mathcal{D},I'}) \ .$$

Finally, by the definition of CSP-CASL sentence translation this is equal to

$$\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}([\![[\![\theta(n(t_1,\ldots,t_k))]\!]_{M',\mu_G,\mu_L}]\!]_{\mathcal{D},I'}) \ ,$$

that is, the right hand side. □

**Lemma 8.11** (Reduct Property) Let $X_G$ and $X_L$ be global and local variable systems respectively. Let $(M', I')$ be a $\Sigma'_{CC}$-model and $\mu_G : \sigma(X_G) \to M'_\perp$ and $\mu_L : \sigma(X_L) \to M'_\perp$ be valuations. Finally, let $pt$ be a process term in $ProcTerms_{\Sigma_{CC}(X_G,X_L)}$. Then

$$[\![[\![pt]\!]_{M'|_\sigma,\mu_G|_\sigma,\mu_L|_\sigma}]\!]_{\mathcal{D},I'|_\theta} = \hat{\alpha}^{\mathcal{D}}_{\sigma,M'}([\![[\![\theta(pt)]\!]_{M',\mu_G,\mu_L}]\!]_{\mathcal{D},I'})$$

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

*Proof.* Kahsai's proof of the reduct property [Kah10] can be extended to this setting in a straightforward way. An extra base case is required (Lemma 8.10) to deal with process names and Lemma 8.9, which relates the CASL evaluation and the sentence translation of defined sort symbols . □

The reduct property allows us to establish the satisfaction condition.

**Theorem 8.12** (Satisfaction Condition) Let $(M', I')$ be a $\Sigma'_{CC}$-model and let $\varphi$ be a $\Sigma_{CC}$-sentence, then the following holds:

$$(M', I') \models_{\Sigma'_{CC}} \theta(\varphi) \iff (M', I')|_\theta \models_{\Sigma_{CC}} \varphi \ .$$

*Proof.* The proof is the same as Kahsai's proof [Kah10] except that we apply our reduct property (Lemma 8.11) instead of Kahsai's. This change covers our extended setting with process names. □

This concludes our presentation of CSP-CASL as three institutions, one for each of the main CSP semantics. Next, we discuss how to allow generic and parametrised CSP-CASL specifications.

## 8.2 Parametrisation: Pushouts and Amalgamation

We now present results concerning pushouts and amalgamation of CSP-CASL closely following [OMR12]. The existence of pushouts and amalgamation properties shows that an institution has good modularity properties. Amalgamation is a major technical assumption in the study of specification semantics [DGS93, ST88].

We will shortly show that CSP-CASL fails to have the amalgamation property (see Section 4.7.2). In order to give CSP-CASL an amalgamation property which enables parametrisation and instantiation of CSP-CASL specifications (see Section 4.7.2), we weaken the original definition of amalgamation. What is actually required for parametrisation and instantiation of specifications is

1. a unique construction of a signature from existing signatures (obviously related via morphisms), and

2. a suitable amalgamation of models and model morphisms.

We now show that CSP-CASL's signature category fails to have pushouts and thus, CSP-CASL fails to have the amalgamation property (see Section 4.7.2). To do this, we first establish a lemma regarding CSP-CASL signature morphisms.

**Lemma 8.13** CSP-CASL signature morphisms between signatures with acyclic sub-sort relations are injective on sorts.

*Proof.* Let $\sigma(s) = \sigma(t)$. By reflexivity, $\sigma(s) \leq \sigma(t)$ and $\sigma(s) \geq \sigma(t)$. By reflection, $s \leq t$ and $s \geq t$. By acyclicity, $s = t$. □

**Lemma 8.14** CSPCASLSIG does not have pushouts.

*Proof.* As shown in Propositions 9 and 10 of [MR07], pushouts of injective sort maps (in the category of sets) exist and are injective, but pushouts in the category of sets and injective maps do not exist (even if signatures are restricted to those with discrete sub-sort relations). This is because mediating morphisms can be non-injective on sorts. By Lemma 8.13, this negative result also carries over to CSP-CASL. □

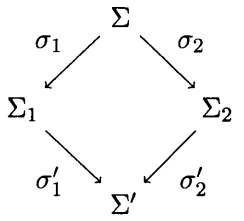
Even though CSP-CASL does not have the standard amalgamation property, we can enable parametrisation and instantiation of specifications with a weaker property, namely, *quasi-amalgamation.*

An institution $(\mathbf{SIGN}, \mathbf{sen}, \mathbf{mod}, \models)$ has *quasi-amalgamation*, if there exists a category $\mathbf{SIGN}^{Rich}$ such that

1. **SIGN** is a wide sub-category[2] of $\mathbf{SIGN}^{Rich}$.

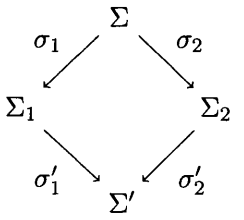
2. $\mathbf{SIGN}^{Rich}$ has pushouts. Furthermore, any pushout

[2] A category **C** is a wide sub-category of **D** if and only if **C** is a sub-category of **D**, and $|\mathbf{C}| = |\mathbf{D}|$ (i.e., **C** and **D** contain the same collection of objects).




in $\mathbf{SIGN}^{Rich}$, such that $(\Sigma, \sigma_1, \sigma_2)$ is a span in **SIGN**, leads to a commuting square in **SIGN**, that is, $\sigma'_1$ and $\sigma'_2$ are morphisms in **SIGN** and $\sigma'_1 \circ \sigma_1 = \sigma'_2 \circ \sigma_2$.

3. given any pushout




in $\mathbf{SIGN}^{Rich}$, such that $(\Sigma, \sigma_1, \sigma_2)$ is a span in **SIGN**, the following two conditions hold:

- for any two models $M_1 \in |\mathbf{mod}(\Sigma_1)|$ and $M_2 \in |\mathbf{mod}(\Sigma_2)|$ such that $M_1|_{\sigma_1} = M_2|_{\sigma_2}$, there exists a unique model $M' \in |\mathbf{mod}(\Sigma')|$ such that $M'|_{\sigma'_1} = M_1$ and $M'|_{\sigma'_2} = M_2$ (we call $M'$ the *amalgamation* of $M_1$ and $M_2$); and
- for any two model morphisms $h_1 : M_1^A \to M_1^B$ in $\mathbf{mod}(\Sigma_1)$ and $h_2 : M_2^A \to M_2^B$ in $\mathbf{mod}(\Sigma_2)$ such that $h_1|_{\sigma_1} = h_2|_{\sigma_2}$, there exists a unique model morphism $h' : M'^A \to M'^B$ in $\mathbf{mod}(\Sigma')$ such that $h'|_{\sigma'_1} = h_1$ and $h'|_{\sigma'_2} = h_2$ (we call $h'$ the *amalgamation* of $h_1$ and $h_2$).

Quasi-amalgamation is enough to allow for instantiations of structured specifications. We outline this for the case of a generic specification with a single parameter.

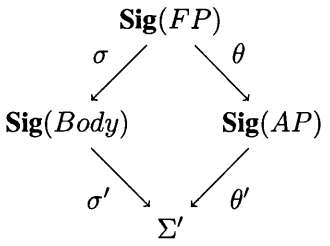
Given a named specification $SP[FP] = Body$ with signature inclusion $\sigma : \mathbf{Sig}(FP) \to \mathbf{Sig}(Body)$, an actual parameter specification $AP$, and a fitting morphism $\theta : \mathbf{Sig}(FP) \to \mathbf{Sig}(AP)$, $SP[AP \textbf{ fit } \theta]$ is a specification with

$\mathbf{Sig}(SP[AP \textbf{ fit } \theta]) \quad := \quad \Sigma'$

$\mathbf{Mod}(SP[AP \textbf{ fit } \theta]) \quad :=$

$\{M' \in |\mathbf{mod}(\Sigma')| \mid M'|_{\sigma'} \in \mathbf{Mod}(Body) \wedge M'|_{\theta'} \in \mathbf{Mod}(AP)\}$

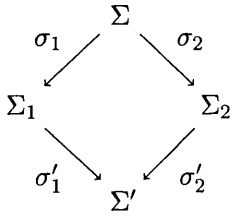
where




is the pushout in $\mathbf{SIGN}^{Rich}$ of the span $(\mathbf{Sig}(FP), \sigma, \theta)$ in **SIGN** (and thus, by the property quasi-amalgamation, the above square is a commuting square in **SIGN**).

To establish that CSP-CASL has quasi-amalgamation we first establish several lemmas.

**Lemma 8.15** Let the following be a pushout of CASL signatures:




with $\sigma_1$ and $\sigma_2$ sub-sort-reflecting, and let $\leq$, $\leq_1$, $\leq_2$ and $\leq'$ be the sub-sort relations of $\Sigma$, $\Sigma_1$, $\Sigma_2$ and $\Sigma'$, respectively. Then

$$\begin{aligned} \leq' \quad = \quad & \sigma_1'(\leq_1) \cup \sigma_2'(\leq_2) \\ \cup \quad & \{(\sigma_1'(s_1), \sigma_2'(s_2)) \mid \exists s \in \Sigma : s_1 \leq_1 \sigma_1(s), \sigma_2(s) \leq_2 s_2\} \\ \cup \quad & \{(\sigma_2'(s_2), \sigma_1'(s_1)) \mid \exists s \in \Sigma : s_2 \leq_2 \sigma_2(s), \sigma_1(s) \leq_1 s_1\} \end{aligned}$$

*Proof.* The inclusion from right to left is clear. For the converse inclusion, by the pushout property it suffices to show that the union is reflexive and transitive. Reflexivity is clear. Concerning transitivity, note that by construction of the pushout, any chaining of $\leq_1$ and $\leq_2$ must go through $\Sigma$. With this, we show that any alternating chain of $\leq_1$- and $\leq_2$-pairs of length more than two can be reduced. Indeed, let $s \leq_1 \sigma_1(u)$, $\sigma_2(u) \leq_2 \sigma_2(v)$ and $\sigma_1(v) \leq_1 t$. By reflection of $\sigma_2$, $u \leq v$, hence $\sigma_1(u) \leq \sigma_1(v)$ and by transitivity of $\leq_1$, $s \leq_1 t$. A $\leq_2$-$\leq_1$-$\leq_2$ chain is treated dually. □

As is done in [MR07], we now form the super-category $\text{CSPCASLSIG}^{plain}$ of signatures for CSP-CASL required for quasi-amalgamation (that is, $\mathbf{SIGN}^{Rich}$).

Let $\text{CSPCASLSIG}^{plain}$ be CSPCASLSIG with the reflection and weak non-extension restrictions (from *ResSubPCFOL*$^{=}$) dropped. Then we have:

**Lemma 8.16** The category $\text{CSPCASLSIG}^{plain}$ has pushouts. Furthermore, any pushout in $\text{CSPCASLSIG}^{plain}$ of a span in CSPCASLSIG leads to a commuting square in CSPCASLSIG (although not a pushout in CSPCASLSIG).

*Proof.* The category of CASL signatures has pushouts [Mos98]. We need to extend this to process names in $\text{CSPCASLSIG}^{plain}$. This is done by considering fully qualified process names, and taking their pushout in **SET**. Let $N'$ be constructed as is done in the standard CASL pushout construction (e.g., for function symbols).

Now let a pushout square be given as above, and assume that $\sigma_1$ and $\sigma_2$ are reflecting and weakly non-extending. We show that $\sigma_1'$ has these properties as well (the argument for $\sigma_2'$ is the same).

Let $\leq$, $\leq_1$, $\leq_2$ and $\leq'$ be the sub-sort relations of $\Sigma$, $\Sigma_1$, $\Sigma_2$ and $\Sigma'$ respectively. Concerning reflection, assume that $\sigma_1'(s) \leq' \sigma_1'(t)$. Lemma 8.15 tells us how pairs in the $\leq'$ relation look. Since both endpoints of our pair are in the image of $\sigma_1'$, with the method of the proof of Lemma 8.15, we can eliminate any $\sigma_2'(\leq_2)$ relation pair from the representation of $\sigma_1'(s) \leq' \sigma_1'(t)$. Hence, we arrive at $\sigma_1'(s)\sigma_1'(\leq_1)\sigma_1'(t)$ and thus $s \leq_1 t$.

Concerning weak non-extension, let $\sigma_1'(s_1) \leq' u' \geq' \sigma_1'(t_1)$. We need to show that there exists some $\bar{u_1} \in \Sigma_1$ with $s_1 \leq_1 \bar{u_1} \geq_1 t_1$ and $\sigma_1'(\bar{u_1}) \leq' u'$. By construction of the pushout, there is either some $u_1 \in \Sigma_1$ with $\sigma_1'(u_1) = u'$ (in this case, we are done), or some $u_2 \in \Sigma_2$ with $\sigma_2'(u_2) = u'$. In the latter case, since $\sigma_1'(s_1) \leq' \sigma_2'(u_2) \geq' \sigma_1'(t_1)$, by Lemma 8.15 we obtain $v, w \in \Sigma$ with $s_1 \leq_1 \sigma_1(v)$, $\sigma_2(v) \leq_2 u_2 \geq_2 \sigma_2(w)$ and $\sigma_1(w) \geq_1 t_1$. By weak non-extension of $\sigma_2$, we obtain $\bar{u} \in \Sigma$ with $v \leq \bar{u} \geq w$ and $\sigma_2(\bar{u}) \leq_2 u_2$. Choose $\bar{u_1} = \sigma_1(\bar{u})$, then we have $\sigma_1(v) \leq_1 \bar{u_1} \geq_1 \sigma_1(w)$. By commutativity of the pushout and as $\sigma_2'(\sigma_2(\bar{u})) \leq' \sigma_2'(u_2) = u'$, we obtain $\sigma_1'(\bar{u_1}) \leq' u'$. □

The following lemma is required in order to show that CSP-CASL has quasi-amalgamation.

**Lemma 8.17** In CASL, pushouts of reflecting and weakly non-extensive signature morphisms exhibit the amalgamation property.

*Proof.* Using notation as above, we consider the pushout signature in enriched CASL (for details see [SMT+05]), which differs from CASL in that sub-sort pre-orders are generalised to sub-sort categories. The amalgamation exists if the sub-sort category in the pushout is thin, that is, a pre-order. In order to show this, we must show that any two paths of sub-sort injections with same start and end point in the pushout are equal, otherwise we would violate the 'thin' condition on the category, that is, we would not have a pre-order. Lemma 8.15 tells us the possible shapes of such paths. In cases where one of the endpoints is in the intersection of the images of $\sigma_1'$ and $\sigma_2'$, it must originate in $\Sigma$, and we are done by reflection. Otherwise, both paths live in the same summand w.r.t. the union in the statement of Lemma 8.15. For the first two summands, the result is clear. Now consider the third summand (the fourth one is treated similarly), that is, we have two paths

1. $s_1 \leq_1 \sigma_1(s), \sigma_2(s) \leq_2 s_2$
2. $s_1 \leq_1 \sigma_1(t), \sigma_2(t) \leq_2 s_2$

By weak non-extension of $\sigma_2$, there is some $u \in \Sigma$ with $s \leq u \geq t$ and $\sigma_2(u) \leq s_2$. But then the paths can be rewritten as

1. $s_1 \leq_1 \sigma_1(s) \leq_1 \sigma_1(u), \sigma_2(u) \leq_2 s_2$

2. $s_1 \leq_1 \sigma_1(t) \leq_1 \sigma_1(u), \sigma_2(u) \leq_2 s_2$

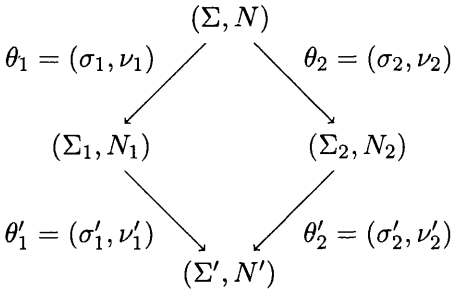
and since $\Sigma_1$ is thin (i.e., has a sub-sort pre-order), both paths are equal. □

With the super-category CSPCASLSIG$^{plain}$ and the previous lemmas, we can now establish that CSP-CASL has quasi-amalgamation.

**Theorem 8.18** CSP-CASL has the quasi-amalgamation property for the Traces, Failures/Divergences and Stable-Failures semantics.

*Proof.* CSPCASLSIG$^{plain}$ is a wide sub-category of CSPCASLSIG, as CSPCASLSIG$^{plain}$ is the same as CSPCASLSIG, but with relaxed conditions on signature morphisms. Lemma 8.16 establishes that CSPCASLSIG$^{plain}$ has pushouts and any such pushout of a span in CSPCASLSIG leads to a commuting square in CSPCASLSIG.

We now show that models and model morphisms can be amalgamated. Let




be the CSPCASLSIG$^{plain}$-pushout of the span $((\Sigma, N), \theta_1, \theta_2)$ in CSPCASLSIG. Furthermore, let $(M_i, I_i)$ be a $(\Sigma_i, N_i)$-model $(i = 1, 2)$ w.r.t. the Traces, Failures/Divergences or Stable-Failures semantics such that $(M_1, I_1)|_{\theta_1} = (M_2, I_2)|_{\theta_2}$. We construct a $(\Sigma', N')$-model $(M', I')$ as follows. Let $M'$ be the amalgamation of $M_1$ and $M_2$ in CASL. It exists due to Lemma 8.17.

It remains to amalgamate the process parts. Let

$$I'(n'(a'_1, \ldots, a'_k)) := \begin{cases} \alpha^{\mathcal{D}}_{\sigma'_1, M'}(I_1(n_1(\alpha_{\sigma'_1, M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma'_1, M'}{}^{-1}(a'_k)))), \text{if } n_1 \in N_1 \text{ with } \nu'_1(n_1) = n' \\ \alpha^{\mathcal{D}}_{\sigma'_2, M'}(I_2(n_2(\alpha_{\sigma'_2, M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma'_2, M'}{}^{-1}(a'_k)))), \text{if } n_2 \in N_2 \text{ with } \nu'_2(n_2) = n' \end{cases}$$

for all process names $n'_{w', comms'} \in N'$ and alphabet elements $a'_1, \ldots, a'_k \in \overline{s'_1}_{M'} \times \ldots \times \overline{s'_k}_{M'}$ where $w' = \langle s'_1, \ldots, s'_k \rangle$.

This construction is well-defined because $(M_1, I_1)|_{\theta_1} = (M_2, I_2)|_{\theta_2}$, as commutativity of signature morphisms in the pushout lifts to the alphabet and domain levels and the fact that the covariant and contravariant domain translations are inverse (Lemmas 7.16 and 7.18). The partial inverses of parameters are defined thanks to the strict translation of parameter sorts of process names. As $(M_i, I_i)$ is a CSP-CASL model $(i = 1, 2)$ and thus satisfies the controlled traces condition (a requirement of CSP-CASL models), it follows that $I'$ also satisfies the controlled traces condition, and therefore $(M', I')$ is a CSP-CASL model. It follows from the

construction of $I'$ that $(M', I')|_{\theta'_i} = (M_i, I_i)$ $(i = 1, 2)$. Due to the typing of process names in signature morphisms and the facts that $I'|_{\theta'_1} = I_1$ and $I'|_{\theta'_2} = I_2$, $I'$ is unique (and $M'$ is unique since amalgamation in CASL is unique).

Left to show is that model morphisms can be amalgamated in a similar way. We use the notation $(M_1, I_1) \oplus (M_2, I_2)$ for the amalgamation of CSP-CASL models $(M_1, I_1)$ and $(M_2, I_2)$, yielding the amalgamated CSP-CASL model $(M_1 \oplus M_2, I_1 \oplus I_2) = (M', I')$ where $M'$ is the amalgamation of CASL models $M_1$ and $M_2$, and $I'$ is the amalgamation of the process interpretation map $I_1$ and $I_2$ as constructed above.

Let $h_i : (M_i^A, I_i^A) \to (M_i^B, I_i^B)$ be $(\Sigma_i, N_i)$-model morphisms $(i = 1, 2)$ such that $h_1|_{\theta_1} = h_2|_{\theta_2}$. By the definition of CSP-CASL model morphisms (see Section 8.1.2), we know $h_1 : M_1^A \to M_1^B$ and $h_2 : M_2^A \to M_2^B$ are CASL model morphisms. As the CASL institution has the amalgamation property, we can form the unique amalgamated CASL model morphism $h' : M_1^A \oplus M_2^A \to M_1^B \oplus M_2^B$. Furthermore, we know $h'|_{\sigma'_1} = h_1$ and $h'|_{\sigma'_2} = h_2$.

By the definition of CSP-CASL model morphisms, we know $h' : (M_1^A, I_1^A) \oplus (M_2^A, I_2^A) \to (M_1^B, I_1^B) \oplus (M_2^B, I_2^B)$ is a CSP-CASL model morphism if

$$\alpha^{\mathcal{D}}_{h'}(I_1^A \oplus I_2^A(n'(\vec{a'}))) \bowtie I_1^B \oplus I_2^B(n'(\alpha_{h'}(\vec{a'})))$$

for all process names $n' \in N'$ and all suitable parameters $\vec{a'}$ where $\bowtie$ is $\sqsupseteq_{\mathcal{T}}$ for $\mathcal{D} = \mathcal{T}$, $\sqsubseteq_{\mathcal{N}}$ for $\mathcal{D} = \mathcal{N}$, and $\sqsupseteq_{\mathcal{F}}$ for $\mathcal{D} = \mathcal{F}$. We now show this property always holds for $h'$.

Let $n'$ be a process name in $N'$ and $\vec{a'}$ be suitable parameters. Thanks to the pushout we know names in $N'$ either originate from $N_1$ or $N_2$. Assume without loss of generality that $n'$ originates from $N_1$, that is, there exists $n_1 \in N_1$ such that $\nu'_1(n_1) = n'$. Thanks to our above definition of CSP-CASL model amalgamation we must show

$$\alpha^{\mathcal{D}}_{h'}(\alpha^{\mathcal{D}}_{\sigma'_1, M_1^A \oplus M_2^A}(I_1^A(n_1(\alpha_{\sigma'_1, M_1^A \oplus M_2^A}{}^{-1}(\vec{a'}))))) \bowtie \alpha^{\mathcal{D}}_{\sigma'_1, M_1^B \oplus M_2^B}(I_1^B(n_1(\alpha_{\sigma'_1, M_1^B \oplus M_2^B}{}^{-1}(\alpha_{h'}(\vec{a'})))))\ .$$

As $h_1$ is a CSP-CASL model morphism, we know that for this particular $n_1$ and parameters $\alpha_{\sigma'_1, M_1^A \oplus M_2^A}{}^{-1}(\vec{a'})$ that

$$\alpha^{\mathcal{D}}_{h_1}(I_1^A(n_1(\alpha_{\sigma'_1, M_1^A \oplus M_2^A}{}^{-1}(\vec{a'})))) \bowtie I_1^B(n_1(\alpha_{h_1}(\alpha_{\sigma'_1, M_1^A \oplus M_2^A}{}^{-1}(\vec{a'}))))$$

holds. By applying $\alpha^{\mathcal{D}}_{\sigma'_1, M_1^B \oplus M_2^B}$ to both sides, as it preserves refinement (Corollary 7.15), we obtain

$$\alpha^{\mathcal{D}}_{\sigma'_1, M_1^B \oplus M_2^B}(\alpha^{\mathcal{D}}_{h_1}(I_1^A(n_1(\alpha_{\sigma'_1, M_1^A \oplus M_2^A}{}^{-1}(\vec{a'}))))) \bowtie \alpha^{\mathcal{D}}_{\sigma'_1, M_1^B \oplus M_2^B}(I_1^B(n_1(\alpha_{h_1}(\alpha_{\sigma'_1, M_1^A \oplus M_2^A}{}^{-1}(\vec{a'})))))\ .$$

As $\alpha^{\mathcal{D}}_{\sigma'_1,M_1^B\oplus M_2^B} \circ \alpha^{\mathcal{D}}_{h_1} = \alpha^{\mathcal{D}}_{h'} \circ \alpha^{\mathcal{D}}_{\sigma'_1,M_1^A\oplus M_2^A}$ (this follows from Lemma 7.29 and the fact $\mathcal{D}$ is a functor, Lemma 7.6), we obtain

$$(\alpha^{\mathcal{D}}_{h'}(\alpha^{\mathcal{D}}_{\sigma'_1,M_1^A\oplus M_2^A} I_1^A(n_1(\alpha_{\sigma'_1,M_1^A\oplus M_2^A}{}^{-1}(\vec{a'}))))) \bowtie \alpha^{\mathcal{D}}_{\sigma'_1,M_1^B\oplus M_2^B}(I_1^B(n_1(\alpha_{h_1}(\alpha_{\sigma'_1,M_1^A\oplus M_2^A}{}^{-1}(\vec{a'})))))\ .$$

By Lemma 7.29, we know $\alpha_{h_1} \circ \alpha_{\sigma'_1,M_1^A\oplus M_2^A}{}^{-1} = \alpha_{\sigma'_1,M_1^B\oplus M_2^B}{}^{-1} \circ \alpha_{h'}$, thus we obtain

$$(\alpha^{\mathcal{D}}_{h'}(\alpha^{\mathcal{D}}_{\sigma'_1,M_1^A\oplus M_2^A} I_1^A(n_1(\alpha_{\sigma'_1,M_1^A\oplus M_2^A}{}^{-1}(\vec{a'}))))) \bowtie \alpha^{\mathcal{D}}_{\sigma'_1,M_1^B\oplus M_2^B}(I_1^B(n_1(\alpha_{\sigma'_1,M_1^B\oplus M_2^B}{}^{-1}(\alpha_{h'}(\vec{a'})))))$$

and conclude that $h'$ is a CSP-CASL model morphism in $\mathbf{mod}((\Sigma', N'))$.

As $h'|_{\sigma'_1} = h_1$ and $h'|_{\sigma'_2} = h_2$, we know that $h'|_{\theta'_1} = h_1$ and $h'|_{\theta'_2} = h_2$. That is, the amalgamated CSP-CASL model morphism reduces to its component parts. Uniqueness is guaranteed as $h'$ is the unique amalgamation of the CASL model morphisms $h_1$ and $h_2$. This completes the proof of amalgamation. □

As CSP-CASL now forms institutions we can use the institution independent structuring mechanisms presented in Section 4.7 to create structured CSP-CASL. This allows specifications to be constructed using the specification building operators **and**, **then**, **hide**, and **rename**. Generic CSP-CASL specifications and instantiations can now be formed as we have established quasi-amalgamation and met the required pre-requisites of the institution independent structuring mechanisms, namely, we have given a unique construction of signatures and a suitable amalgamation of models and model morphisms. Formalising CSP-CASL as institutions and establishing quasi-amalgamation has established CSP-CASL as a proper structured specification language.

## 8.3 CSP-CASL with Channels

In order for CSP-CASL to be of practical use we need to extend it with the notion of channels (see Section 2.2.3). This extension leads to further CSP-CASL institutions built on top of the institutions presented in Section 8.1, with extended notions of signatures and sentences. Most prominently, the notion of a signature is extended by a third component $C$. Thus, a signature becomes a triple:

$$(\Sigma_{Data}, C, \Sigma_{Proc})$$

where $C$ is a finite set of names typed by non-empty lists over $S$. We require $C$ to be closed under the sub-sort relation[3] $\leq^*$, that is, if $c_{\langle s_1,\ldots,s_k\rangle} \in C$ and $\langle u_1,\ldots,u_k\rangle \leq^* \langle s_1,\ldots,s_k\rangle$, then $c_{\langle u_1,\ldots,u_k\rangle} \in C$.

[3] $\leq^*$ stands for the pointwise extension of the sub-sort relation $\leq$ to strings of sorts.

CSP-CASL with channels can be reduced to CSP-CASL (without channels) as follows: each CSP-CASL signature with a channel component is translated to a CSP-CASL theory $\Phi(\Sigma)$, where each channel is coded as a new sort (with new inverse functions relating the new sort with the original one) and each CSP-CASL $\Sigma$-sentence $\varphi$ is translated to a CSP-CASL $\Phi(\Sigma)$-sentence $\alpha(\varphi)$ by reducing channel communication to ordinary communication using the new channel sorts. Models and satisfaction can then be easily borrowed from CSP-CASL by letting $\mathbf{mod}^{CCWC}(\Sigma_{CC}) := \mathbf{mod}^{CC}(\Phi(\Sigma_{CC}))$ and $(M, I) \models^{CCWC}_{\Sigma} \varphi$ iff $(M, I) \models^{CC}_{\Phi(\Sigma)} \alpha(\varphi)$ where the superscripts $CCWC$ indicate CSP-CASL with channels and $CC$ indicates CSP-CASL without channels. This is an instance of borrowing logical structure in the sense of [CM97].

We now present an example which illustrates this construction. Consider the following CSP-CASL specification of a simple vending machine.

**spec** VM_WITH_CHANNELS =
  **data sorts** *Coin*, *Item*
    **ops** *tea*, *coffee* : *Item*
  **channels** *Payment* : *Coin*; *Dispense* : *Item*
  **process** *VM* : *Payment*, *Dispense*;
    $VM = Payment\,?\,coin :: Coin \rightarrow$
      $(Dispense\,!\,tea \rightarrow VM \,\Box\, Dispense\,!\,coffee \rightarrow VM)$
**end**

Here, there are two sorts: *Coin* representing coins taken as payment and *Item* representing dispensable items. There are two constants *tea* and *coffee* of sort *Item*, representing the only two items that this vending machine offers. There are two channels: *Payment* which runs over sort *Coin* and *Dispense* which runs over sort *Item*. The *Payment* channel is to be used when a customer is inputting coins into the machine and the *Dispense* channel when the machine is providing items to the customer. The process part contains a single process name *VM* which can communicate over both channels. There is a single axiom constraining the behaviour of *VM* such that it takes a coin on the channel *Payment* and then delivers a choice of *tea* or *coffee* on the channel *Dispense*. After completion of the delivery, the machine starts over.

The specification VM_WITH_CHANNELS is encoded in CSP-CASL without channels as the following specification:

**spec** VM_WITHOUT_CHANNELS =
  **data sorts** *Coin*, *Item*, *Payment_Coin*, *Dispense_Item*
    **ops** *tea*, *coffee* : *Item*;
      *pack_Payment_Coin* : $Coin \rightarrow Payment\_Coin$;
      *unpack_Payment_Coin* : $Payment\_Coin \rightarrow Coin$;
      *pack_Dispense_Item* : $Item \rightarrow Dispense\_Item$;
      *unpack_Dispense_Item* : $Dispense\_Item \rightarrow Item$
    $\forall\, x : Coin \bullet unpack\_Payment\_Coin(pack\_Payment\_Coin(x)) = x$
    $\forall\, x : Payment\_Coin \bullet pack\_Payment\_Coin(unpack\_Payment\_Coin(x)) = x$
    $\forall\, x : Item \bullet unpack\_Dispense\_Item(pack\_Dispense\_Item(x)) = x$
    $\forall\, x : Dispense\_Item \bullet pack\_Dispense\_Item(unpack\_Dispense\_Item(x)) = x$

**process** *VM* : *Payment_Coin*, *Dispense_Item*;
$VM = \Box\ packedCoin :: Payment\_Coin \rightarrow$
$(pack\_Dispense\_Item(tea) \rightarrow VM \ \Box\ pack\_Dispense\_Item(coffee) \rightarrow VM)$
**end**

Here, we still have the sorts *Coin* and *Item*, but we also have the additional sorts *Payment_Coin* and *Dispense_Item*, which are used to model the two channels. In the process part a communication of sort *Payment_Coin* would represent a communication of *Coin* over the channel *Payment*. We also have four additional operation symbols, namely: *pack_Payment_Coin*, *unpack_Payment_Coin*, *pack_Dispense_Item*, and *unpack_Dispense_Item*. These are used to "pack" and "unpack" raw values into and out of the sorts representing the channels. The axioms state that the "pack" and corresponding "unpack" functions are inverse. With these symbols now available we can construct the process part. The *VM* process now communicates over the sort version of the channels. It now receives a value of sort *Payment_Coin* instead of receiving an element of sort *Coin* on the channel *Payment*. We simulate the sending of values down channels by "packing" them into the relevant sort and then communicating the result, for example, to send the value *tea* down the channel *Dispense* we communicate the value *pack_Dispense_Item(tea)*.

This construction allows us to translate any CSP-CASL specification into an equivalent specification which does not use channels.

In the rest of the thesis when we refer to CSP-CASL we are actually referring to Structured CSP-CASL with channels.

## 8.4 Possible Extensions

There are several possible extensions to CSP-CASL that can be studied. Here, we outline three of them, namely signature morphisms which allow for the shrinking of communication sets, overloading on process names, and signature morphisms with ground terms.

### 8.4.1 Signature morphisms which allow for shrinking of communication sets

Following the ideas from the CSP institutions (see Section 4.6) one could allow CSP-CASL signature morphisms to shrink the communication sets of process names. We demonstrate this idea here by temporally changing the definition of CSP-CASL signature morphisms and showing via an example how such a notion is useful. Unfortunately, this definition fails to give us good amalgamation properties. We discuss this via a counter example.

For the rest of this section, assume that the notion of CSP-CASL signature morphisms is defined as follows.

Given CSP-CASL signatures $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$ and $\Sigma'_{CC} = (\Sigma'_{Data}, \Sigma'_{Proc})$, with $S$ being the sort set of $\Sigma_{Data}$, a CSP-CASL signature morphism is a pair $\theta = (\sigma, \nu) : \Sigma_{CC} \rightarrow \Sigma'_{CC}$ where:

- $\sigma : \Sigma_{Data} \rightarrow \Sigma'_{Data}$ is a restricted sub-sorted (i.e., $ResSubPCFOL^{=}$) signature morphism (see Section 4.4).

- $\nu = (\nu_{w,comms})_{w \in S^*, comms \in S_\downarrow}$ is a family of functions such that

$$\nu_{w,comms} : N_{w,comms} \to \cup_{comms' \in (\downarrow(\sigma(comms)))_\downarrow} N'_{\sigma(w),comms'}$$

  is a mapping of process names. Another way to express this is that a process name $n \in N_{w,comms}$ is mapped to $\nu_{w,comms}(n) = n'$, where $n' \in N'_{\sigma(w),comms'}$ and $\forall y \in comms' \bullet \exists x \in comms \bullet y \leq \sigma(x)$ ("the target is dominated by the source"). We also write $\nu(n_{w,comms}) = n'_{\sigma(w),comms'}$.

This definition allows, for example, a process name $n$ with communication set $\{s, t, u\}$ to be mapped to a process name $n'$ with communication set $\{s\}$, where the data part of the signature morphism is the identity map. We now illustrate why such a feature would be desirable from the modelling perspective.

We develop a single user ATM (cash dispensing machine) in CSP-CASL. We develop this system via three levels of abstraction: starting from the most abstract architectural level (ARCH), via the abstract component level (ACL), to the concrete component level (CCL). We first develop the system utilising shrinking communication sets. Following this, we show an alternative development which does not utilise shrinking communication sets. This alternative development can be adopted instead of the original design without affecting the methodology or modelling too much, but at the cost of taking design decisions earlier.

#### 8.4.1.1 Using Shrinking Communication Sets

The system consists of two components: A user and an ATM. The user can request an amount of money to withdraw from their account via the ATM, after which the ATM will either decide to allow the withdrawal or refuse it. Full specifications are in Appendix C.1.

We first model the system, on the architectural level, by stating there are two sorts *Number* and *Decision*. Each component is specified in a separate specification where the component is modelled as a loose process which can communicate both sorts *Number* and *Decision*. We then combine these components via the CSP synchronous parallel operator in a system specification.

At the abstract component level, we model numbers using integers via the CASL data type *Int* provided by the standard CASL libraries. CASL also provides sorts *Pos* (of positive natural numbers) which are a sub-sort of *Nat* (natural numbers), which in turn are a sub-sort of *Int*. As our communication sets must be sub-sort closed, both the ATM and user processes now communicate over the set $\{Int, Nat, Pos, Decision\}$. We model the *User* process as:

*User* : *Int*, *Nat*, *Pos*, *Decision*;
$User = (\sqcap\, x :: Int \to User) \sqcap (\Box\, y :: Decision \to User)$

Here, the user internally decides to either send an integer (i.e., a withdrawal request) or receive a decision, and then start over. We do not model the protocol at this level, only the fact that these are the only two possible actions which can happen repeatedly. The ATM is modelled in a similar fashion.

Finally, we model the system at the concrete component level. Here, we decide that the user should only be able to request a positive amount to withdraw. We decide that the ATM shall

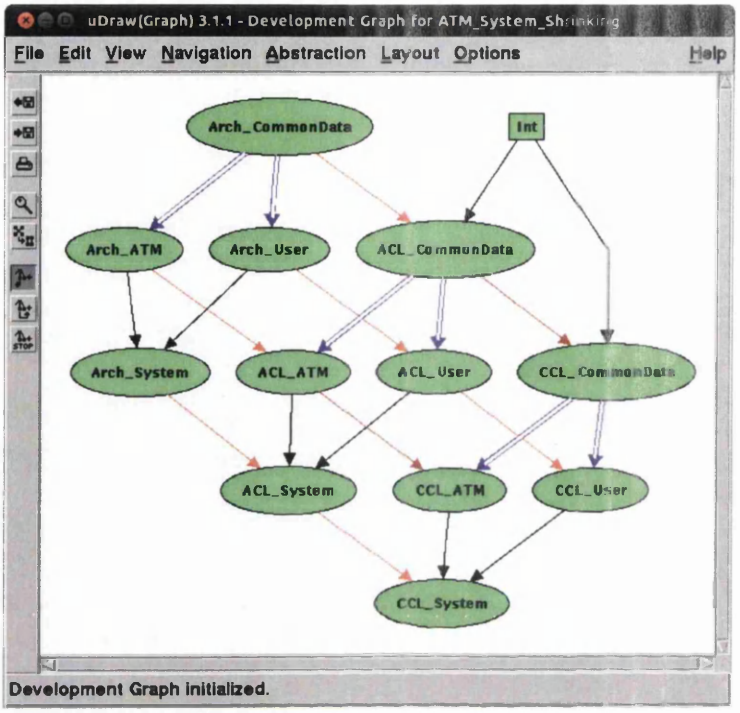



Figure 8.3: Possible development graph of the ATM system if shrinking was implemented in HETS.

still be open to the possibility of receiving non-positive amounts (possibly for the purposes of interacting with other administrative systems). We also model the protocol at this level: the user requests a withdrawal and then the ATM sends a decision message. Following this the user is free to request further withdrawals. To this end, we model the user as

$User : Pos, Decision;$
$User = \sqcap\ amount :: Pos \rightarrow \Box\ decision :: Decision \rightarrow User$

Here, we require the user to only communicate using the sorts *Pos* and *Decision*, thus disallowing them to send non-positive withdrawal requests.

Furthermore, at this level we add state to the ATM for recording the users balance. The ATM has a starting balance for the user modelled by the constant *startingBalance*. We provide an auxiliary process *ATMAux* which has as a parameter an integer recording the user balance. Negative values of this parameter indicate that the user is debit (possibly utilising overdraft facilities).

Finally, we specify several views which establish the formal refinements between the specifications at the three abstraction levels. The so called development graph of these specifications can be seen in Figure 8.3, under the assumption that such shrinking is implemented within HETS. The purple double lined arrows represent CASL imports into CSP-CASL, the

black arrows represent imports, and the red arrows represent views (or refinements) between specifications.

The view

**view** ACL2CCL_USER : ACL_USER **to** CCL_USER =
*User* : {*Int*, *Nat*, *Pos*, *Decision*} $\mapsto$ *User* : {*Pos*, *Decision*}

illustrates a signature morphism with a shrinking communication set. We must specify the process name mapping in the view as we change (shrink) the communication set of the *User* process.

In summary we have modelled a simple ATM system at three levels of abstraction while utilising shrinking communication sets.

#### 8.4.1.2 Without using Shrinking Communication Sets

We now discuss how it is possible to have a similar but alternative development, whilst avoiding the use of signature morphisms which shrink communication sets. Thus, this example fits into the notion of CSP-CASL defined in Section 8.1, which does not allow for shrinking communication sets.

In an alternative development, see Appendix C.2, we make design decisions at higher levels of abstraction in order to avoid shrinking the communication sets of processes. Already at the architectural levels we decide to use two sorts to represent numbers, one for the user and one for the ATM, where the ATM's number include the user's numbers. This is done by the CASL specification:

**sort** *Decision*
**sort** *UserNumber* < *ATMNumber*

We are then able to develop these two sorts independently for the ATM and user. To this end, we develop *ATMNumber* to *Int*, while we develop *UserNumber* to *Pos* at the ACL level. This allows us to already have appropriate communications sets in place at this level and avoid shrinking communication alphabets between the ACL to the CCL levels. The downside of this is that already at the ACL level we have had to make the design decision that the user will communicate over positive natural numbers:

*User* : *Pos*, *Decision*;
*User* = ($\sqcap$ *x* :: *Pos* $\rightarrow$ *User*) $\sqcap$ ($\Box$ *y* :: *Decision* $\rightarrow$ *User*)

This example illustrates that while signature morphisms which allow communication sets to be shrunk can be somewhat useful in modelling, there are alternative developments, which can be adopted instead. These alternative developments are similar to the original development, but take design decisions at higher levels of abstraction. Hence, forbidding signature morphisms with shrinking communications sets does not severely restrict modelling. It comes, however, at the cost of less flexibility should a "wrong" design decision be discovered later in the design process.

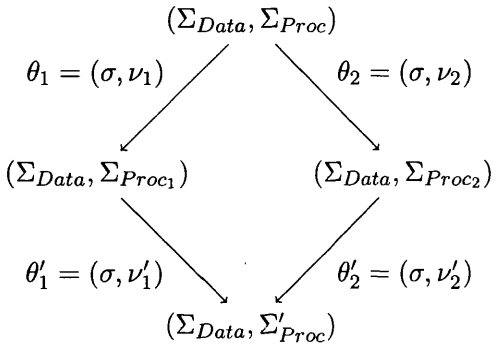
We conclude this extension with a counter example that shows our notion of CSP-CASL signature morphisms, which allow for shrinking of communication sets, fails to have good amalgamation properties.

#### 8.4.1.3 A Counter Example for Amalgamation

Here, we demonstrate the problems of creating an amalgamated model when using shrinking alphabets.

We keep the data part constant throughout this example and work in the CSP Traces semantics.

Consider the following pushout in CSPCASLSIG$^{plain}$




such that $((\Sigma_{Data}, \Sigma_{Proc}), \theta_1, \theta_2)$ is a span in CSPCASLSIG, where the signatures are as follows:

- $\Sigma_{Data}$ is given by the following CASL specification:

  **spec** D =
  **sorts** $t, u < s$
  **ops** $a : t$;
  $b : u$
  • $a = b$
  **end**

- $\Sigma_{Proc}$ contains a single process name $n_{\langle\rangle,\{s,t,u\}}$.
- $\Sigma_{Proc_1}$ contains a single process name $n_{\langle\rangle,\{t\}}$.
- $\Sigma_{Proc_2}$ contains a single process name $n_{\langle\rangle,\{u\}}$.
- $\Sigma_{Proc_2}$ contains a single process name $n_{\langle\rangle,\emptyset}$.

The signature morphisms are as follows:

- $\sigma = Id_{\Sigma_{Data}}$.

- $\nu_1(n_{\langle\rangle,\{s,t,u\}}) = n_{\langle\rangle,\{t\}}$.
- $\nu_2(n_{\langle\rangle,\{s,t,u\}}) = n_{\langle\rangle,\{u\}}$.
- $\nu'_1(n_{\langle\rangle,\{t\}}) = n_{\langle\rangle,\emptyset}$.
- $\nu'_2(n_{\langle\rangle,\{u\}}) = n_{\langle\rangle,\emptyset}$.

This choice of signatures and signature morphisms results in a pushout.

Now let $M$ be some data model satisfying the above axiom $a = b$. Let $(M, I_1)$ and $(M, I_2)$ be CSP-CASL models of $(\Sigma_{Data}, \Sigma_{Proc_1})$ and $(\Sigma_{Data}, \Sigma_{Proc_2})$ respectively, such that $I_1|_{\theta_1} = I_2|_{\theta_2}$, and $(M, I_1)$ satisfies the process sentence $n = a \rightarrow \text{STOP}$ and $(M, I_2)$ satisfies the process sentence $n = b \rightarrow \text{STOP}$, that is, $I_1(n) = \{\langle\rangle, \langle\overline{a}\rangle\}$ and $I_2(n) = \{\langle\rangle, \langle\overline{b}\rangle\}$. Thanks to the sub-sort relation, the axiom $a = b$ and CSP-CASL's definition of reduct, we have $I_1|_{\theta_1}(n) = I_2|_{\theta_2}(n) = \{\langle\rangle, \langle\overline{a}\rangle\}$. Thus, any amalgamated model $I'$ of $I_1$ and $I_2$ will need to contain this common trace. This would cause $cTr(I'(n_{\langle\rangle,\emptyset}))$ to contain at least the trace $\langle\overline{a}\rangle$ and thus, fall outside the declared communication set (which in this case is the empty set). Thus, it violates the CSP-CASL model definition.

This problem has occurred because we have shrunk the communication sets in two incompatible ways. Once to the sub-sort $t$, and once to the sub-sort $u$, where the axiom $a = b$ ensures that these sub-sorts share at least one element. There is no way for the pushout signature to capture just these shared elements via sort symbols, thus we must settle for the intersection $\{t\} \cap \{s\} = \emptyset$ of the sort sets. This in turn means the controlled traces of amalgamated models are outside the declared communication set.

This example shows that sort "$s \cap t$" would be needed in the pushout signature in order to make amalgamation work. This is problematic as:

- Adding it to CASL would increase the CASL type system. Baumeister [Bau99] has studied and disregarded disjunction of data types in algebraic specification.
- Allowing "$s \cap t$" only in the communication sets of the process part, would mean that we are unable to reason about it in the CASL part. This goes against the design of CSP-CASL where data in the process part shall be analysable in CASL.

Here, we have considered a possible alternative version of CSP-CASL which allows signature morphisms to shrink the communication sets of process names. We have discussed why this would be a desirable feature from the modelling perspective. This particular approach however, fails to provide good amalgamation properties, which we have discussed via a counter example.

The next extension might allow for the benefits of shrinking communication sets without changing the definition of CSP-CASL signature morphisms.

### 8.4.2 Overloaded Process Names

Further research could introduce a notion of overloading on process names which mimics overloading on CASL functions and predicate symbols. We surmise that a suitable overloading relation $\sim_N$ might be $P_{w_1,comms_1} \sim_N P_{w_2,comms_2}$ if and only if there exists $w \in S^*$,

$comms' \in S_{\downarrow}$ such that $w \leqslant w_1$, $w \leqslant w_2$, $comms_1 \leqslant comms'$, and $comms_2 \leqslant comms'$. Signature morphisms would then have to preserve overloaded process names. It is not obvious that this construction will work out, as the signature morphism restrictions of CSP-CASL might interfere with the preservation of the overloading relation.

Assuming this did work out, the models would need to then agree when they are "cast" into the common larger communication sets. This would then force process names with the same underlying name to agree on behaviour that falls in the common parts of their communication sets.

A suitable notion of overloading might be able to simulate the desired behaviour of shrinking communication sets outlined in Section 8.4.1, whilst preserving quasi-amalgamation.

### 8.4.3 Signature Morphisms With Ground Terms

Another extension that could be studied would be to allow process maps such as $P \mapsto Q(0)$ where we fill in the parameters with ground terms. We can currently simulate this by introducing new process names that keep the existing parameter types and introduce additional axioms to specify the ground terms, for example, $P \mapsto Q$, where we have the axiom $Q = Q'(0)$. This workaround would get tedious as specifications get larger. This could also be formalised as syntactic sugar which would not require any changes to the CSP-CASL construction described within this thesis.

Within this chapter we have presented the construction of CSP-CASL as three institutions, one for each of the main CSP semantics. As CSP-CASL now forms various institutions, we are able to use all the structuring mechanisms of CASL presented in Section 4.7. We have presented results on pushouts and amalgamation properties which are essential for using generic and instantiated CSP-CASL specifications. We have also extended CSP-CASL with channels, thus forming three additional institutions: CSP-CASL with channels for each of the CSP semantics. We concluded this chapter with a discussion about various extensions of CSP-CASL which could be studied.

*Chapter 9*

# Refinement and Compositional Proof Calculi Over Structured CSP-CASL

**Contents**



Our overall aim in formalising CSP-CASL as institutions (in Chapter 8) was to allow system development using CSP-CASL. As a result of this formalisation, we can use the structuring mechanisms presented in Section 4.7 to obtain Structured CSP-CASL. This allows us to write Structured CSP-CASL specifications that can capture systems in a compositional way. Our next task is to understand how we can develop such specifications and reason about them. The first step in enabling the development of specifications is to develop a suitable refinement notion for CSP-CASL. Once a suitable refinement notion has been developed, we may study if it is possible to develop CSP-CASL specifications and reason about them in a compositional manner.

In this chapter, we first discuss the new features of CSP-CASL. We then develop a new refinement notion that supports multiple process names and loose process semantics. Following this, we design a compositional calculus that allow us to establish refinements over structured CSP-CASL specifications. We then develop a calculus dedicated to deadlock analysis of structured specifications and a specialised rule for deadlock analysis of networks. Finally, we develop a refinement calculus for Structured CSP-CASL that we show is complete provided structured specifications are restricted to certain forms.

**ccspec** CSPCASLSPEC =
  **data sorts** $s$
    **ops** $a, b : s$
  **process let** $P\ (x{:}s) = x \rightarrow \text{STOP}$
    $Q = b \rightarrow \text{STOP}$
    **in** $P\ (a) \parallel Q$
**end**

**logic** CSPCASL
**spec** MULTICSPCASLSPEC =
  **data sorts** $s$
    **ops** $a, b : s$
  **process** $P\ (s) : s; Q : s; System : s;$
    $P\ (x) = x \rightarrow \text{STOP}$
    $Q = b \rightarrow \text{STOP}$
    $System = P\ (a) \parallel Q$
**end**

Figure 9.1: Introduction of process names in Structured CSP-CASL.

## 9.1 From CSP-CASL to Structured CSP-CASL

Before we start to study development in CSP-CASL, we first illustrate the new features of CSP-CASL, that is, process names and loose process semantics.

To a wide extent, specification in the large concerns the control over namespaces. CSP-CASL as designed in [Rog06] (see Chapter 5), however, neglected the aspect of process names: one specification, take for example the specification CSPCASLSPEC in Figure 9.1, which denotes one (unnamed) system. Consequently, the semantics of this specification is given as one family of process denotations. Over the CSP Traces semantics $\mathcal{T}$, this family has the following structure: in CASL models $M$ with $M \models a = b$, the terms $a$ and $b$ can synchronise, and we obtain the denotation $\{\langle\rangle, \langle\overline{a_M}\rangle\}$. Here, $\overline{a_M}$ is the communication corresponding to the interpretation of the constant $a$ in the model $M$ (see Section 7.2.1). In CASL models $N$ with $N \models \neg a = b$, the terms $a$ and $b$ do not synchronise, thus the process part is in deadlock and we obtain the denotation $\{\langle\rangle\}$. Overall, the semantics (i.e., the model class) of the CSPCASLSPEC is the family

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

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

In contrast to this, Structured CSP-CASL, as we develop it in this thesis, offers the possibility to bind denotations to process names. Rather than representing one system, a Structured CSP-CASL specification provides a collection of components. The specification MULTICSP-CASLSPEC in Figure 9.1, written in Structured CSP-CASL, is 'semantically equivalent' to the specification CSPCASLSPEC.

Over the Traces semantics $\mathcal{T}$, we obtain as semantics for MULTICSPCASLSPEC: for CASL models $M$ with $M \models a = b$, the process interpretation function $I$ where

$$\begin{array}{lcl} I(P(x)) & = & \{\langle\rangle, \langle x\rangle\} \text{ for all } x \in \overline{s}_M \\ I(Q) & = & \{\langle\rangle, \langle\overline{b_M}\rangle\} \\ I(System) & = & \{\langle\rangle, \langle\overline{a_M}\rangle\} \end{array}$$

Here, $\overline{s}_M$ is the set of communications that is obtained from the carrier set of $s$ in model $M_\perp$ (see Section 7.2.1). For CASL models $N$ with $N \models \neg a = b$, the process interpretation function $J$ where

$$\begin{array}{lcl} J(P(x)) & = & \{\langle\rangle, \langle x\rangle\} \text{ for all } x \in \overline{s}_N \\ J(Q) & = & \{\langle\rangle, \langle\overline{b_N}\rangle\} \\ J(System) & = & \{\langle\rangle\} \end{array}$$

The overall model class of MULTICSPCASLSPEC is finally

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

In this example, Structured CSP-CASL simply adds process name information: hiding the information concerning the system's components, in our example the process names $P$ and $Q$, leads back to the original semantics of CSP-CASL. Structured CSP-CASL however also extends the original CSP-CASL language design by introducing the possibility of loose processes:

**logic** CSPCASL
**spec** LOOSEPROCESSES =
  **data sorts** $s$
    **ops** $a, b : s$
  **process** $P : s; Q : s; R : s;$
    $Q = a \rightarrow P$
    $R = P \between b \rightarrow \text{SKIP}$
**end**

The above specification LOOSEPROCESSES leaves the behaviour or the process name $P$ unspecified. However, it does state that the interpretations of $Q$ and $R$ depend on $P$. $Q$ communicates $a$ and then behaves like $P$. $R$ behaves like $P$, and, should $P$ successfully terminate, communicates $b$ and terminates successfully. On the semantical level, looseness in the process part means that one CASL model $M$ is paired with various functions $I$ within the model class of the specification. Let, for instance, $O$ be a CASL model with $O(s) = \{*, +\}$, $O(a) = *$, and $O(b) = +$. Let $I_1$ and $I_2$ be process interpretation maps (relative to $O$) with

$$\begin{array}{lcllcl} I_1(P) & = & \{\langle\rangle\} & I_2(P) & = & \{\langle\rangle, \langle\checkmark\rangle\} \\ I_1(Q) & = & \{\langle\rangle, \langle\overline{*}\rangle\} & I_2(Q) & = & \{\langle\rangle, \langle\overline{*}\rangle, \langle\overline{*}, \checkmark\rangle\} \\ I_1(R) & = & \{\langle\rangle\} & I_2(R) & = & \{\langle\rangle, \langle\overline{+}\rangle\langle\overline{+}, \checkmark\rangle\} \end{array}$$

Then $(O, I_1)$ and $(O, I_2)$ are both models of the specification LOOSEPROCESSES. This example illustrates that the process part of CSP-CASL specifications can also have loose semantics.

## 9.2 CSP-CASL Refinement for Loose Process Semantics

We now turn our attention to defining a new notion of CSP-CASL refinement that supports process names and loose process semantics. Refinement is essential in the development of specifications from high levels of abstractions to lower more concrete levels of abstraction. By

establishing formal refinements one can be sure such development steps are reasonable (see Chapters 3 and 5 for examples). Refinement also allows fundamental properties like deadlock freedom to be established (see Chapter 5 for an example).

In the original CSP-CASL (see Chapter 5), the notion of refinement was straightforward: $(D,P) \rightsquigarrow^{\sigma}_{\mathcal{D}} (D',P')$ if

$$I'|_{\sigma} \subseteq I \wedge \forall M' \in I' \bullet d_{M'|_{\sigma}} \sqsubseteq_{\mathcal{D}} \hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(d'_{M'})$$

where $(d_M)_{M\in I}$ and $(d'_{M'})_{M'\in I'}$ are the families of denotations for the specifications $(D,P)$ and $(D',P')$, respectively. Essentially, two points are compared for each data model. This idea is not directly applicable to our new setting and needs to be extended to cover loose process semantics.

To this end, we define a new refinement relation for CSP-CASL as presented in [OMR12]. The main challenge with defining a refinement notion for CSP-CASL is that CSP-CASL combines two different worlds, namely, algebraic specification in the form of CASL and process algebra in the form of CSP. These two settings each have a notion of refinement, but the notions differ in their underlying ideas. CSP has a notion of refinement between individual processes, for example, in the Traces semantics, $pt \sqsubseteq_{\mathcal{T}} pt'$ means that $pt'$ has fewer traces than $pt$, that is, $\mathit{traces}(pt') \subseteq \mathit{traces}(pt)$. On the other hand, the CASL family of languages uses model class inclusion as the simplest notion of refinement [Mos04]: $SP_1 \rightsquigarrow SP_2$ if $SP_2$ has fewer models than $SP_1$, that is, $\mathbf{Mod}(SP_2) \subseteq \mathbf{Mod}(SP_1)$. To cater for renaming, this notion can be extended by a signature morphism $\sigma$. In this case one defines $SP_1 \rightsquigarrow^{\sigma} SP_2$ if the reduced model class of $SP_2$ is contained within the model class of $SP_1$, that is, $\mathbf{Mod}(SP_2)|_{\sigma} \subseteq \mathbf{Mod}(SP_1)$. When combining these worlds through institution theory, one has to recognise that these two refinement notions follow different ideas: While CSP refinement talks about refinement of individual models, CASL refinement talks about refinement of model classes.

This should become clear with the following notion: a Structured CSP-CASL specification $SP$ is *single-valued* (written as Single-valued($SP$)), if there is no looseness in the processes, that is, any two $SP$-models with the same data part coincide. Traditional CSP refinement is about refinement between different single-valued process specifications – reducing the amount of internal non-determinism – whereas algebraic specification uses model class inclusion which mainly captures different degrees of *looseness* of specifications.

How to reconcile these two worlds is not clear. We want refinement in CSP-CASL to capture different degrees of looseness not only for data, but also for processes. Hence, we adopt a model class inclusion notion for CSP-CASL refinement. However, model class inclusion is not enough as we also want to capture CSP refinement between different single-valued specifications. Model class inclusion alone would obviously never lead to such refinements between single-valued specifications. Thus, we introduce the notion of *refinement closure* (and here, "refinement" is meant in the CSP sense, not in the model class inclusion sense) which will form part of the definition of CSP-CASL refinement.

Given a CSP-CASL specification $SP$ with signature $(\Sigma_{Data}, \Sigma_{Proc})$, its refinement closure *RefCl*($SP$) is defined as follows:

- the signature of *RefCl*($SP$) is that of $SP$,

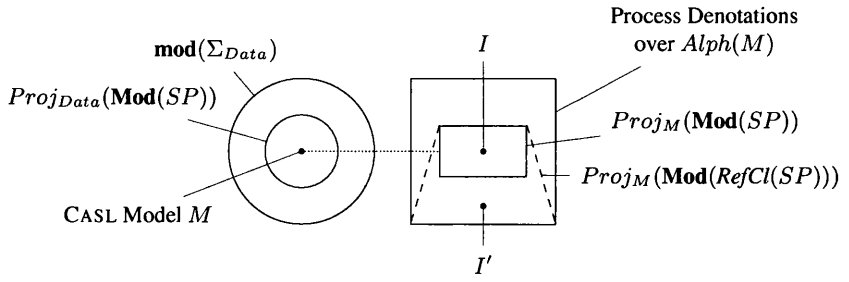



Figure 9.2: Diagram illustrating refinement closure of CSP-CASL specifications.

- the model class of $RefCl(SP)$ (i.e., $\mathbf{Mod}_{\mathcal{D}}(RefCl(SP))$) consists of those CSP-CASL models $(M', I')$ for which there exists a model $(M, I)$ of $SP$ such that
  - $M = M'$, that is, they have the same data part, and
  - $I \sqsubseteq_{\mathcal{D}} I'$, that is, for each $n \in \Sigma_{Proc}$ and all suitable alphabet elements $a_1, \ldots, a_k$,

$$I(n(a_1, \ldots, a_k)) \sqsubseteq_{\mathcal{D}} I'(n(a_1, \ldots, a_k))$$

  for each CSP semantics $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

The refinement closure of a specification adds all models which are CSP refinements of existing models to the model class.

Alternatively, the semantics of $RefCl(SP)$ can be expressed as a structured specification

$$SP \textbf{ then } p_1 \sqsubseteq_{\mathcal{D}} q_1, \ldots, p_n \sqsubseteq_{\mathcal{D}} q_n \textbf{ hide } p_1, \ldots, p_n \textbf{ with } q_1 \mapsto p_1, \ldots, q_n \mapsto p_n$$

where $p_1, \ldots, p_n$ are the process names of $SP$ (we assume here that all of them are unparametrised), $q_1, \ldots, q_n$ are new process names, and $p \sqsubseteq_{\mathcal{D}} q$ can be expressed as $p = p \sqcap q$ (see Section 2.4).

Figure 9.2 depicts the notion of refinement closure. Given a model $M$ of the data part of specification $SP$, we consider all of its possible process interpretation "partners" relative to $SP$: $Proj_M(\mathbf{Mod}(SP)) = \{I \mid (M, I) \in \mathbf{Mod}(SP)\}$ – this is represented by the rectangle. The refinement closure (represented by the half-dashed trapezium) includes all $I'$ such that there exists some $I \in Proj_M(\mathbf{Mod}(SP))$ that refines to $I'$.

With this notion, we are ready to define a notion of refinement that is suitable for CSP-CASL:

$$SP_1 \rightsquigarrow^{\theta}_{\mathcal{D}} SP_2 \text{ iff } \mathbf{Mod}_{\mathcal{D}}(SP_2)|_{\theta} \subseteq \mathbf{Mod}_{\mathcal{D}}(RefCl(SP_1))$$

for $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$. We omit $\mathcal{D}$ if it is clear from the context and we also omit $\theta$ if it is the identity signature morphism. This notion reconciles CASL refinement based on model class inclusion with CSP refinement based on inclusion of trace sets, failure sets, etc. Two specifications $SP_1$ and $SP_2$ are *equivalent* with respect to CSP semantics $\mathcal{D}$, written $SP_1 \equiv_{\mathcal{D}}$

$SP_2$, if their signatures and model classes with respect to $\mathcal{D}$ coincide. We drop the subscript $\mathcal{D}$ when the particular CSP semantics is clear from the context.

The following properties show that our refinement notion is well behaved and has the properties one would expect from any refinement notion.

**Lemma 9.1** (Basic Refinement Properties) The following basic properties are sound.

1. *RefCl* is monotonic, that is: if $\mathbf{Mod}(SP_1) \subseteq \mathbf{Mod}(SP_2)$, then $\mathbf{Mod}(\mathit{RefCl}(SP_1)) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_2))$.
2. *RefCl* is idempotent, that is $\mathit{RefCl}(SP) \equiv \mathit{RefCl}(\mathit{RefCl}(SP))$.
3. $\rightsquigarrow$ is reflexive and transitive.
4. If $SP_1 \rightsquigarrow SP_2$ and $SP_2 \rightsquigarrow SP_1$, then $\mathit{RefCl}(SP_1) \equiv \mathit{RefCl}(SP_2)$.
5. If $SP_1 \rightsquigarrow SP_2$, $SP_2 \rightsquigarrow SP_1$, and both are single-valued, then $SP_1 \equiv SP_2$.

*Proof.* We prove each property individually below.

1. follows easily from the definition of *RefCl*.
2. this follows from reflexivity and transitivity of CSP refinement (i.e., $\sqsubseteq$).
3. follows from 1 and 2.
4. follows from 1 and 2.
5. By 4, $\mathit{RefCl}(SP1) \equiv \mathit{RefCl}(SP2)$. Since different single-valued processes have different refinement closures, $SP1 \equiv SP2$. □

Following ideas given in [KR09] we obtain a decomposition theorem for refinements of basic CSP-CASL specifications (i.e., CSP-CASL presentations). A basic CSP-CASL specification $SP$ can be expressed as a pair $(D, P)$ where $D$ is the data part and a $P$ is the process part. Each part contains the respective signature and axioms (where $P$ is dependent on $D$). This allows us to (syntactically) decompose a refinement between basic CSP-CASL specifications into a data refinement and a process refinement.

**Lemma 9.2** (Decomposition Rule) The following rule is sound.

$$\frac{\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D) \qquad (D', \theta(P)) \rightsquigarrow_{\mathcal{D}} (D', P')}{(D, P) \rightsquigarrow^{\theta}_{\mathcal{D}} (D', P')}$$

where $\theta = (\sigma, \nu)$ is a CSP-CASL signature morphism, and $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$. Here, $\theta(P)$ is the translation of the process part $P$ along $\theta$ and has the same signature as $P'$.

*Proof.* We show $\mathbf{Mod}(D', P')|_\theta \subseteq \mathbf{Mod}(\mathit{RefCl}(D, P))$. Let $(M, I) \in \mathbf{Mod}(D', P')|_\theta$. We show $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(D, P))$. We know there exists $(M', I') \in \mathbf{Mod}(D', P')$ such that $(M', I')|_\theta = (M, I)$. From the first assumption we know $M \in \mathbf{Mod}(D)$. By the second assumption we know there exists $J'$ such that $(M', J') \in \mathbf{Mod}(D', \theta(P))$ with $J' \sqsubseteq I'$. Thanks to $\hat{\alpha}^{\mathcal{D}}_{\theta,M'}$ preserving CSP refinement (Corollary 7.15), we know $J'|_\theta \sqsubseteq I'|_\theta = I$. If we can show $(M, J'|_\theta) \in \mathbf{Mod}(D, P)$, then we will have found a witness for $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(D, P))$. As we already know $M \in \mathbf{Mod}(D)$, it is sufficient to show

$$(M, J'|_\theta) \models \varphi_p \text{ for all } \varphi_p \in P \ .$$

By the satisfaction condition (Lemma 8.12), this is equivalent to showing

$$(M', J') \models \theta(\varphi_p) \text{ for all } \varphi_p \in P$$

which holds as $(M', J') \in \mathbf{Mod}(D', \theta(P))$. Thus, $(M, J'|_\theta) \in \mathbf{Mod}(D, P)$ and we conclude $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(D, P))$. □

The above lemma allows us to decompose a CSP-CASL refinement (between basic specifications) into a CASL refinement (i.e., $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$) and a process refinement $(D', \theta(P)) \rightsquigarrow_{\mathcal{D}} (D', P')$. The former proof obligation can be discharged using CASL's proof tool, HETS [MML07]. The latter can be proven using the tool CSP-CASL-Prover [ORI09].

## 9.3 Compositional Refinement Analysis

As CSP-CASL now forms institutions we can use the structuring mechanisms presented in Section 4.7 to create Structured CSP-CASL. This allows us to use the operations **and**, **rename** and **hide** on CSP-CASL specifications. With such structured specifications, the question arises if and how we can reason over such specifications in a compositional way.

In order to establish refinements between structured CSP-CASL specifications using CSP-CASL-Prover it is first necessary to reduce the refinement goal to a refinement between basic CSP-CASL specifications. The decomposition rule (see Section 9.2) can then applied so that tools such as CSP-CASL-Prover can be utilised. We show that it is possible to reduce specifications over structured CSP-CASL specifications to refinement over basic CSP-CASL specifications in many useful cases.

As a first proof of concept, we show that the specification building operators are monotonic w.r.t. the structuring operations, cf. [BCH99]. This requires, in our case, certain side conditions, most prominently for the structured union operation on specifications. Here, the conditions deal with the following non-monotonic situation of CSP-CASL refinement: there exist CSP-CASL specifications $SP_1, SP_1'$ and $SP_2$ with[1]

$$\mathbf{Mod}(SP_1') \subseteq \mathbf{Mod}(SP_1),\ \mathbf{Mod}(SP_1 \textbf{ and } SP_2) = \emptyset,\ \mathbf{Mod}(SP_1' \textbf{ and } SP_2) \neq \emptyset \ .$$

[1]Consider over the Traces semantics $SP_1 = (D, P = a \rightarrow \mathit{Stop})$, $SP_2 = (D, P = \mathit{Stop})$, and $SP_1' = (D, P = \mathit{Stop})$ where $D$ is a consistent CASL specification that declares a constant $a$. Then $SP_1$ **and** $SP_2$ is inconsistent, $SP_1 \rightsquigarrow_{\mathcal{T}} SP_1'$, and $SP_1'$ **and** $SP_2$ has model $(M, I)$ with $I(P) = \{\langle\rangle\}$ and $M \in \mathbf{Mod}(D)$.

To avoid such situations we introduce a notion of *process consistency* on CSP-CASL specifications. Two CSP-CASL specifications $SP_1$ and $SP_2$ are *process consistent*, written as ProcConst$(SP_1, SP_2)$, if for all $M \in (Proj_{Data}(\mathbf{Mod}(SP_1)) \cap Proj_{Data}(\mathbf{Mod}(SP_2)))$ such that there exists $(M, I_1) \in \mathbf{Mod}(SP_1)$ and $(M, I_2) \in \mathbf{Mod}(SP_2)$, it is the case that there exists $(M, J) \in \mathbf{Mod}(SP_1) \cap \mathbf{Mod}(SP_2)$.

Furthermore, we occasionally require that CSP-CASL signature morphisms are injective on process names, that is, process names are not collapsed. A CSP-CASL signature morphism $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ is injective on process names if for all process names $n_{w_n, comms_n}$ and $m_{w_m, comms_m}$ in $\Sigma_{CC}$ it is the case that

$$\nu(n_{w_n, comms_n}) = \nu(m_{w_m, comms_m}) \text{ implies } n_{w_n, comms_n} = m_{w_m, comms_m} \ .$$

Note that $\theta$ being injective on process names can have restrictions on the data part $\sigma$ of the signature morphism as data forms part of the identity of process names.

**Lemma 9.3** The following proof rules are sound over $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$:

$$\frac{SP_1 \rightsquigarrow SP_1' \quad \text{ProcConst}(SP_1, SP_2) \quad \text{Single-valued}(SP_i) \text{ for } i = 1 \vee i = 2}{(SP_1 \textbf{ and } SP_2) \rightsquigarrow (SP_1' \textbf{ and } SP_2)}$$

$$\frac{SP_1 \rightsquigarrow SP_1' \qquad SP_1 \equiv \mathit{RefCl}(SP_1)}{(SP_1 \textbf{ and } SP_2) \rightsquigarrow (SP_1' \textbf{ and } SP_2)}$$

$$\frac{SP \rightsquigarrow SP' \qquad \theta \text{ is injective on process names}}{(SP \textbf{ rename } \theta) \rightsquigarrow (SP' \textbf{ rename } \theta)}$$

$$\frac{SP \rightsquigarrow SP'}{(SP \textbf{ hide } \theta) \rightsquigarrow (SP' \textbf{ hide } \theta)}$$

where $\theta : \Sigma_{CC} \to \Sigma'_{CC}$ is a CSP-CASL signature morphism.

*Proof.* Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ be a CSP-CASL signature morphism. We prove each proof rule individually.

**Structured and rule 1** We show $\mathbf{Mod}(SP_1' \textbf{ and } SP_2) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$. Let $(M', I') \in \mathbf{Mod}(SP_1' \textbf{ and } SP_2)$, then we know $(M', I') \in \mathbf{Mod}(SP_1')$ and $(M', I') \in \mathbf{Mod}(SP_2)$. By the first assumption we know $(M', I') \in \mathbf{Mod}(\mathit{RefCl}(SP_1))$, that is, there exists $I$ such that $(M', I) \in \mathbf{Mod}(SP_1)$ with $I \sqsubseteq I'$. We also know $M' \in Proj_{Data}(\mathbf{Mod}(SP_1)) \cap Proj_{Data}(\mathbf{Mod}(SP_2))$, thus by assumption we know there exists $(M', J') \in \mathbf{Mod}(SP_1 \textbf{ and } SP_2)$. It is the case that either $SP_1$ or $SP_2$ is single valued.

**Case 1:** $SP_1$ **is single valued** As $(M', I) \in \mathbf{Mod}(SP_1)$ and $(M', J') \in \mathbf{Mod}(SP_1)$ we know $I = J'$. Finally, we can conclude $(M', I') \in \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$ as $I \sqsubseteq I'$.

**Case 2: $SP_2$ is single valued** As $(M', I') \in \mathbf{Mod}(SP_2)$ and $(M', J') \in \mathbf{Mod}(SP_2)$ we know $I' = J'$. Finally, as $I' \sqsubseteq I'$ we can conclude $(M', I') \in \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$.

**Structured and rule 2** We show $\mathbf{Mod}(SP_1' \textbf{ and } SP_2) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$. Let $(M', I') \in \mathbf{Mod}(SP_1' \textbf{ and } SP_2)$. We know $(M', I') \in \mathbf{Mod}(SP_1')$ and $(M', I') \in \mathbf{Mod}(SP_2)$. By the first assumption we know that there exists $I$ such that $(M', I) \in \mathbf{Mod}(SP_1)$ with $I \sqsubseteq I'$. As $SP_1$ is refinement closed, we know $(M', I') \in \mathbf{Mod}(SP_1)$, thus $(M', I') \in \mathbf{Mod}(SP_1 \textbf{ and } SP_2)$. Finally as $I' \sqsubseteq I'$, we can conclude $(M', I') \in \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$.

**Structured rename rule** We show $\mathbf{Mod}(SP' \textbf{ rename } \theta) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP \textbf{ rename } \theta))$. Let $(M', I') \in \mathbf{Mod}(SP' \textbf{ rename } \theta)$, then we know $(M', I')|_\theta \in \mathbf{Mod}(SP')$. By assumption we know there exists $I$ such that $(M'|_\sigma, I) \in \mathbf{Mod}(SP)$ with $I \sqsubseteq I'|_\theta$.

We now construct $J'$ relative to $M'$ as:

$$J'(n'(a_1', \ldots, a_k')) := \begin{cases} \alpha^{\mathcal{D}}_{\sigma,M'}(I(n(\alpha_{\sigma,M'}{}^{-1}(a_1'), \ldots, \alpha_{\sigma,M'}{}^{-1}(a_k')))) & \text{if } n \in \Sigma_{CC} \text{ with } \nu(n) = n' \\ \mathit{Top}^{\mathcal{D}}_{comms'} & \text{otherwise} \end{cases}$$

for all process names $n'_{w',comms'} \in \Sigma'_{CC}$ and alphabet elements $a_1', \ldots, a_k' \in \overline{s'_1}_{M'} \times \ldots \times \overline{s'_k}_{M'}$ where $w' = \langle s_1', \ldots, s_k' \rangle$.

This definition is well formed as $\theta$ is injective on process names, thus there is no ambiguity in the choice of denotations. The partial inverses of parameters are defined thanks to the strict translation of the parameter sorts of process names. As $(M'|_\sigma, I)$ satisfies the controlled traces condition (a requirement of CSP-CASL models) and as $cTr^{\mathcal{D}}(\mathit{Top}^{\mathcal{D}}_{comms'}) \in \mathcal{T}(comms')$ (Lemma 7.19), it follows that $J'$ also satisfies the controlled traces condition, and is thus a CSP-CASL model.

By the construction of $J'$ we know that $J' \sqsubseteq I'$. This is the case because for names in $\Sigma_{CC}$, we know $J'$'s denotation matches $I$'s denotation (after translation by $\alpha^{\mathcal{D}}_{\sigma,M'}$) which refines to $I'$'s reduced denotation. For all other names not in $\Sigma_{CC}$ we know that $J'$ denotation is $\mathit{Top}^{\mathcal{D}}_{comms'}$ for which $I'$ denotation will be a refinement (see Lemma 7.19). We also know from the construction and Lemma 7.16 that $J'|_\theta = I$. We now have $(M'|_\sigma, J'|_\theta) = (M', J')|_\theta \in \mathbf{Mod}(SP)$, thus $(M', J') \in \mathbf{Mod}(SP \textbf{ rename } \theta)$. Finally as $J' \sqsubseteq I'$ we know $(M', I') \in \mathbf{Mod}(\mathit{RefCl}(SP \textbf{ rename } \theta))$.

**Structured hide rule** We show $\mathbf{Mod}(SP' \textbf{ hide } \theta) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP \textbf{ hide } \theta))$. Let $(M, I) \in \mathbf{Mod}(SP' \textbf{ hide } \theta)$ then we know there exists $(M', I') \in \mathbf{Mod}(SP')$ such that $(M, I) = (M', I')|_\theta$. By assumption we know there exists $J'$ such that $(M', J') \in \mathbf{Mod}(SP)$ with $J' \sqsubseteq I'$. Therefore $(M, J'|_\theta) \in \mathbf{Mod}(SP \textbf{ hide } \theta)$. As $J' \sqsubseteq I'$ then we also know $J'|_\theta \sqsubseteq I'|_\theta = I$, thus we can conclude that $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(SP \textbf{ hide } \theta))$. □

The rules for **and** involve rather strong preconditions, where we hope that it will be possible to obtain better results in the future.

Renaming and refinement involving the same signature morphism can be exchanged. Furthermore, compositional translations can be collapsed into one, and the specification union operator distributes over translation.

**Lemma 9.4** The following statements hold:

1. $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$ implies $SP \rightsquigarrow^{\theta} SP'$. Furthermore, the converse is not true in general.

2. Provided that $\theta$ is injective on process names, we also have:
   $SP \rightsquigarrow^{\theta} SP'$ implies $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$.

3. $(SP \textbf{ rename } \theta_1) \textbf{ rename } \theta_2 \equiv SP \textbf{ rename } (\theta_2 \circ \theta_1)$.

4. $(SP_1 \textbf{ and } SP_2) \textbf{ rename } \theta \equiv (SP_1 \textbf{ rename } \theta) \textbf{ and } (SP_2 \textbf{ rename } \theta)$.

*Proof.* Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ be a CSP-CASL signature morphism and let $SP$ and $SP'$ be $\Sigma_{CC}$- and $\Sigma'_{CC}$-specifications respectively. We show each implication separately

**Statement 1** We show $SP \rightsquigarrow^{\theta} SP'$, that is, $Mod(SP')|_{\theta} \subseteq \textbf{Mod}(\mathit{RefCl}(SP))$. Let $(M, I) \in \textbf{Mod}(SP')|_{\theta}$, then there exists $(M', I') \in \textbf{Mod}(SP')$ such that $(M', I')|_{\theta} = (M, I)$. By assumption we know $(M', I') \in \textbf{Mod}(\mathit{RefCl}(SP \textbf{ rename } \theta))$, thus we know there exists $J'$ such that $(M', J') \in \textbf{Mod}(SP \textbf{ rename } \theta)$ and $J' \sqsubseteq I'$. By the definition of the structured **rename** operator we know $(M, J'|_{\theta}) \in \textbf{Mod}(SP)$. As $J'|_{\theta} \sqsubseteq I$, we can conclude $(M, I) \in \textbf{Mod}(\mathit{RefCl}(SP))$.

We now show the converse, $SP \rightsquigarrow^{\theta} SP'$ implies $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$, is not true in general. In particular signature morphisms which are non-injective on process names are problematic. Consider the following two CSP-CASL specifications:

**spec** SP =
  **data free type** $s ::= a \mid b$
  **process** $P : s$; $Q : s$;
    $P = a \to STOP$;
    $Q = a \to STOP \sqcap b \to STOP$
**end**

**spec** SP′ =
  **data free type** $s ::= a \mid b$
  **process** $P' : s$;
    $P' = a \to STOP$
**end**

There is a signature morphism from the signature of $SP$ to the signature of $SP'$ where the process names $P$ and $Q$ are both mapped to $P'$. Each specification has a single model. It is easy to show $SP \rightsquigarrow^{\theta} SP'$. However, $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$ does not hold as $\textbf{Mod}(SP \textbf{ rename } \theta) = \emptyset$. Thanks to the signature morphism, any model $(M', I') \in \textbf{Mod}(SP \textbf{ rename } \theta)$ would have to give an interpretation to $P'$ such that $I'|_{\theta}(P) = I'|_{\theta}(Q)$, which is impossible due to the process equations in $SP$.

**Statement 2** We show $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$, That is, by definition, $\textbf{Mod}(SP') \subseteq \textbf{Mod}(\mathit{RefCl}(SP \textbf{ rename } \theta))$. Let $(M', I') \in \textbf{Mod}(SP')$, then we know by assumption that $(M', I')|_\theta \in \textbf{Mod}(\mathit{RefCl}(SP))$. Thus, we know there exists $I$ such that $(M'|_\sigma, I) \in \textbf{Mod}(SP)$ and $I \sqsubseteq I'|_\theta$.

We now construct $J'$ (relative to $M'$) as is done in the proof of the structured **rename** rule of Lemma 9.3, that is,

$$J'(n'(a'_1, \ldots, a'_k)) := \begin{cases} \alpha^{\mathcal{D}}_{\sigma,M'}(I(n(\alpha_{\sigma,M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma,M'}{}^{-1}(a'_k)))) & \text{if } n \in \Sigma_{CC} \text{ with } \nu(n) = n' \\ \mathit{Top}^{\mathcal{D}}_{comms'} & \text{otherwise} \end{cases}$$

for all process names $n'_{w',comms'} \in \Sigma'_{CC}$ and alphabet elements $a'_1, \ldots, a'_k \in \overline{s'_1}_{M'} \times \ldots \times \overline{s'_k}_{M'}$ where $w' = \langle s'_1, \ldots, s'_k \rangle$.

By the construction of $J'$ we know that $(M', J')$ is a CSP-CASL model, $J' \sqsubseteq I'$ and $J'|_\theta = I$. Finally, we have $(M'|_\sigma, J'|_\theta) = (M', J')|_\theta \in \textbf{Mod}(SP)$. Therefore $(M', J') \in \textbf{Mod}(SP \textbf{ rename } \theta)$. As $J' \sqsubseteq I'$, we can conclude $(M', I') \in \textbf{Mod}(\mathit{RefCl}(SP \textbf{ rename } \theta))$.

**Statements 3 and 4** Follows from the definition of the structuring operator and are institution independent. □

## 9.4 Compositional Deadlock Analysis

The deadlock analysis presented in [KR09] is practically limited to dealing with a small number of processes in parallel. It involves the construction of a so-called sequential process – which has a size that is exponential in the number of parallel components involved (this method was used in the deadlock analysis of the EP2 dialogue in Section 5.4). Here, we prove deadlock freedom in a far more elegant way.

For the rest of this section, as usual for deadlock analysis in the context of CSP, we work in the Stable-Failures semantics $\mathcal{F}$ only. Furthermore, we assume all processes and process terms to be divergence free.

Within this section we use the notation of $\mathit{failures}(\_)$ in place of $\mathit{snd}([\![\_]\!]_{\mathcal{F}})$. Thus, we often write $\mathit{failures}([\![pt]\!]_{(M,I),\mu_G,\mu_L})$ to mean the failures of the process term resulting from the CASL evaluation of the process term $pt$ with respect to model $(M, I)$ and valuations $\mu_G$ and $\mu_L$, that is, $\mathit{failures}([\![pt]\!]_{(M,I),\mu_G,\mu_L}) = \mathit{snd}([\![[\![pt]\!]_{M,\mu_G,\mu_L}]\!]_{\mathcal{F},I})$. In this respect we also write $\alpha_{\sigma,M}(\mathit{failures}([\![pt]\!]_{(M,I),\mu_G,\mu_L}))$ for $\mathit{snd}(\alpha^{\mathcal{F}}_{\sigma,M}([\![[\![pt]\!]_{M,\mu_G,\mu_L}]\!]_{\mathcal{F},I}))$ (similarly for the contravariant translation) to ease notation.

### 9.4.1 Deadlock Freedom in Structured CSP-CASL Specifications

We first define what it means for a process term to be deadlock free in the context of a CSP-CASL specification (be it basic or structured). We then present a collection of proof rules for deadlock analysis over the structuring operators.

Let $SP$ be a CSP-CASL specification with signature $\Sigma_{CC}$, $X_G$ and $X_L$ be global and local variable systems respectively over $\Sigma_{CC}$, and let $pt$ be a process term over signature $\Sigma_{CC}$ with variable systems $X_G$ and $X_L$. We say: $pt$ is *deadlock free* in specification $SP$, written as

$$pt\ isDFin\ SP$$

if for all models $(M, I) \in \mathbf{Mod}(SP)$, for all variable valuations $\mu_G : X_G \to M_\perp$ and $\mu_L : X_L \to M_\perp$, and for all traces $s \in \mathit{Alph}(M)^*$ it holds

$$(s, \mathit{Alph}(M)^\checkmark) \notin failures([\![pt]\!]_{(M,I),\mu_G,\mu_L}) \ .$$

We now show that deadlock freedom is compatible with the structuring operations:

**Lemma 9.5** The following proof rules are sound:

$$\frac{SP \rightsquigarrow^{\theta}_{\mathcal{F}} SP' \qquad pt\ isDFin\ SP}{\theta(pt)\ isDFin\ SP'} \qquad \frac{pt\ isDFin\ SP_1}{pt\ isDFin\ (SP_1\ \mathbf{and}\ SP_2)}$$

$$\frac{pt\ isDFin\ SP}{\theta(pt)\ isDFin\ (SP\ \mathbf{rename}\ \theta)} \qquad \frac{\theta(pt)\ isDFin\ SP'}{pt\ isDFin\ (SP'\ \mathbf{hide}\ \theta)}$$

where $\theta : \Sigma_{CC} \to \Sigma'_{CC}$ is a CSP-CASL signature morphism.

*Proof.* Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ be a CSP-CASL signature morphism, $X_G$ and $X_L$ be global and local variable systems respectively, and let $pt$ be a process term over $\Sigma_{CC}$ with $X_G$ and $X_L$. We prove each proof rule individually.

**Refinement rule** Let $SP$ be a $\Sigma_{CC}$-specification and $SP'$ be a $\Sigma'_{CC}$-specification. Assume $\theta(pt)$ has a deadlock in $SP'$, then we know there exists a model $(M', I') \in \mathbf{Mod}(SP')$, valuations $\mu_G : \sigma(X_G) \to M'_\perp$ and $\mu_L : \sigma(X_L) \to M'_\perp$, and a trace $s' \in \mathit{Alph}(M')^*$ such that

$$(s', \mathit{Alph}(M')^\checkmark) \in failures([\![\theta(pt)]\!]_{(M',I'),\mu_G,\mu_L}) \ .$$

As $SP$ refines to $SP'$ we know that $(M', I') \in \mathbf{Mod}(\mathit{RefCl}(SP))$, thus there exists $I$ such that $(M|_\sigma, I) \in \mathbf{Mod}(SP)$ with $I \sqsubseteq I'|_\theta$.

By the reduct property (Lemma 8.11) we know

$$failures([\![pt]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma}) = \hat{\alpha}_{\sigma,M'}(failures([\![\theta(pt)]\!]_{(M',I'),\mu_G,\mu_L})) \ .$$

Thanks to the controlled traces we know $\hat{\alpha}_{\sigma,M'}$ preserves deadlocks (Lemma 7.20), thus we obtain

$$(s, \mathit{Alph}(M'|_\sigma)^\checkmark) \in failures([\![pt]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma})$$

where $s' = \alpha^{*\checkmark}_{\sigma,M'}(s)$ for some $s$ (as $pt$ is a process term over $\Sigma_{CC}$ there must be a corresponding $s$ for $s'$).

As $I \sqsubseteq I'|_\theta$, and the given refinement is monotonic with respect to the CSP operators [Ros05], it follows that

$$failures([\![pt]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma}) \subseteq failures([\![pt]\!]_{(M'|_\sigma,I),\mu_G|_\sigma,\mu_L|_\sigma}) \ ,$$

thus

$$(s, Alph(M'|_\sigma)^\checkmark) \in failures([\![pt]\!]_{(M'|_\sigma,I),\mu_G|_\sigma,\mu_L|_\sigma}) \ .$$

This is a witness that $pt$ has a deadlock in $SP$. This is a contradiction and we conclude that $\theta(pt)$ is deadlock free in $SP'$.

**Structured and rule** As $pt$ is deadlock free in $SP_1$ and the structured **and** operator only restricts model classes (and thus cannot introduce a model with a deadlock), it follows that $pt$ is deadlock free in $SP_1$ **and** $SP_2$.

**Structured rename rule** Let $SP$ be a $\Sigma_{CC}$-specification. Assume the process term $\theta(pt)$ has a deadlock in ($SP$ **rename** $\theta$), the we know there exists a model $(M', I') \in$ **Mod**($SP$ **rename** $\theta$), valuations $\mu_G : \sigma(X_G) \to M'_\bot$ and $\mu_L : \sigma(X_L) \to M'_\bot$, and a trace $s' \in Alph(M')^*$ such that

$$(s', Alph(M')^\checkmark) \in failures([\![\theta(pt)]\!]_{(M',I'),\mu_G,\mu_L}) \ .$$

As we have a renaming we also know $(M', I')|_\theta \in$ **Mod**($SP$). By the reduct property (Lemma 8.11) we know

$$failures([\![pt]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma}) = \hat{\alpha}_{\sigma,M'}(failures([\![\theta(pt)]\!]_{(M',I'),\mu_G,\mu_L})) \ .$$

As $\hat{\alpha}_{\sigma,M'}$ preserves deadlocks (Lemma 7.20) we obtain

$$(s, Alph(M'|_\sigma)^\checkmark) \in failures([\![pt]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma})$$

where $s' = \alpha^{*\checkmark}_{\sigma,M'}(s)$ for some $s$ (as $pt$ is a process term over $\Sigma_{CC}$ there must be a corresponding $s$ for $s'$). This is a witness that $pt$ has a deadlock in $SP$. This is a contradiction and we conclude that $\theta(pt)$ is deadlock free in ($SP$ **rename** $\theta$).

**Structured hide rule** Let $SP'$ be a CSP-CASL specification with signature $\Sigma'_{CC}$. Assume $pt$ has a deadlock in ($SP'$ **hide** $\theta$), then we know there exists a model $(M, I) \in$ **Mod**($SP'$ **hide** $\theta$), valuations $\mu_G|_\sigma : X_G \to M_\bot$ and $\mu_L|_\sigma : X_L \to M_\bot$, and a trace $s \in Alph(M)^*$ such that

$$(s, Alph(M)^\checkmark) \in failures([\![pt]\!]_{(M,I),\mu_G|_\sigma,\mu_L|_\sigma}) \ .$$

By the definition of the **hide** operator we know there exists $(M', I') \in$ **Mod**($SP'$) such that $(M', I')|_\theta = (M, I)$. By the reduct property (Lemma 8.11) we know

$$failures([\![pt]\!]_{(M,I),\mu_G|_\sigma,\mu_L|_\sigma}) = \hat{\alpha}_{\sigma,M'}(failures([\![\theta(pt)]\!]_{(M',I'),\mu_G,\mu_L})) \ .$$

As $\hat{\alpha}_{\sigma,M'}$ preserves deadlocks (Lemma 7.20), thus we obtain

$$(s', Alph(M')^\checkmark) \in failures([\![\theta(pt)]\!]_{(M',I'),\mu_G,\mu_L}) \ .$$

where $s' = \alpha^{*\checkmark}_{\sigma,M'}(s)$. This is a witness that $\theta(pt)$ has a deadlock in $SP'$. This is a contradiction and we conclude that $pt$ is deadlock free in ($SP'$ **hide** $\theta$). □

The above proof rules allow one to show deadlock freedom by decomposing structured specifications. However, it may still be a difficult task to prove deadlock freedom for complex systems involving parallel processes. We describe a technique for dealing with this situation in the following section.

### 9.4.2 Deadlock Analysis of Networks

In order to study deadlock analysis of networks of processes, we lift a definition originally formulated over CSP in [RSR04] to CSP-CASL. This captures the notion of processes being responsive to one another. For example, a server $Q$ is responsive to a client $P$ if whenever the client needs participation from the server, the server is prepared to engage in it. That is, the sever does not introduce a new deadlock.

We first define what it means for one process to be responsive to another. Let $P$ and $Q$ be process terms over signature $\Sigma_{CC}$ with global and local variable systems $X_G$ and $X_L$ respectively. Let $A_P$ and $A_Q$ be downward and upward closed super sets of the constituent alphabets of the process terms $P$ and $Q$ respectively (i.e., $sorts(P) \subseteq A_P$, $\downarrow A_P = A_P$, and $\uparrow A_P = A_P$, similar for $A_Q$)[2], and let $J = A_P \cap A_Q$ be the set of all shared communications sorts. Finally, let $J' \in J_\downarrow$ and $X = \overline{J'} \cup \{\checkmark\}$. Then, we define the *responds to live* property as:

$$Q :: A_Q \; ResToLive^{\checkmark} \; P :: A_P \; on \; J' \; in \; SP$$

if for all models $(M, I) \in \mathbf{Mod}(SP)$, all variable valuations $\mu_G : X_G \to M_\perp$ and $\mu_L : X_L \to M_\perp$, and for all traces $s \in Alph(M)^{\checkmark}$ it holds that

$$(s, X) \in failures([\![P \,|[\, J \,]|\, Q]\!]_{(M,I),\mu_G,\mu_L}) \Longrightarrow (s, X) \in failures([\![P]\!]_{(M,I),\mu_G,\mu_L})$$

A server $Q$ is responsive to a client $P$ if any deadlock occurring in the parallel combination of the server and client originates from the client alone.

With this notion it is possible to establish deadlock freedom of networks in a compositional way (lifted from the CSP level [RSR04] to CSP-CASL). We can start with a deadlock free network and add on responsive processes without adding deadlocks.

**Lemma 9.6** Let $SP$ be a CSP-CASL specification with signature $\Sigma_{CC}$. Moreover, let $P_i$ (for $1 \leq i \leq k$) and $Q$ be process terms over $\Sigma_{CC}$ with global and local variable systems $X_G$ and $X_L$ respectively. Finally, let $A_i$ and $A_Q$ be downward and upwards closed super-sets of the constituent alphabet of the process terms $P_i$ for $1 \leq i \leq k$ and $Q$ respectively.[3] If

- $A_i \cap A_j \cap A_Q = \emptyset$ for all $i$ and $j$ where $1 \leq i, j \leq k$ and $i \neq j$,
- $A_i \cap A_Q \neq \emptyset$ for at least one $i$ where $1 \leq i \leq k$,
- $Network(\{(P_1, A_1), \ldots, (P_k, A_k)\})$ $isDFin$ $SP$, and

[2]Upward closure is defined in the obvious way: $\uparrow X = \{y \in S \mid \exists x \in X \bullet x \leq y\}$. The condition "upward and downward closed" is required due to CASL sub-sorting. It ensures that the sort set $J$ comprises all shared communications.

[3]In the original publication by Roscoe [RSR04] they use $J$ as the alphabet of $Q$. We use $A_Q$ here for clarity to avoid a clash in the previous section with $J$ representing shared events.

- $Q :: A_Q$ $ResToLive^{\checkmark}$ $P_i :: A_i$ *on* $(A_i \cap A_Q)$ *in* $SP$ for each $i$ where $1 \leq i \leq k$ and $A_i \cap A_Q \neq \emptyset$

then $Network(\{(P_1, A_1), \ldots, (P_k, A_k), (Q, A_Q)\})$ $isDFin$ $SP$.

*Proof.* We first introduce some convenient shorthands for this proof. Let $N'$ be shorthand for $Network(\{(P_1, A_1), \ldots, (P_k, A_k), (Q, A_Q)\})$, $N$ be shorthand for $Network(\{(P_1, A_1), \ldots, (P_k, A_k)\})$, and $A = \bigcup_{i=1}^{k} A_i$ be shorthand for the combined alphabet of each process term's alphabet $A_i$.

Now assume $N'$ has a deadlock, then we know there exists a model $(M, I)$, valuations $\mu_G : X_G \to M_\perp$ and $\mu_L : X_L \to M_\perp$ and a trace $s \in \mathit{Alph}(M)^*$ such that

$$(s, \mathit{Alph}(M)^{\checkmark}) \in failures([\![N']\!]_{(M,I),\mu_G,\mu_L}) \ .$$

We can decompose this failure into failures contributed by the components of the network. That is, there exists two failures $(s', X) \in failures([\![N]\!]_{(M,I),\mu_G,\mu_L})$ and $(s'', Y) \in failures([\![Q]\!]_{(M,I),\mu_G,\mu_L})$ such that $s' = s \upharpoonright \overline{A}$ and $s'' = s \upharpoonright \overline{A_Q}$ and where $X$ and $Y$ are maximal refusal sets. We can further decompose the failure $(s', X)$ into failures $(s_i, X_i) \in failures([\![P_i]\!]_{(M,I),\mu_G,\mu_L})$ such that $s_i = s' \upharpoonright \overline{A_i} = s \upharpoonright \overline{A_i}$ and where $X_i$ are maximal refusals.

We know that $X \cup Y = \mathit{Alph}(M)^{\checkmark}$ as we decomposed a deadlocked failure. From this we can derive the fact $(X \cap \overline{A}^{\checkmark}) \cup (Y \cap \overline{A_Q}^{\checkmark}) = (\overline{A} \cup \overline{A_Q})^{\checkmark}$. As a consequence of the maximality of the failures, we know that each of them is either $\mathit{Alph}(M)$ or contains $\checkmark$. As $N$ is deadlock free in $SP$, we also know that $X \neq \mathit{Alph}(M)^{\checkmark}$. This leaves us with two cases, either $X = \mathit{Alph}(M)$ and no $P_i$ can refuse $\checkmark$ (i.e., $\checkmark \notin X_i$ for all $1 \leq i \leq k$), or $\checkmark \in X$ and $\emptyset \neq (\overline{A}^{\checkmark} - X) \subseteq \overline{A_Q}$. We now consider these two cases separately.

Consider the first case. By assumption we know there exists $i$ such that $A_i \cap A_Q \neq \emptyset$, thus for this $i$ we know $\overline{A_i} \cap \overline{A_Q} \neq \emptyset$. We also know $\checkmark \in Y$ as $X \cup Y = \mathit{Alph}(M)^{\checkmark}$. From these facts it follows that

$$(s_i, (\overline{A_i} \cap \overline{A_Q})^{\checkmark}) \in failures([\![P_i \,|[\, A_i \cap A_Q \,]|\, Q]\!]_{(M,I),\mu_G,\mu_L})$$

and

$$(s_i, (\overline{A_i} \cap \overline{A_Q})^{\checkmark}) \notin failures([\![P_i]\!]_{(M,I),\mu_G,\mu_L}) \ .$$

This contradicts our assumption that $Q$ is responsive to $P$, thus this case is impossible.

In the second case we know there exists $i$ such that $(\overline{A_i} - X_i) \cap \overline{A_Q} \neq \emptyset$. Furthermore we can establish that $\checkmark \in X_i$. From these facts we can establish $\overline{A_i} - X_i \subseteq Y$, which leads us to the fact $(\overline{A_i} \cap \overline{A_Q})^{\checkmark} \subseteq X_i \cup Y$. This leads us to the same contradiction as in the first case and we conclude that $N'$ is indeed deadlock free in $SP$. □

This lemma provides an elegant proof technique: we start with a network we want to show is deadlock free, we identify a responsive process and pull it out. It remains to show that the sub-network is deadlock free. This can be repeated until the network consists of a single component, at which point deadlock freedom is easy to prove (as there is no parallel combination). The property responds to live has a characterisation in terms of refinement [RSR04] and thus,

can be proven, for example, by CSP-CASL-Prover; the conditions concerning communication alphabets can be proven algorithmically.

In order to use this proof technique, one must establish responsiveness of processes with respect to structured specifications. To ease this burden we provide a proof calculus tailored for the property responds to live:

**Lemma 9.7** The following proof rules are sound:

$$\frac{Q :: A_Q \; ResToLive^{\checkmark} \; P :: A_P \; on \; J' \; in \; SP_1}{Q :: A_Q \; ResToLive^{\checkmark} \; P :: A_P \; on \; J' \; in \; (SP_1 \; \mathbf{and} \; SP_2)}$$

$$\frac{Q :: A_Q \; ResToLive^{\checkmark} \; P :: A_P \; on \; J' \; in \; SP}{\theta(Q) :: \sigma(A_Q) \; ResToLive^{\checkmark} \; \theta(P) :: \sigma(A_P) \; on \; \sigma(J') \; in \; (SP \; \mathbf{rename} \; \theta)}$$

$$\frac{\theta(Q) :: \sigma(A_Q) \; ResToLive^{\checkmark} \; \theta(P) :: \sigma(A_P) \; on \; \sigma(J') \; in \; SP'}{Q :: A_Q \; ResToLive^{\checkmark} \; P :: A_P \; on \; J' \; in \; (SP' \; \mathbf{hide} \; \theta)}$$

where $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$.

*Proof.* Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ be a CSP-CASL signature morphism, $X_G$ and $X_L$ be global and local variable systems respectively, and let $pt$ be a process term over $\Sigma_{CC}$ with $X_G$ and $X_L$. We prove each proof rule individually. In the following let $J = A_P \cap A_Q$.

**Structured and rule** As $Q$ is responsive to $P$ on $J'$ in $SP_1$ and the structured **and** operator only restricts model classes, it follows that $Q$ is responsive to $P$ on $J'$ in $SP_1$ **and** $SP_2$.

**Structured rename rule** Let $SP$ be a $\Sigma_{CC}$-specification and $(M', I')$ be a $\Sigma'_{CC}$-model in $\mathbf{Mod}(SP \; \mathbf{rename} \; \theta)$. Furthermore, let $\mu_G : \sigma(X_G) \to M'_\perp$ and $\mu_L : \sigma(X_L) \to M'_\perp$ be valuations, and $s' \in Alph(M')^{*\checkmark}$ be a trace such that

$$(s', \overline{\sigma(J')}_{M'} \cup \{\checkmark\}) \in failures([\![\theta(P) \,|[\, \sigma(J) \,]|\, \theta(Q)]\!]_{(M',I'),\mu_G,\mu_L}) \; .$$

As we have a renaming we also know $(M', I')|_\theta \in \mathbf{Mod}(SP)$. By the reduct property, (Lemma 8.11) we know

$$failures([\![P \,|[\, J \,]|\, Q]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma}) = \hat{\alpha}_{\sigma,M'}(failures([\![\theta(P) \,|[\, \sigma(J) \,]|\, \theta(Q)]\!]_{(M',I'),\mu_G,\mu_L})) \; .$$

By applying Lemma 7.20, and the fact that $\overline{\sigma(J')}_{M'} \cup \{\checkmark\} = \alpha^{\mathcal{P}\checkmark}_{\sigma,M'}(\overline{J'}_{M'|_\sigma} \cup \{\checkmark\})$, we obtain

$$(s, \overline{J'}_{(M'|_\sigma)} \cup \{\checkmark\}) \in failures([\![P \,|[\, J \,]|\, Q]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma})$$

where $s' = \alpha^{*\checkmark}_{\sigma,M'}(s)$. By assumption we know

$$(s, \overline{J'}_{M'|_\sigma} \cup \{\checkmark\}) \in failures([\![P]\!]_{(M',I')|_\theta,\mu_G|_\sigma,\mu_L|_\sigma}) \; .$$

Finally, by applying the reduct property (Lemma 8.11) and Lemma 7.20 in reverse, we obtain

$$(s', \overline{\sigma(J')}_{M'} \cup \{\checkmark\}) \in failures([\![\theta(P)]\!]_{(M',I'),\mu_G,\mu_L}) \ .$$

We conclude that $\theta(Q)$ is responsive to $\theta(P)$ on $J'$ in $(SP \textbf{ rename } \theta)$.

**Structured hide rule** Let $SP'$ be a CSP-CASL specification with signature $\Sigma'_{CC}$, and let $(M, I) \in \textbf{Mod}(SP' \textbf{ hide } \theta)$ be a model. Furthermore, let $\mu_G|_\sigma : X_G \to M_\perp$ and $\mu_L|_\sigma : X_L \to M_\perp$ be valuations, and $s \in Alph(M)^{*\checkmark}$ be a trace such that

$$(s, \overline{J'}_M \cup \{\checkmark\}) \in failures([\![P \,|[\, J \,]|\, Q]\!]_{(M,I),\mu_G|_\sigma,\mu_L|_\sigma}) \ .$$

We know there exists a model $(M', I') \in \textbf{Mod}(SP')$ such that $(M', I')|_\theta = (M, I)$. By the reduct property (Lemma 8.11) we know

$$failures([\![P \,|[\, J \,]|\, Q]\!]_{(M,I),\mu_G|_\sigma,\mu_L|_\sigma}) = \hat{\alpha}_{\sigma,M'}(failures([\![\theta(P) \,|[\, \sigma(J) \,]|\, \theta(Q)]\!]_{(M',I'),\mu_G,\mu_L})) \ .$$

By applying Lemma 7.20, and the fact that $\overline{\sigma(J')}_{M'} \cup \{\checkmark\} = \alpha^{\mathcal{P}\checkmark}_{\sigma,M'}(\overline{J'}_M \cup \{\checkmark\})$, we obtain

$$(s', \overline{\sigma(J')}_{M'} \cup \{\checkmark\}) \in failures([\![\theta(P) \,|[\, \sigma(J) \,]|\, \theta(Q)]\!]_{(M',I'),\mu_G,\mu_L})$$

where $s' = \alpha^{*\checkmark}_{\sigma,M'}(s)$. By assumption we know

$$(s', \overline{\sigma(J')}_{M'} \cup \{\checkmark\}) \in failures([\![\theta(P)]\!]_{(M',I'),\mu_G,\mu_L}) \ .$$

Finally, by applying the reduct property (Lemma 8.11) and Lemma 7.20 in reverse, we obtain

$$(s, \overline{J'}_M \cup \{\checkmark\}) \in failures([\![P]\!]_{(M,I),\mu_G|_\sigma,\mu_L|_\sigma}) \ .$$

We conclude that $Q$ is responsive to $P$ on $J'$ in $(SP' \textbf{ hide } \theta)$. □

This proof technique allows deadlock freedom of processes to be established relatively to structured CSP-CASL specifications in an elegant and compositional manner. We show an example of using this technique in Chapter 10.

The above network lemma (Lemma 9.6) and calculi illustrate the successful application of techniques from CSP to CSP-CASL and the institution independent structuring mechanisms. We expect other techniques from CSP to also lift successfully to CSP-CASL.

## 9.5 A Complete Refinement Calculus

Here, we develop a proof calculus which allows for the compositional reasoning of refinements in Structured CSP-CASL, following the style of Bidoit et al. [BCH99]. We show not only soundness of this calculus, but also completeness provided that the structured specifications are restricted to certain (mostly) reasonable constructions principles.

First, we define the notion of *conservative extension.* A CSP-CASL Specification $SP'$ is a *conservative extension* of a specification $SP$ (written $SP'$ *ConsExt* $SP$) if $\mathbf{Sig}(SP) \subseteq \mathbf{Sig}(SP')$ and for all models $(M, I) \in \mathbf{Mod}(SP)$ there exists a model $(M', I') \in \mathbf{Mod}(SP')$ such that $(M', I')|_\theta = (M, I)$, where $\theta$ is the embedding signature morphism from $\mathbf{Sig}(SP)$ to $\mathbf{Sig}(SP')$.

In the following lemmas we assume a proof system which defines a relation $SPI \vdash \varphi$ between structured CSP-CASL specifications and formulae.

**Lemma 9.8** The following proof rules are sound over $\mathcal{T}, \mathcal{N}$, and $\mathcal{F}$ relative to a sound calculus for $SPI \vdash \varphi$.

$$\frac{SPI \vdash \varphi \text{ for all } \varphi \in \mathbf{Ax}(D, P)}{(D, P) \rightsquigarrow SPI}$$

$$\frac{SP' \rightsquigarrow SPI' \qquad SPI' \textit{ ConsExt } SPI}{SP' \textbf{ hide } \theta \rightsquigarrow SPI} \quad \begin{array}{l}\text{where } \theta \text{ is} \\ \text{the embedding from} \\ \mathbf{Sig}(SPI) \text{ to } \mathbf{Sig}(SP')\end{array}$$

$$\frac{SP \rightsquigarrow SPI' \textbf{ hide } \theta \qquad \theta \text{ is injective on } \textit{process names}}{SP \textbf{ rename } \theta \rightsquigarrow SPI'}$$

$$\frac{\begin{array}{cc} SP_1 \rightsquigarrow SPI' \textbf{ hide } \theta_1 & SP_2 \rightsquigarrow SPI' \textbf{ hide } \theta_2 \\ \theta_1 \text{ and } \theta_2 \text{ are injective on } \textit{process names} & \textit{image}(\nu_1) \cap \textit{image}(\nu_2) = \emptyset \end{array}}{(SP_1 \textbf{ rename } \theta_1) \textbf{ and } (SP_2 \textbf{ rename } \theta_2) \rightsquigarrow SPI'} \quad \begin{array}{l}\theta_1 = (\sigma_1, \nu_1), \\ \theta_2 = (\sigma_2, \nu_2)\end{array}$$

$$\frac{\begin{array}{cc} SP_1 \rightsquigarrow SPI & SP_2 \rightsquigarrow SPI \\ \text{ProcConst}(SP_1, SP_2) & \text{Single-valued}(SP_i) \text{ for } i = 1 \vee i = 2 \end{array}}{SP_1 \textbf{ and } SP_2 \rightsquigarrow SPI}$$

Note that in the 2nd rule, the signature of $SP'$ must be equal to the signature of $SPI'$ as the refinement does not involve a signature morphism.

*Proof.* We prove each rule individually.

**Proof of rule 1** Assume $SPI \vdash \varphi$ for all $\varphi \in \mathbf{Ax}(D, P)$, we show $(D, P) \rightsquigarrow SPI$, that is, $\mathbf{Mod}(SPI) \subseteq \mathbf{Mod}(\textit{RefCl}(D, P))$. Let $(M, I) \in \mathbf{Mod}(SPI)$. We claim $(M, I) \in \mathbf{Mod}(D, P) \subseteq \mathbf{Mod}(\textit{RefCl}(D, P))$. To this end, let $\varphi \in \mathbf{Ax}(D, P)$. We show $(M, I) \models \varphi$. Since $SPI \vdash \varphi$ and by the soundness of $\vdash$ we can conclude $(M, I) \models \varphi$.

**Proof of rule 2** Assume $SP' \rightsquigarrow SPI'$ and $SPI'$ *ConsExt* $SPI$. We show $\mathbf{Mod}(SPI) \subseteq \mathbf{Mod}(\textit{RefCl}(SP' \textbf{ hide } \theta))$. Let $(M, I) \in \mathbf{Mod}(SPI)$. By conservative extension we know there exists $(M', I') \in \mathbf{Mod}(SPI')$ such that $(M', I')|_\theta = (M, I)$. By assumption we know $(M', I') \in \mathbf{Mod}(\textit{RefCl}(SP'))$. Therefore there exists $(M', J') \in \mathbf{Mod}(SP')$ such that $J' \sqsubseteq I'$. By the definition of hiding we know $(M, J'|_\theta) \in \mathbf{Mod}(SP' \textbf{ hide } \theta)$. As $J' \sqsubseteq I'$ we know $J'|_\theta \sqsubseteq I'|_\theta = I$, thus $(M, I) \in \mathbf{Mod}(\textit{RefCl}(SP' \textbf{ hide } \theta))$.

**Proof of rule 3** Assume $SP \rightsquigarrow SPI'$ **hide** $\theta$ and $\theta$ is injective on *process names*. We show **Mod**$(SPI') \subseteq$ **Mod**$(RefCl(SP$ **rename** $\theta))$. Let $(M', I') \in$ **Mod**$(SPI')$. By definition of hiding we have $(M', I')|_\theta \in$ **Mod**$(SPI'$ **hide** $\theta)$. By assumption we know $(M', I')|_\theta \in$ **Mod**$(RefCl(SP))$. Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$, then there exists $(M'|_\sigma, I) \in$ **Mod**$(SP)$ such that $I \sqsubseteq I'|_\theta$.

We now construct a witness $J'$ which allows us to show $(M', I') \in$ **Mod**$(RefCl(SP$ **rename** $\theta))$. To this end construct $J'$ (relative to $M'$) as is done in the proof of the structured **rename** rule of Lemma 9.3, that is,

$$J'(n'(a'_1, \ldots, a'_k)) := \begin{cases} \alpha^{\mathcal{D}}_{\sigma,M'}(I(n(\alpha_{\sigma,M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma,M'}{}^{-1}(a'_k)))) & \text{if } n \in \Sigma_{CC} \text{ with } \nu(n) = n' \\ Top^{\mathcal{D}}_{comms'} & \text{otherwise} \end{cases}$$

for all process names $n'_{w',comms'} \in \Sigma'_{CC}$ and alphabet elements $a'_1, \ldots, a'_k \in \overline{s'_1}_{M'} \times \ldots \times \overline{s'_k}_{M'}$ where $w' = \langle s'_1, \ldots, s'_k \rangle$.

By the construction of $J'$ we know that $(M', J')$ is a CSP-CASL model, $J' \sqsubseteq I'$ and $J'|_\theta = I$. Finally, we have $(M'|_\sigma, J'|_\theta) = (M', J')|_\theta \in$ **Mod**$(SP)$. Therefore $(M', J') \in$ **Mod**$(SP$ **rename** $\theta)$. As $J' \sqsubseteq I'$ we know $(M', I') \in$ **Mod**$(RefCl(SP$ **rename** $\theta))$.

As $(M'|_\sigma, I) \in$ **Mod**$(SP)$ and $J'|_\theta = I$, we have $(M', J')|_\theta \in$ **Mod**$(SP)$. By definition of renaming we have $(M', J') \in$ **Mod**$(SP$ **rename** $\theta)$. As $J' \sqsubseteq I'$, we know $(M', I') \in$ **Mod**$(RefCl(SP$ **rename** $\theta))$.

**Proof of rule 4** Assume $SP_1 \rightsquigarrow SPI'$ **hide** $\theta_1$, $SP_2 \rightsquigarrow SPI'$ **hide** $\theta_2$, signature morphisms $\theta_1$ and $\theta_2$ are injective on *process names*, and $image(\nu_1) \cap image(\nu_2) = \emptyset$. We show **Mod**$(SPI') \subseteq$ **Mod**$(RefCl((SP_1$ **rename** $\theta_1)$ **and** $(SP_2$ **rename** $\theta_2)))$. Let $(M', I') \in$ **Mod**$(SPI')$. By definition of hiding we have $(M', I')|_{\theta_1} \in$ **Mod**$(SPI'$ **hide** $\theta_1)$ and $(M', I')|_{\theta_2} \in$ **Mod**$(SPI'$ **hide** $\theta_2)$. By assumption we know that $(M', I')|_{\theta_1} \in$ **Mod**$(RefCl(SP_1))$ and $(M', I')|_{\theta_2} \in$ **Mod**$(RefCl(SP_2))$. Therefore, we know there exists model $(M'|_{\sigma_1}, I_1) \in$ **Mod**$(SP_1)$ such that $I_1 \sqsubseteq I'|_{\theta_1}$ and $(M'|_{\sigma_2}, I_2) \in$ **Mod**$(SP_2)$ such that $I_2 \sqsubseteq I'|_{\theta_2}$. Let the type of $\theta_i$ be $\Sigma_{CCi} \to \Sigma_{CC}{}'$ for $i = 1, 2$.

We now construct $J'$ (relative to $M'$) similarly to the construction in the proof of the structured **rename** rule of Lemma 9.3, that is,

$$J'(n'(a'_1, \ldots, a'_k)) := \begin{cases} \alpha^{\mathcal{D}}_{\sigma_1,M'}(I_1(n_1(\alpha_{\sigma_1,M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma_1,M'}{}^{-1}(a'_k)))) & \text{if } n_1 \in \Sigma_{CC1} \text{ with } \nu_1(n_1) = n' \\ \alpha^{\mathcal{D}}_{\sigma_2,M'}(I_2(n_2(\alpha_{\sigma_2,M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma_2,M'}{}^{-1}(a'_k)))) & \text{if } n_2 \in \Sigma_{CC2} \text{ with } \nu_2(n_2) = n' \\ Top^{\mathcal{D}}_{comms'} & \text{otherwise} \end{cases}$$

for all process names $n'_{w',comms'} \in N'$ and alphabet elements $a'_1, \ldots, a'_k \in \overline{s'_1}_{M'} \times \ldots \times \overline{s'_k}_{M'}$ where $w' = \langle s'_1, \ldots, s'_k \rangle$.

$J'$ is well formed since $\theta_1$ and $\theta_2$ are injective on process names and their images do not overlap. Furthermore, by the construction of $J'$ we know that, $(M', J')$ is a CSP-CASL model, $J'|_{\theta_1} = I_1$, $J'|_{\theta_2} = I_2$, and $J' \sqsubseteq I'$.

As $(M'|_{\sigma_1}, I_1) \in \mathbf{Mod}(SP_1)$ and $J'|_{\theta_1} = I_1$, we have $(M', J')|_{\theta_1} \in \mathbf{Mod}(SP_1)$. As $(M'|_{\sigma_2}, I_2) \in \mathbf{Mod}(SP_2)$ and $J'|_{\theta_2} = I_2$, we also have $(M', J')|_{\theta_2} \in \mathbf{Mod}(SP_2)$. By definition of renaming, we have $(M', J') \in \mathbf{Mod}(SP_1 \textbf{ rename } \theta_1)$ and $(M', J') \in \mathbf{Mod}(SP_2 \textbf{ rename } \theta_2)$. Thus, $(M', J') \in \mathbf{Mod}((SP_1 \textbf{ rename } \theta_1) \textbf{ and } (SP_2 \textbf{ rename } \theta_2))$. Finally, as $J' \sqsubseteq I'$, we know $(M', I') \in \mathbf{Mod}(\mathit{RefCl}((SP_1 \textbf{ rename } \theta_1) \textbf{ and } (SP_2 \textbf{ rename } \theta_2)))$.

**Proof of rule 5** Assume $SP_1 \rightsquigarrow SPI$, $SP_2 \rightsquigarrow SPI$, ProcConst$(SP_1, SP_2)$, and either $SP_1$ is single-valued or $SP_2$ is single-valued. We show $\mathbf{Mod}(SPI) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$. Let $(M, I) \in \mathbf{Mod}(SPI)$. By assumption we know $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(SP_1))$ and $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(SP_2))$. Thus, we know there exists $(M, I_1) \in \mathbf{Mod}(SP_1)$ such that $I_1 \sqsubseteq I$ and there exists $(M, I_2) \in \mathbf{Mod}(SP_2)$ such that $I_2 \sqsubseteq I$. By process consistency we know there exists $(M, J) \in \mathbf{Mod}(SP_1) \cap \mathbf{Mod}(SP_2) = \mathbf{Mod}(SP_1 \textbf{ and } SP_2)$. Let w.l.o.g. $SP_1$ be single-valued. As $(M, J) \in \mathbf{Mod}(SP_1)$ and $(M, I_1) \in \mathbf{Mod}(SP_1)$ then $J = I_1$. As $(M, J) \in \mathbf{Mod}(SP_1 \textbf{ and } SP_2)$ and $J \sqsubseteq I$, we have $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(SP_1 \textbf{ and } SP_2))$. □

Provided that the structured specifications are restricted to certain (mostly) reasonable constructions principles, we obtain completeness as well.

**Lemma 9.9** The proof rules in Lemma 9.8 are complete relative to a complete calculus $\vdash$ and where all structured specifications satisfy the following restrictions:

- Each basic specification $(D, P)$ is either
  - refinement closed (i.e., $(D, P) \equiv \mathit{RefCl}(D, P)$); or
  - have all formulae preserved by CSP refinement, that is, for any models $(M, I)$ and $(M, J)$ of $(D, P)$ such that $I \sqsubseteq J$, it holds that $(M, I) \models \varphi$ iff $(M, J) \models \varphi$ for all $\varphi \in \mathbf{Ax}(D, P)$.
- All renamings are injective.
- Any specification union is either of the form
  - $(SP_1 \textbf{ rename } \theta_1) \textbf{ and } (SP_2 \textbf{ rename } \theta_2)$ where signature morphisms $\theta_1$ and $\theta_2$ are injective on *process names* and $\mathit{image}(\nu_1) \cap \mathit{image}(\nu_2) = \emptyset$; or
  - $SP_1 \textbf{ and } SP_2$ where ProcConst$(SP_1, SP_2)$ and either $SP_1$ is single-valued or $SP_2$ is.

*Proof.* We prove that if $SP \rightsquigarrow SPI$ then there exists a derivation tree concluding with $SP \rightsquigarrow SPI$. We do this by induction over the structure of $SP$. Note: In the following we use $SP$ and $SPI$ and the primed variants ($SP'$ and $SPI'$, respectively) according to which signature they have.

**Base Case: 1st Rule** Let $SP \equiv (D, P)$. We know by assumption that $\mathbf{Mod}(SPI) \subseteq \mathbf{Mod}(\mathit{RefCl}(D, P))$. We find a derivation tree concluding with $(D, P) \rightsquigarrow SPI$. It is enough to show $SPI \vdash \varphi$ for all $\varphi \in \mathbf{Ax}(D, P)$ as we can then use the first rule to create our full tree concluding in $(D, P) \rightsquigarrow SPI$. Furthermore, by the completeness of $\vdash$, it is enough to show $SPI \models \varphi$ for all $\varphi \in \mathbf{Ax}(D, P)$.

To this end, let $\varphi \in \mathbf{Ax}(D, P)$. Let $(M, I) \in \mathbf{Mod}(SPI)$. By assumption we know $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(D, P))$. By assumption we know either $(D, P)$ is refinement closed or CSP refinement preserves the axioms of $(D, P)$. If $(D, P)$ is refinement closed, then we know $(M, I) \in \mathbf{Mod}(D, P)$. Therefore $(M, I) \models \varphi$. We now consider the case of CSP refinement preserving the axioms of $(D, P)$. As $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(D, P))$, there exists $(M, J) \in \mathbf{Mod}(D, P)$ such that $J \sqsubseteq I$. We know $(M, J) \models \varphi$. As CSP refinement preserves $\varphi$ we know $(M, I) \models \varphi$.

**Induction Step: 2nd Rule** Let $SP \equiv SP'$ **hide** $\theta$. We know by assumption $\mathbf{Mod}(SPI) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP' \textbf{ hide } \theta))$. We must find a derivation tree concluding with $SP' \textbf{ hide } \theta \rightsquigarrow SPI$. It is enough to show $SP' \rightsquigarrow SPI'$ and $SPI'$ *ConsExt* $SPI$ thanks to rule 2 and the induction hypothesis for some CSP-CASL specification $SPI'$.

To this end, choose $SPI' = \mathit{RefCl}(SP')$. Note that $\mathit{RefCl}(SP')$ is a specification as refinement closure can be expressed as a structured specification, see Section 9.2. Then we have $SP' \rightsquigarrow SPI'$. We show $SPI'$ *ConsExt* $SPI$, that is, for all models $(M, I) \in \mathbf{Mod}(SPI)$ we must find a model $(M', I') \in \mathit{Mod}(SPI')$ such that $(M', I')|_\theta = (M, I)$. To this end, let $(M, I) \in \mathbf{Mod}(SPI)$. Then we know $(M, I) \in \mathbf{Mod}(\mathit{RefCl}(SP' \textbf{ hide } \theta))$, that is, there exists model $(M, J) \in \mathbf{Mod}(SP' \textbf{ hide } \theta)$ such that $J \sqsubseteq I$. By the definition of hiding we know there exists $(M', J') \in \mathbf{Mod}(SP')$ such that $(M', J')|_\theta = (M, J)$.

Let $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$. We now construct $I'$ (relative to $M'$) similarly to the construction of $J'$ in the proof of the structured **rename** rule of Lemma 9.3, that is,

$$I'(n'(a'_1, \ldots, a'_k)) := \begin{cases} \alpha^{\mathcal{D}}_{\sigma,M'}(I(n(\alpha_{\sigma,M'}{}^{-1}(a'_1), \ldots, \alpha_{\sigma,M'}{}^{-1}(a'_k)))) & \text{if } n \in \Sigma_{CC} \text{ with } \nu(n) = n' \\ J'(n'(a'_1, \ldots, a'_k)) & \text{otherwise} \end{cases}$$

for all process names $n'_{w',comms'} \in \Sigma'_{CC}$ and alphabet elements $a'_1, \ldots, a'_k \in \overline{s'_1}_{M'} \times \ldots \times \overline{s'_k}_{M'}$ where $w' = \langle s'_1, \ldots, s'_k \rangle$.

$I'$ is well defined since $\nu$ is assumed to be injective. By the construction of $I'$ we know that, $(M', I')$ is a CSP-CASL model, $I'|_\theta = I$,

We now show $J' \sqsubseteq I'$. Let $n'$ be a process name and $a'_1, \ldots, a'_k$ be appropriate alphabet parameters. We show $J'(n'(a'_1, \ldots, a'_k)) \sqsubseteq I'(n'(a'_1, \ldots, a'_k))$. We make a case distinction on whether there exists $n \in \Sigma_{CC}$ such that $\nu(n) = n'$.

**Case** $\nu(n) = n'$ Let $\alpha_{\sigma,M'}{}^{-1}(a'_i) = a_i$ for $1 \leqslant i \leqslant k$. We show $J'(n'(a'_1, \ldots, a'_k)) \sqsubseteq \alpha^{\mathcal{D}}_{\sigma,M'}(I(n(a_1, \ldots, a_k)))$. As $J \sqsubseteq I$ and $J = J'|_\theta$ we know $J'|_\theta(n(a_1, \ldots, a_k)) \sqsubseteq I(n(a_1, \ldots, a_k))$, that is, $\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(J'(n'(a'_1, \ldots, a'_k))) \sqsubseteq I(n(a_1, \ldots, a_k))$. As $\alpha^{\mathcal{D}}_{\sigma,M'}$ preserves refinement (Corollary 7.15), we obtain $\alpha^{\mathcal{D}}_{\sigma,M'}(\hat{\alpha}^{\mathcal{D}}_{\sigma,M'}(J'(n'(a'_1, \ldots, a'_k)))) \sqsubseteq$

$\alpha^{\mathcal{D}}_{\sigma,M'}(I(n(a_1,\ldots,a_k)))$. As $\alpha^{\mathcal{D}}_{\sigma,M'} \circ \hat{\alpha}^{\mathcal{D}}_{\sigma,M'}$ is identity (thanks to the controlled traces and Lemma 7.18), we arrive at $J'(n'(a'_1,\ldots,a'_k)) \sqsubseteq \alpha^{\mathcal{D}}_{\sigma,M'}(I(n(a_1,\ldots,a_k)))$.

**Case otherwise** As $I'(n'(a'_1,\ldots,a'_k)) = J'(n'(a'_1,\ldots,a'_k))$, by reflexivity of CSP refinement, we have $J'(n'(a'_1,\ldots,a'_k)) \sqsubseteq I'(n'(a'_1,\ldots,a'_k))$.

As $(M',J') \in \mathbf{Mod}(SP')$ and $J' \sqsubseteq I'$, we know $(M',I') \in \mathbf{Mod}(\mathit{RefCl}(SP')) = \mathbf{Mod}(SPI')$.

**Induction Step: 3rd Rule** Let $SP \equiv SP_1$ **rename** $\theta$. We know by assumption that $\mathbf{Mod}(SPI') \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_1$ **rename** $\theta))$. We must find a derivation tree that concludes with the statement $SP_1$ **rename** $\theta \rightsquigarrow SPI'$. It is enough to show $SP_1 \rightsquigarrow SPI'$ **hide** $\theta$ thanks to rule 3 and the induction hypothesis, i.e., $\mathbf{Mod}(SPI'$ **hide** $\theta) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_1))$.

To this end, let $(M,I) \in \mathbf{Mod}(SPI'$ **hide** $\theta)$, then there exists $(M',I') \in \mathbf{Mod}(SPI')$ such that $(M',I')|_\theta = (M,I)$. By assumption we know $(M',I') \in \mathbf{Mod}(\mathit{RefCl}(SP_1$ **rename** $\theta))$, i.e. there exists $J'$ such that $(M',J') \in \mathbf{Mod}(SP_1$ **rename** $\theta)$ and $J' \sqsubseteq I'$. By definition of renaming we know $(M',J')|_\theta = (M,J|_\theta) \in \mathbf{Mod}(SP_1)$. As $J' \sqsubseteq I'$ we know $J'|_\theta \sqsubseteq I'|_\theta = I'$, thus $(M,I) \in \mathbf{Mod}(\mathit{RefCl}(SP_1))$.

**Induction Step: 4th Rule** Let $SP \equiv (SP_1$ **rename** $\theta_1)$ **and** $(SP_2$ **rename** $\theta_2)$. We know by assumption $\mathbf{Mod}(SPI') \subseteq \mathbf{Mod}(\mathit{RefCl}((SP_1$ **rename** $\theta_1)$ **and** $(SP_2$ **rename** $\theta_2)))$. We must find a derivation tree concluding with $(SP_1$ **rename** $\theta_1)$ **and** $(SP_2$ **rename** $\theta_2) \rightsquigarrow SPI'$. Thanks to rule 4, the induction hypothesis and our assumptions it is enough to show $SP_1 \rightsquigarrow SPI'$ **hide** $\theta_1$ and $SP_2 \rightsquigarrow SPI'$ **hide** $\theta_2$, where $\theta_1 = (\sigma_1,\nu_1)$ and $\theta_2 = (\sigma_2,\nu_2)$.

We show $SP_1 \rightsquigarrow SPI'$ **hide** $\theta_1$, that is, $\mathbf{Mod}(SPI'$ **hide** $\theta_1) \subseteq \mathbf{Mod}(SP_1)$. Let $(M,I) \in \mathbf{Mod}(SPI'$ **hide** $\theta_1)$. We know there exists $(M',I') \in \mathbf{Mod}(SPI')$ such that $(M',I')|_{\theta_1} = (M,I)$. By assumption we know $(M',I') \in \mathbf{Mod}(\mathit{RefCl}((SP_1$ **rename** $\theta_1)$ **and** $(SP_2$ **rename** $\theta_2)))$. By definition of *RefCl* we know there exists $(M',J') \in \mathbf{Mod}((SP_1$ **rename** $\theta_1)$ **and** $(SP_2$ **rename** $\theta_2))$ such that $J' \sqsubseteq I'$. Therefore $(M',J') \in \mathbf{Mod}(SP_1$ **rename** $\theta_1)$, thus, $(M',J')|_{\theta_1} = (M,J'|_{\theta_1}) \in \mathbf{Mod}(SP_1)$. As $J \sqsubseteq I$ we know $J'|_{\theta_1} \sqsubseteq I'|_{\theta_1} = I$, thus $(M,I) \in \mathbf{Mod}(\mathit{RefCl}(SP_1))$. Showing $SP_2 \rightsquigarrow SPI'$ **hide** $\theta_2$ is analogous.

**Induction Step: 5th Rule** Let $SP \equiv SP_1$ **and** $SP_2$. We know by assumption that $\mathbf{Mod}(SPI) \subseteq \mathbf{Mod}(\mathit{RefCl}(SP_1$ **and** $SP_2))$. We must find a derivation tree concluding with $SP_1$ **and** $SP_2 \rightsquigarrow SPI$. Thanks to rule 5, the induction hypothesis and our assumptions it is enough to show $SP_1 \rightsquigarrow SPI$ and $SP_2 \rightsquigarrow SPI$.

We show $SP_1 \rightsquigarrow SPI$. Let $(M,I) \in \mathbf{Mod}(SPI)$, then by assumption we know $(M,I) \in \mathbf{Mod}(\mathit{RefCl}(SP_1$ **and** $SP_2))$. There exists $(M,J) in \mathbf{Mod}(SP_1$ **and** $SP_2)$ such that $J \sqsubseteq I$. As $(M,I) \in \mathbf{Mod}(SP_1)$ and $J \sqsubseteq I$, we have $(M,J) \in \mathbf{Mod}(\mathit{RefCl}(SP_1))$. Showing $SP_2 \rightsquigarrow SPI$ is analogous. □

We have shown that a complete refinement calculus is possible with our notion of CSP-CASL refinement, provided Structured CSP-CASL specifications are restricted to certain forms.

The less reasonable restrictions are necessary to deal with the structured **and** operator. This result indicates that the structured **and** operator from algebraic specification might not be the correct operator to join specifications when dealing with process algebras.

In this chapter we have introduced a new refinement notion for CSP-CASL which reconciles the algebraic specification world of CASL and the process algebra world of CSP. The refinement notion is based upon model class inclusion and takes CSP refinement closure into account. We have shown that this notion of refinement is well suited for system development and behaves well with respect to the structuring mechanisms introduced in Section 4.7. We have shown several calculi that allow compositional reasoning over structured CSP-CASL specifications. We have also lifted a proof technique for establishing deadlock freedom of networks [RSR04] from CSP to CSP-CASL. This proof technique makes it possible to prove deadlock freedom of networks in an elegant and compositional manner. We surmise that it is possible to lift other properties and techniques from CSP to CSP-CASL in a similar way. Finally, we provided a complete refinement calculus for CSP-CASL refinement.

*Chapter 10*

# Application

## Contents



CSP-CASL aims to support the modelling and verification of critical systems such as EP2 (see Section 1.1 for details). Here we demonstrate, closely following [OMR12], our results on a simpler system that nonetheless exhibits the same typical challenges as EP2.

The system we use is that of an online shopping system, which allows a customer to buy goods. The online shopping system exhibits the same dialogue patterns and architectural structure as EP2. However, for simplicity, it features fewer message types. The online shopping system is composed of four distinct components: a customer, coordinator, warehouse and payment system. The coordinator takes a central role much like the terminal in EP2: all other components only communicate directly with the coordinator.

In this chapter we discuss how to model and verify this online shopping system. To this end, we model the system in CSP-CASL where we make use of loose processes and structuring in a natural way. We then prove deadlock freedom of the system using our deadlock analysis method for networks and our compositional deadlock freedom calculus from Section 9.4. This example shows that Structured CSP-CASL is useful in practice for the modelling and verification of large critical systems.

As briefly discussed in Section 8.4, our signature morphisms do not allow for the mapping of unparametrised process names to parametrised process names instantiated with ground terms. This can be simulated by introducing auxiliary process names. For the sake of readability we present the following example where signature morphisms map unparametrised process names to parametrised process names instantiated with ground terms.

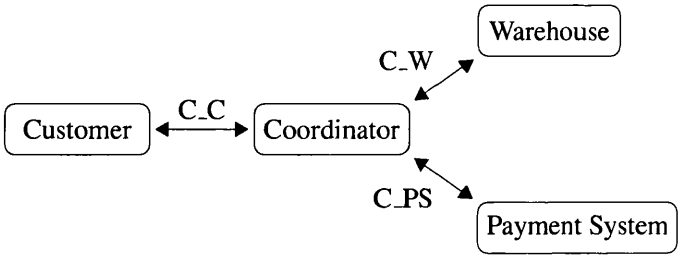



Figure 10.1: Architecture of the online shop example.

## 10.1 Specifying an Online Shop

The online shop is a typical distributed system. It comprises of several components: a customer, a warehouse, a payment system, and a coordinator. We describe each component of the system at various levels of abstraction, mirroring the abstraction levels used in EP2. We consider three levels of abstraction, each one increasing in detail. These are:

**Architectural Level (Arch)** This level describes the static system architecture and does not prescribe any other details of the components or their interactions.

**Abstract Component Level (ACL)** At this level we describe the protocols between each of the components. That is, we describe the message types and sequences of messages that may be sent to and from each component.

**Concrete Component Level (CCL)** This level adds state and computational requirements to each of the components.

### 10.1.1 Natural Language System Overview

The online shop consists of four components: a customer, a warehouse, a payment system, and a coordinator. The communication structure is pointwise only: the coordinator communicates with the three other components in a star like network. The customer, warehouse and payment system only communicate with the coordinator, see Figure 10.1.

The customer may ask the coordinator to perform actions such as: to login, to add an item to the basket, to remove an item from the basket, to checkout, etc. The coordinator then responds to the customer with an appropriate response message. All communication (on a channel) follows this pattern of a request message followed by a response message. The coordinator may ask the warehouse to reserve an item, to release an item that has previously been reserved, and to dispatch the reserved items. The payment system allows the coordinator to take payments for goods. Before the customer performs any of the above actions, they are required to login. A customer may also choose to logout after which they must login again in order to use the service.

**spec** GENERIC_SHOP [*RefCl*(ARCH_CUSTOMER)] [*RefCl*(ARCH_WAREHOUSE)]
[*RefCl*(ARCH_PAYMENTSYSTEM)] [*RefCl*(ARCH_COORDINATOR)] =
**process** *System* : *C_C*, *C_W*, *C_PS*;
*System* = *Coordinator* [ *C_C*, *C_W*, *C_PS* || *C_C*, *C_W*, *C_PS* ]
(*Customer* [ *C_C* || *C_W*, *C_PS* ]
(*Warehouse* [ *C_W* || *C_PS* ] *PaymentSystem*))
**end**

Figure 10.2: Generic specification of the online shop example.

### 10.1.2 CSP-CASL Specification

We now formalise the natural language specification in CSP-CASL where we make use of loose processes and structuring. This demonstrates the ability of CSP-CASL to elegantly model such systems.

We follow the abstraction levels described above and create three shop specifications. One for each level of abstraction. Below, we discuss in detail the specification of the architectural and abstract component levels. Appendix D provides full specifications of these levels.

The architecture of the shop does not change and only components are developed from one level to another. We wish to mirror this formally. To this end, we use generic and instantiated specifications. These allow us to develop the components and instantiate a generic shop specification with the developed components. The generic shop will establish the architecture of the system with respect to the instantiated parameters.

The generic shop specification shown in Figure 10.2 (full specifications are included in Appendix D) describes the network layout, which remains unchanged at all levels of abstraction. The generic shop takes as formal parameters the refinement closure of the architectural components. Each component declares a channel and a 'main' process which communicates over that channel. The generic specification then uses these channels and main processes to form the overall *System* process as a network of the components' main processes.

We take refinement closures as formal parameters because the formal parameters represent all possible models. Thus, we use the refinement closure to capture all possible CSP refinements. Without this, instantiations would not be well defined as we would not have model class inclusion. We are able use refinement closures within specifications, such as formal parameters, as there is a syntactic characterisation of refinement closure expressible in CSP-CASL, see Section 9.2 for details.

Once we have specified the components at the other two levels of abstraction, we can form all three shop specifications. We do this by instantiating the generic shop specification with the appropriate components (see Figure 10.3). We are able to use instantiated CSP-CASL specifications as we have proven that our CSP-CASL institutions exhibit a suitable amalgamation property (see Theorem 8.18), which in turn defines the model class of such instantiated specifications. In order for these instantiations to be well formed, the model classes of the actual parameters must be included in the model classes of the formal parameters. Thanks to the refinement closures of the formal parameters (and the definition of CSP-CASL refinement), this

**spec** ARCH_SHOP = GENERIC_SHOP [ARCH_CUSTOMER] [ARCH_WAREHOUSE]
[ARCH_PAYMENTSYSTEM] [ARCH_COORDINATOR]
**end**

**spec** ACL_SHOP = GENERIC_SHOP [ACL_CUSTOMER] [ACL_WAREHOUSE]
[ACL_PAYMENTSYSTEM] [ACL_COORDINATOR]
**end**

**spec** CCL_SHOP = GENERIC_SHOP [CCL_CUSTOMER] [CCL_WAREHOUSE]
[CCL_PAYMENTSYSTEM] [CCL_COORDINATOR]
**end**

Figure 10.3: Instantiations of the generic online shop specification.

is equivalent to showing that the architectural components refine to each of the instantiated components.

We now focus on specifying the individual components in CSP-CASL. Figure 10.4 shows the development graph after HETS has successfully parsed and analysed the online shop specifications. The black arrows show the import structure of the specifications, while the red arrows show open proof obligations resulting from the instantiations.

We have decided that the logging in and logging out behaviour is important enough to be captured already at the architectural level, although we do this in a loose way. Figure 10.5 shows the architectural customer specification. We specify that the main *Customer* process should only communicate with the coordinator over the channel *C_C*. The channel is declared to communicate values of the sort *D_C* which is loosely specified and represents all data that can be communicated between the coordinator and the customer. We specify that there are several sub-processes which are intended to be specialised during development. All the sub-process are loosely specified except for the body process which specifies how the sub-processes are connected, and the main and logout processes which dictate when login and logout request messages are sent. We leave loose the behaviour that should happen upon successful and failed logins and logouts. The body process uses internal non-deterministic choice to join the sub-processes as the customer will make the choice of how to use the shop.

The coordinator shown in Figure 10.6 is similar. However, instead of sending login and logout requests, it instead sends login and logout responses. We also join the sub-process using the external choice operator as the coordinator should offer all services to the customer. The coordinator processes are also declared to communicate over the three channels connecting the coordinator to the customer, warehouse and payment system. The other two components are similar and declare sub-processes which are left open (see Appendix D.2).

The CSP-CASL specifications of the components at the abstract component level add more detail to that of their counterparts at the architectural level. Appendix D.3 presents the specifications of the components at the abstract component level.

Figure 10.7 shows part of the CASL specification of the data used as communications be-

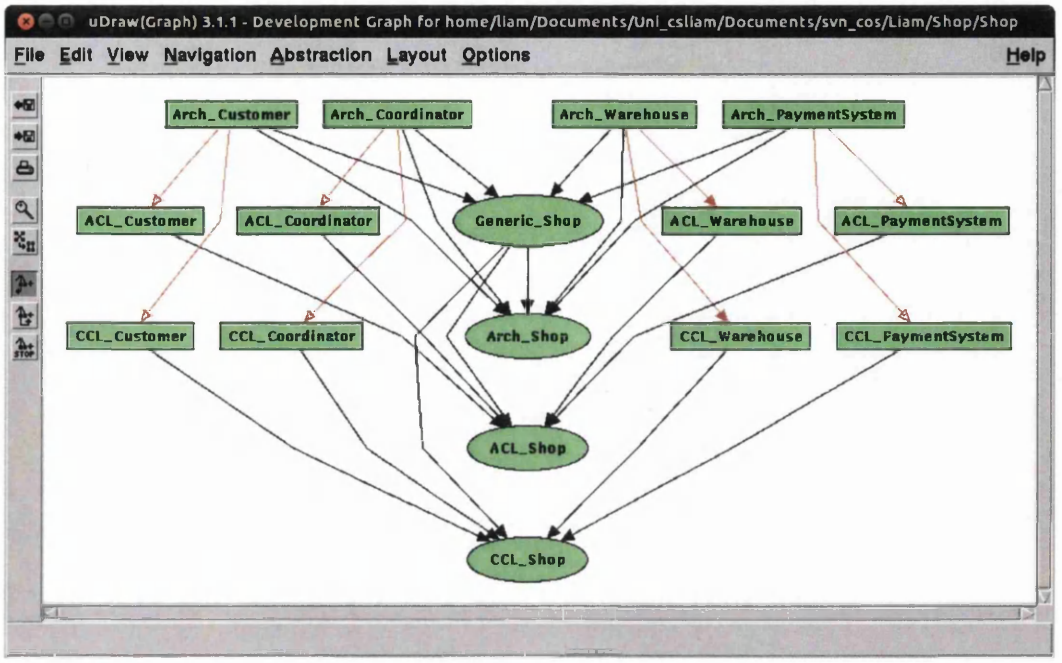



Figure 10.4: Screen-shot showing HETS' development graph of the online shop specifications after instantiation.

tween the customer and coordinator at the abstract component level. Here, we have refined (in an informal sense) the sort *D_C* and added many sub-sorts using the CASL free type construct. We add a new sub-sort for each type of communication message, for example, a view basket request message (*ViewBasketReq*) and an associated view basket response message (*ViewBasketRes*). The free type ensures there are no other message types available and also that the message types do not overlap.

Figure 10.8 shows part of the specification of the abstract component level customer. Here, each of the sub-processes, which were left loose at the architectural level, are 'filled in' and have an associated process equation. For instance, the process *Customer_ViewBasket* is now specified to have the behaviour of non-deterministically sending a defined view basket request message to the coordinator, followed by receiving a corresponding response message. Following this interaction the sub-process behaves like the customer's body process. Note that we do not specify precise messages, instead we only specify message types.

The coordinator at the abstract component level is specified in a similar manner, but instead of sending requests and receiving responses, it waits for requests and sends responses of the correct sort. The other components are specified similarly.

**spec** ARCH_CUSTOMER =
  **data** ARCH_CUSTOMER_DATA
  **channel** $C\_C : D\_C$
  **process** $Customer : C\_C$;
    $Customer\_SuccessfulLogin : C\_C$;
    $Customer\_FailedLogin : C\_C$;
    $Customer\_Body : C\_C$;
    $Customer\_ViewCatalogue : C\_C$;
    $Customer\_ViewBasket : C\_C$;
    $Customer\_AddItem : C\_C$;
    $Customer\_RemoveItem : C\_C$;
    $Customer\_Checkout : C\_C$;
    $Customer\_Logout : C\_C$;
    $Customer\_SuccessfulLogout : C\_C$;
    $Customer\_FailedLogout : C\_C$;
    $Customer = C\_C\,!\,x :: LoginReq \rightarrow$
      $(Customer\_SuccessfulLogin\ ;\ Customer\_Body$
      $\Box\ Customer\_FailedLogin\ ;\ Customer)$;
    $Customer\_Logout = C\_C\,!\,x :: LogoutReq \rightarrow$
      $(Customer\_SuccessfulLogout\ ;\ Customer$
      $\Box\ Customer\_FailedLogout\ ;\ Customer\_Body)$;
    $Customer\_Body = Customer\_ViewCatalogue \sqcap Customer\_ViewBasket$
      $\sqcap\ Customer\_AddItem \sqcap Customer\_RemoveItem$
      $\sqcap\ Customer\_Checkout \sqcap Customer\_Logout$
**end**

Figure 10.5: The architectural customer specification.

## 10.2 Establishing Well Formed Instantiations

Refinements obligations originate from two sources: instantiations and development steps. Here, we focus on the former. For instantiations to be well formed we must show that the models of the actual parameters are contained within the models of the formal parameters, see Section 4.7.2. The instantiations we create in Figure 10.3 generate twelve individual refinement obligations. Four are proven trivially, while the other eight require a little effort. Within the rest of this chapter we use $C$ to denote customer, $Co$ to denote coordinator, $W$ to denote warehouse and $PS$ to denote payment system. We also drop the communications sets within the network construction

The first four proof obligations come from the instantiation of the generic shop with the architectural components themselves. Thus, in order for the architectural instantiation (i.e., the CSP-CASL specification ARCH_SHOP in Figure 10.3) to be well formed we must show

$$\mathbf{Mod}(\text{ARCH\_COMP}) \subseteq \mathbf{Mod}(RefCl(\text{ARCH\_COMP}))$$

**spec** ARCH_COORDINATOR =
...
**process** *Coordinator* : *C_C*, *C_W*, *C_PS*;
*Coordinator_SuccessfulLogin* : *C_C*;
*Coordinator_FailedLogin* : *C_C*;
*Coordinator_Body* : *C_C*, *C_W*, *C_PS*;
...
*Coordinator* = *C_C* ? *x* :: *LoginReq* →
(*Coordinator_SuccessfulLogin* ; *Coordinator_Body*
⊓ *Coordinator_FailedLogin* ; *Coordinator*);
*Coordinator_Logout* = *C_C* ? *x* :: *LogoutReq* →
(*Coordinator_SuccessfulLogout* ; *Coordinator*
⊓ *Coordinator_FailedLogout* ; *Coordinator_Body*);
*Coordinator_Body* = *Coordinator_ViewCatalogue* □ *Coordinator_ViewBasket*
□ *Coordinator_AddItem* □ *Coordinator_RemoveItem*
□ *Coordinator_Checkout* □ *Coordinator_Logout*
...

Figure 10.6: The architectural coordinator specification.

**spec** ACL_COMM_COORDINATOR_CUSTOMER_DATA = ACL_COMMON_DATA
**then sorts** *LoginReq*, *SuccessfulLoginRes*, *FailedLoginRes*,...
*ViewBasketReq*, *ViewBasketRes*, *AddItemReq*, ...
**free type** *D_C* ::= *sort LoginReq* ... | *sort ViewBasketReq*
...
| *sort SuccessfulLoginRes* | *sort FailedLoginRes* ...
| *sort ViewBasketRes* ...

Figure 10.7: CASL specification of communications between the customer and coordinator at the abstract component level.

**spec** ACL_CUSTOMER =
**data** ACL_CUSTOMER_DATA
**channel** $C\_C : D\_C$
**process** $Customer : C\_C;\ldots$
$Customer\_Body : C\_C;$
$Customer\_ViewBasket : C\_C;\ldots$
$Customer = C\_C\,!\,x :: LoginReq \rightarrow$
$(Customer\_SuccessfulLogin\ ;\ Customer\_Body$
$\Box\ Customer\_FailedLogin\ ;\ Customer);\ldots$
$Customer\_ViewBasket = C\_C\,!\,x :: ViewBasketReq\ def \rightarrow$
$C\_C\ ?\ y :: ViewBasketRes \rightarrow Customer\_Body;\ldots$
$Customer\_Body = Customer\_ViewCatalogue \sqcap Customer\_ViewBasket\ldots$

Figure 10.8: Excerpt of the abstract component level customer specification.

for each component. Thanks to the refinement closure included in each formal parameter of the generic shop specification, this goal is actually the CSP-CASL refinement ARCH_COMP $\rightsquigarrow$ ARCH_COMP. As CSP-CASL refinement is reflexive (Lemma 9.1), we know this holds.

By the same logic, for the ACL and CCL instantiations to be well formed, we must show the following two CSP-CASL refinements:

- ARCH_COMP $\rightsquigarrow^{\theta_{Arch2ACL\_Comp}}$ ACL_COMP, and
- ARCH_COMP $\rightsquigarrow^{\theta_{Arch2CCL\_Comp}}$ CCL_COMP.

for each component, where the signature morphisms are the embeddings between the signatures of the components of the respective abstraction levels. As the components involved in these refinements are structured specifications which only utilise the structured union and renaming operations, and not the hiding operation, we can flatten them into basic specifications (see Section 4.7.1). Refinements between such basic specifications can be proven with CSP-CASL-Prover as discussed in Section 9.2.

Such refinements hold as the architectural level components specify a set of process names (the main process, the body process and sub-processes) and only give equations to the main and body process names, that is, they only bind the main and body process names' behaviour to CSP processes. All sub-processes are left loosely specified. The abstract component level components, on the other hand, specify the same set of process names and additionally define the behaviour for most of the sub-processes. The equations of the main and body processes remain the same as they were at the architectural level.

As only the sub-processes are additionally constrained, and as these are loosely specified at the architectural level, the models of the components at the abstract component level are included in the models of the (CSP) refinement closures of the architectural components.

## 10.3 Verification of Deadlock Freedom

Trying to prove deadlock freedom of the system at the ACL level in a naive way is infeasible. The traditional method would involve blowing up the network to an equivalent totally sequential process and then proving deadlock freedom on this sequential process. Unfortunately this sequential process grows exponentially with respect to the number of communications of each component.

We illustrate how to prove deadlock freedom using the technique presented in Section 9.4. We discuss the core part of the proof, and explain how to scale it up for the whole system. The proof rule from Lemma 9.6 reduces the network of processes step by step. We start at the point where the network has been reduced to two processes only:

**spec** REDUCED_ARCH_SHOP [*RefCl*(ARCH_C)] [*RefCl*(ARCH_CO)] =
  **process** *System'* : *C_C* ;
    *System'* = *Coordinator* ||[ *C_C* || *C_C* ]|| *Customer*
**end**

The specification REDUCED_ARCH_SHOP instantiated with ACL components is semantically equivalent to the following specification (without parametrisation):

REDUCED_ACL_SHOP =
(( (*RefCl*(ARCH_C) **rename** $\theta_1$) **and**
   (*RefCl*(ARCH_CO) **rename** $\theta_2$) **and** BODY
) **rename** $\theta_3$) **and** (ACL_C **rename** $\theta_4$) **and** (ACL_CO **rename** $\theta_5$)

Here, all signature morphisms involved are embeddings and the specification BODY is a basic specification with the signature equal to the union of the signatures of the ACL customer and coordinator along with the new process name $System'$, and where the only axiom is that of

*System'* = *Coordinator* ||[ *C_C* || *C_C* ]|| *Customer* .

Our aim is to prove that the process term bound to $System'$ is deadlock free within the specification REDUCED_ACL_SHOP. To this end, we apply Lemma 9.6 and obtain:

$Network(\{Customer, Coordinator\})$ $isDFin$ REDUCED_ACL_SHOP
if (a) $C$ $isDFin$ REDUCED_ACL_SHOP and
   (b) $Co :: C\_C$ $ResToLive^{\checkmark}$ $C :: C\_C$ *on* $C\_C$ *in* REDUCED_ACL_SHOP

To discharge obligation (a), we apply the **and** rule from Lemma 9.5 several times and reduce it to ($C$ $isDFin$ ACL_C **rename** $\theta_4$). Applying the renaming rule (also from Lemma 9.5) results in ($C$ $isDFin$ ACL_C). As ACL_C is a basic specification and the customer process does not involve any parallel operator we can easily discharge this obligation with CSP-CASL-Prover.

Concerning obligation (b), we apply the **and** rule from Lemma 9.7 several times and reduce it to:

$Co :: C\_C$ $ResToLive^{\checkmark}$ $C :: C\_C$ *on* $C\_C$ *in*
  ((ACL_C **rename** $\theta_4$) **and** (ACL_CO **rename** $\theta_5$)) .

As ACL_C and ACL_Co are essentially basic CSP-CASL specifications we can discharge the proof obligation by applying the flattening operation and then using CSP-CASL-Prover. This obligation holds because the coordinator allows the customer to choose the initial action (a request message) and then provides a response message to the customer for this particular type of request (possibly after further communications with other components).

The full proof of deadlock freedom has the same structure. Lemma 9.6 reduces

$$Network(\{Customer, Coordinator, PaymentSystem, Warehouse\})$$

down to $Network(\{Customer\})$ by removing first $Warehouse$, then $PaymentSystem$, and – as shown above – $Customer$ from the network. The resulting obligations can then be reduced to a format where they can be discharged with CSP-CASL-Prover.

In this chapter, we have demonstrated how to model a complex system using Structured CSP-CASL. We have shown that parametrised specifications can be utilised to provide an elegant formalisation. We have demonstrated the use of compositional proof calculi by proving deadlock freedom of the abstract component level of our online shopping system. This has illustrated the importance of compositional reasoning when working with processes and data within complex systems.

*Chapter 11*

# Implementation and Tool Support

**Contents**



In this chapter we discuss the available tool support for CSP-CASL, namely HETS (Heterogeneous Tool Set) [MML07]. HETS is a proof management tool centred around CASL. It supports parsing and static analysis of specifications written in CASL and related languages. It also has the ability to pretty-print specifications in various forms including support for outputting LaTeX code. Proof obligations in HETS may be discharged by utilising several external theorem provers with which HETS interfaces.

Existing work has been carried out which enables HETS to support parsing and static analysis of original CSP-CASL specifications [Gim08]. Here, we discuss the extent of the support for CSP-CASL within HETS and our additions to support Structured CSP-CASL.

## 11.1 HETS and Existing Support for CSP-CASL

HETS is written in the functional programming language Haskell [Jon03, HHJW07]. Haskell is a general purpose programming language based on the lambda calculus. It features lazy evaluation, higher order functions, polymorphism and type classes. HETS makes extended use of type classes to provide a framework which captures institutions and various other concepts such institutions representations.

Gimblett [Gim08] previously implemented support within HETS for CSP-CASL. This allowed HETS to parse, statically analyse and pretty-print CSP-CASL specifications. Structured CASL specifications could be used within the data part of CSP-CASL specifications, but no structuring was available within the process part. The original restriction on signatures, namely,

local top sorts (see Section 4.4), was implemented as a static analysis check that was carried out on all CSP-CASL specifications after parsing had taken place. Some initial efforts had been made to support multiple process names within CSP-CASL specifications. However, only basic functionality was available, and features such as signature morphisms were beyond the scope of the project.

## 11.2 Extending HETS for Structured CSP-CASL

Here, we discuss how we extend the existing support to cover the new features of Structured CSP-CASL. These include signature morphisms and multiple process names.

HETS utilises type classes to capture various notions such as signatures, sentences, parsers, and static analysers. To implement a new logic (or institution) in HETS, one provides new types and functions for the new logic and instantiates the various type classes in appropriate ways. The high level machinery of HETS then uses these instantiated type classes to support the new logic. You can choose to implement a minimum set of classes, which allows for parsing, basic static analysis, and pretty printing of specifications; or to implement additional classes which allow features such as signature morphisms and full structured specifications.

As an example of such a type class, consider the following Haskell code.

```
class (Language lid, Category sign morphism, Ord sentence,
       ...
    => Sentences lid sentence sign morphism symbol
      where
      -- | Sentence translation along a signature morphism
      map_sen :: lid -> morphism -> sentence -> Result sentence
      ...
      -- | Signature printing
      print_sign :: lid -> sign -> Doc
      ...
```

This is the HETS type class for sentences. This code defines the class named `Sentences` which takes as parameters five types: the logic identifier (`lid`), a type for sentences (`sentence`), a type for signatures `sign`, a type for signature morphisms `morphism` and finally, a type for symbols (`symbol`). Symbols are a mechanisms which allows the specifier to write symbol maps in place of full signature morphisms (see [Mos00]). The code before the => symbol are various restrictions on the instantiations of the types, for instance, it is required that the language identifier (`lid`) is already an instantiation of the `Language` type class that HETS provides. There are also various functions which must be provided. The function `map_sen` (short for map sentence) takes the logic identifier, a signature morphism and a sentence, and maps the sentence across the signature morphism. This captures part of the **sen** functor from institutions (see Chapter 4). The type `Result sentence` is a monadic type that captures a sentence which has been produced, but where an error may have occurred in its production. This encodes partial functions in Haskell, however the type contains additionally diagnosis information when errors occur. Such errors usually originate from specification errors. Diagnostic information is provided to help the user track down the mistake. The function `print_sign` pretty-prints

a signature by transforming it into a `Doc`, which is a type to represent plain text in an efficient manner. In order to implement sentences for a new logic, it is necessary to provide new types and functions and to instantiate this type class. Gimblett08 [Gim08] already instantiated this type classes (along with many more), but several functions such as `map_sen` were stubs as CSP-CASL signatures morphisms were not originally considered.

In order to extend the implementation to cover the new CSP-CASL, as presented in this thesis, it was necessary to adapt several types and provide real implementation in the stub functions of the instantiated type classes. The main types that changed were types capturing CSP-CASL signatures and signature morphisms.

We now briefly discuss the types used to capture CSP-CASL signatures and signature morphisms. The type that captures CASL signatures is parametrised with 'holes' for extensions. This was done with foresight to allow CASL extensions to be easily implemented. One simply fills in the holes with the extra information needed for the CASL extension. Here, we define a new type `CspSign` which is the CSP part of CSP-CASL signature. This is then inserted in the hole in the type of CASL signatures to form the type capturing CSP-CASL signatures.

```
type ChanNameMap = MapSet.MapSet CHANNEL_NAME SORT
type ProcNameMap = MapSet.MapSet PROCESS_NAME ProcProfile
data CspSign = CspSign
    { chans :: ChanNameMap
    , procSet :: ProcNameMap
    } deriving (Eq, Ord, Show)

type CspCASLSign = Sign CspSen CspSign
```

Our CSP signature extension is a record with two components, namely a channel map named `chans` and a process map named `procSet`. The type `procSet` is a mapping of process names to sets of process profiles. This allows us to bind a set of profiles (i.e., the type information of the process parameters and communication sets) to a name (i.e., the name of a process). For example, to capture three process names $P_{w_1,comms_1}$, $P_{w_2,comms_2}$, $Q_{w_1,comms_1}$, the map would be

$$\begin{aligned} P &\mapsto \{(w_1, comms_1), (w_2, comms_2)\} \\ Q &\mapsto \{(w_1, comms_1)\} \end{aligned}$$

That is, there are two profiles available for the name $P$ and one available for the name $Q$. This type was modified from the original implementation which mapped process names to profiles, that is, it assigned a single profile to each name and thus did not allow for overloaded process names. Channels work in a similar way.

The `CspSign` record type allows us to capture process names and channel names along with all the type information associated with them. We then use our record type as a parameter to the CASL signature type (`Sign`). The CASL signature type already stores sort symbols, function symbols, predicate symbols and the sub-sort relation. We augment this type with process and channel names to form the new type capturing CSP-CASL signatures (`CspCASLSign`).

A similar setup is required for signature morphisms. We define a CSP extension which contains channel name mappings and process name mappings between the source and target signatures. We use this extension with the CASL signature morphism type to form the type of CSP-CASL signature morphisms.

```
type ChanMap = Map.Map (CHANNEL_NAME, SORT) CHANNEL_NAME
type ProcessMap = Map.Map (PROCESS_NAME, ProcProfile) PROCESS_NAME
data CspAddMorphism = CspAddMorphism
    { channelMap :: ChanMap
    , processMap :: ProcessMap
    } deriving (Eq, Ord, Show)

type CspCASLMorphism =
    CASL_Morphism.Morphism CspSen CspSign CspAddMorphism
```

The type `ProcessMap` is a map from process names (with profiles) to new process names. Process profiles capture both the parameter type of process names and their communication sets. In accordance with the definition of CSP-CASL signature morphisms from Section 8.1.1, the target profile of a process name in a signature morphism is uniquely determined from the data part: there is no choice, both the list of parameter sorts and the communication set must be mapped in accordance with the CASL sort map. This is the reason why we map to process names only and not to pairs of process names and profiles. We choose to record only what is necessary and compute the target profile when it is required. Thus, the theoretical type information of CSP-CASL signature morphisms is captured in Haskell partly by the type system and partly in functions which check whether certain conditions are meet, for example, downward closure. This is inline with how HETS implements CASL signature morphisms. The channel mapping works in a similar way.

Many more implementation details have been coded within HETS which now allow the parsing and static analysis of Structured CSP-CASL. Once the various type classes, signature morphisms and supporting functions were implemented within HETS, the HETS framework gave us the ability to parse and statically analyse Structured CSP-CASL specifications. This was possible as the structuring operators (see Section 4.7) are institution independent and have been implemented as such.

Further functions allow HETS to check that CSP-CASL signature morphisms obey the *reflection* and *weak non-extension* conditions described in Section 4.4. Here, we present the code that performs the *weak non-extension* check.

```
checkWNECondition :: Morphism f CspSign CspAddMorphism -> Result ()
checkWNECondition Morphism
  { msource = sig
  , mtarget = sig'
  , sort_map = sm } = do
  let rel' = sortRel sig'
      supers s signature = Set.insert s $ supersortsOf s signature
      allPairsInSource = LT.cartesian $ sortSet sig
      commonSuperSortsInTarget s1 s2 = Set.intersection
                                       (supers (mapSort sm s1) sig')
                                       (supers (mapSort sm s2) sig')
      {- Candidates are triples (s1,s2,u') such that
      sigma(s1),sigma(s2) < u' -}
      createCandidateTripples (s1, s2) =
        Set.map (\ u' -> (s1, s2, u'))
```

```
        (commonSuperSortsInTarget s1 s2)
    allCandidateTripples =
      Set.unions $ Set.toList $ Set.map createCandidateTripples
      allPairsInSource
    testCandidate (s1, s2, u') =
      let possibleWitnesses = Set.intersection (supers s1 sig)
                                               (supers s2 sig)
          test t = Rel.path (mapSort sm t) u' rel' ||
                   mapSort sm t == u'
      in or $ Set.toList $ Set.map test possibleWitnesses
    failures = Set.filter (not . testCandidate)
               allCandidateTripples
    produceDiag (s1, s2, u') =
      let x = (mapSort sm s1)
          y = (mapSort sm s2)
      in Diag Error
        ("CSP-CASL Signature Morphism Weak Non-Extension Property "
         ++ "Violated:\n'"
         ++ showDoc
            (Subsort_decl [x, y] u' nullRange :: SORT_ITEM ())
            "' in target\nbut no common supersort for the sorts\n'"
         ++ showDoc
            (Sort_decl [s1, s2] nullRange :: SORT_ITEM ())
            "' in source")
        nullRange
    allDiags = map produceDiag $ Set.toList failures
unless (Set.null failures)
  ( Result allDiags Nothing) -- failure with error messages
```

This code takes a CSP-CASL signature morphism $(\sigma, \nu)$ (technically this may not be a CSP-CASL morphism as it may fail the weak-non-extension condition) and produces a verdict, pass or fail, but where extra diagnosis information, such as warnings or error messages, may also be returned. This code uses monads and can been seen as largely procedural. The goal here is to produce a set of all candidates and then check each passes a test. The ones that do not are failures. If there are any failures then we create error messages for each and return a failed verdict.

A candidate is a triple $(s_1, s_2, u')$ such that $\sigma(s_1) \leq' u'$ and $\sigma(s_2) \leq' u'$. We produce all candidates by starting with all pairs $(s_1, s_2)$ of sorts in the source signature and adding all common super-sorts $u'$ (after translating $s_1$ and $s_2$ with $\sigma$) to create the triples. We then test each triple by taking all common super sorts $t$ of $s_1$ and $s_2$ and checking that $\sigma(t) \leq' u'$ (actually we check $\sigma(t) <' u'$ or $\sigma(t) = u'$). If there is at least one super sort that passes this test then this is a witness for the candidate and the candidate passes the test. We then take all candidates which fail the test, we call these failures, and produce a meaningful error message from them. Assuming there is at least one failure, we return a failed verdict with the error messages. Otherwise we return a passed verdict with no error messages.

The run-time of this check seems reasonable. HETS requires only a few seconds to parse and statically analyse our online shop example. most of this time is spent loading and parsing

standard CASL libraries.

## 11.3 Static Semantics of Structured CSP-CASL

Here, we summarise the static semantics and the static analysis of Structured CSP-CASL specifications.

Gimblett [Gim08] programmed the original static analysis for basic CSP-CASL specifications. We had to slightly modify this in the implementation of Structured CSP-CASL. In essence we just added support for extracting the symbols of a specifications. This is a technical detail to support the use of CSP-CASL signature morphisms.

In order for HETS to support Structured CSP-CASL it was necessary to code CSP-CASL signature morphisms within HETS. This involved extending the parser to support symbol maps of process names and channel names. Further to this, analysis of signature morphisms had to be implemented, for example, the *weak-non-extension* check discussed in Section 11.2. With these features implemented, the HETS machinery allowed for static analysis of Structured CSP-CASL automatically with one problem. The local top sorts check was not run on signatures created via union of specifications (i.e., the structured **and** operator on specifications with different signatures). This allowed one to take the union of two specification with local top sorts and create an invalid CSP-CASL specification which violated the condition. Through communication with the HETS development team at the University of Bremen, the HETS framework was modified to allow this extra check to be carried out for such unions. This corrected the problem and now HETS correctly reports problematic CSP-CASL specifications. An example demonstrates this new check in the next section.

## 11.4 HETS in Action

In this section we demonstrate the abilities of HETS on selected sample specifications. We run HETS on the online shop specifications (see Chapter 10 and Appendix D) to check that they are indeed valid CSP-CASL specifications. HETS has checked that the syntax is valid and that they pass static analysis. The static analyser checks all the original CASL conditions. For instance, it checks that each symbol is declared before it is used in sentences. The extra signature morphism conditions *reflection* and *weak non-extension* conditions as described in Section 4.4 are also enforced. We are able to use all the structuring mechanisms presented in Section 4.7 within Structured CSP-CASL specifications. All implicit signature morphisms, such as those introduced by extensions using the **then** construct, are checked that they also satisfy these conditions. It would be easy to falsely assume a Structured CSP-CASL specification meets these requirements without tool support.

We first use HETS to check that the online shop specifications are correct. Figure 10.4 shows the development graph after HETS has successfully parsed and analysed the online shop specifications. The black arrows show the import structure of the specifications, while the red arrows show open proof obligations resulting from the instantiations. HETS produced the development graph with no errors reported from the specifications.

Further to this, all Structured CSP-CASL specifications in this thesis have been checked with HETS to make sure they are valid Structured CSP-CASL specifications.

We now focus on showing invalid examples where HETS reports meaningful error messages to the user. Consider the specification:

**logic** CSPCASL

**spec** HUGO =
  **data sort** $s < t$
**end**

**spec** ERNA =
  **data sort** $s < u$
**end**

**spec** UNION =
  HUGO **and** ERNA
**end**

Both specifications HUGO and ERNA are valid CSP-CASL specifications as they have local top sorts. However when we take the union using the structured **and** operator, we violate the local top sorts condition of CSP-CASL signatures (see Section 4.4). The target signature with three sorts (s), (t) and (u) with (s) being a sub-sort of (t) and (u) violates this property. Thus, the specifications are invalid. HETS produces the following error message:

```
Analyzing file Example1.het as library Example1
logic CspCASL
Analyzing spec Hugo
Analyzing spec Erna
Analyzing spec Union
*** Error /home/liam/Desktop/CC/Example1.het:12.8,
local top element obligation (s < u, t) unfulfilled
hets-svn: user error (Stopped due to errors)
```

HETS reports clear information to the user about the error. This check was originally implemented by Gimblett [Gim08] but had to be additionally performed each time a new specification was created via the various structuring operators.

A similar example shows that HETS checks for the *reflection* property of CSP-CASL signature morphisms.

**logic** CSPCASL

**spec** HUGO =
  **data sorts** $s, t$
**end**

**spec** ERNA =
    HUGO
**then data sort** $s < t$
**end**

Here, HUGO and ERNA are a valid CSP-CASL specifications, however the signature morphism induced by the **then** operator violates the *reflection* condition as $s$ is a sub-sort of $t$ in ERNA but not in HUGO. HETS reports the following message:

```
Analyzing file Example2.het as library Example2
logic CspCASL
Analyzing spec Hugo
Analyzing spec Erna
*** Error /home/liam/Desktop/CC/Example2.het:8.3-9.1,
CSP-CASL Signature Morphism Refl Property Violated:
'sort s < t' in target but not in source
'sort s < t'
hets-svn: user error (Stopped due to errors)
```

Using the reported line numbers the user is quickly able to track down the problem.

Finally, we show HETS reporting violations of the *weak-non-extension* property.

**logic** CSPCASL

**spec** HUGO =
    **data sorts** $S, T$
**end**

**spec** ERNA =
    HUGO
**then data sorts** $S, T < U$
**end**

Similar to the previous example, both HUGO and ERNA are a valid CSP-CASL specifications. However, the induced signature morphism violates the *weak-non-extension* property. HETS reports this as:

```
Analyzing file Example3.het as library Example3
logic CspCASL
Analyzing spec Hugo
Analyzing spec Erna
*** Error /home/liam/Desktop/CC/Example3.het:8.3-9.1,
CSP-CASL Signature Morphism Weak Non-Extension Property
Violated:
's, t < u' in target
but no common supersort for the sorts
```

```
's, t' in source
*** Error /home/liam/Desktop/CC/Example3.het:8.3-9.1,
CSP-CASL Signature Morphism Weak Non-Extension Property
Violated:
't, s < u' in target
but no common supersort for the sorts
't, s' in source
hets-svn: user error (Stopped due to errors)
```

There are two errors here as the property is violated in two symmetric ways.

In this chapter we have discussed the existing implementation of HETS and how we have extended it using the theoretical notions developed within this thesis. Specifically, we have provided a prototypical implementation of the CSP-CASL institutions. This shows that the constructions within this thesis have a practical effect and allow HETS to parse and statically analyse Structured CSP-CASL specifications. This level of tool support is of critical assistance to any user wishing to write such specifications, as errors can be easily over looked.

# Part III

# Conclusion

*Chapter 12*

# Summary

In this thesis, we have defined:

1. the first fully institutional semantics for CSP-CASL relative to the three main CSP semantics, namely, the Traces semantics $\mathcal{T}$, the Failures/Divergences semantics $\mathcal{N}$, and the Stable-Failures semantics $\mathcal{F}$,
2. (to the best of our knowledge) the first setting of dealing with loose processes, and
3. a set of proof calculi for modular reasoning along the structure of CSP-CASL specifications.

An example demonstrates that our concepts are useful in specification practice: not only could we separate concerns while developing our shop example by using suitable structuring mechanisms, we also could use this very same structure for modular reasoning. Finally, a prototypical implementation of the CSP-CASL institutions shows that our concepts are suited for tools.

We have extended CSP-CASL to include so called loose process semantics. This allows processes to be specified without pinning them down to particular instances. It is this looseness in processes (CSP-CASL already supported loose data) which enhances CSP-CASL's expressive power and allows for the capture of systems at high levels of abstraction. This allows an integrated approach to dealing with processes and data in a model theoretic way where systems can be developed in both their data and behavioural aspects.

We have enabled full structuring in CSP-CASL, thus complementing the existing ability of structuring the data part of a specification with the ability to also structure the process part. This allows for real component based development. This structuring includes the ability to create and instantiate generic specifications in CSP-CASL. These enhancements have been achieved by formulating CSP-CASL as various institutions. Full structuring support, which includes parametrisation and instantiation, would not be possible without the extension of loose process semantics addressed above.

As a consequence of this new construction we found that the original refinement notion for CSP-CASL needed updating as it did not cater for loose processes. This is a result of merging

the model theoretic approach of CASL with the denotational approach of CSP. To this end, we developed a new refinement notion to support the development of Structured CSP-CASL specifications with loose process semantics. Our new refinement notion reconciles these two opposing worlds from CASL and CSP. It is this notion of refinement that allows for the formal development of Structured CSP-CASL specifications.

To aid reasoning and proof efforts on Structured CSP-CASL specifications we have developed several proof calculi. These proof calculi allow for properties to be established on structured specifications by establishing related properties on their constituent parts. This means that verification of such structured specifications can be supported by the existing theorem prover CSP-CASL-Prover without any changes, as our refinement calculi can break down proof obligations on structured specifications until only proof obligations on basic specifications remain. At this point the existing tools can discharge such proof obligations on basic CSP-CASL specifications.

Whilst the underlying theorem prover did not need any modifications, we have extended the tool HETS [MML07] with the new version of CSP-CASL.[1] This extension allows HETS to support, among other features, parsing and static analysis of Structured CSP-CASL specifications.

We illustrated compositional modelling in Structured CSP-CASL with an example of an online shopping system. Additionally, we demonstrated our compositional calculi by proving deadlock freedom whilst utilising a specialised rule for networks.

In summary, we have have developed the first structured specification approach for dealing with reactive systems in process algebra, where:

- We present the first integrated specification language for processes and data, where both processes and data are fully and equally supported in a refinement based paradigm.
- Systems can be specified in a compositional manner.
- Systems can be specified at various levels of abstraction, from the high architectural levels to concrete implementation levels.
- Various sound calculi allow for compositional reasoning and verification.
- A completeness result rounds off our study of compositional reasoning.
- We demonstrate our approach with an example of real life quality.

[1] We have extended HETS in cooperation with the HETS development team at Bremen University.

*Chapter 13*

# Future Work

Whilst this thesis has shown how to formalise CSP-CASL as various institutions and how compositional reasoning can be performed over Structured CSP-CASL specifications, there are many more areas that can be studied. In the closer context of CSP-CASL, the following points could be explored: CSP replicated operators, overloading on process names, a method of allowing the communication sets of process names to be shrunk, improved **and** rule within the calculi, structuring supporting the use of the structured **free** operator, more examples of industrial strength, and implementation of the calculi so that reasoning is automated as far as possible.

In the context of integrating process algebra and algebraic specification, one has the impression that the current constructions have a general categorical nature, which should be explored further. For instance, one would like to see a $\pi$-CASL "automatically" constructed by proving that the $\pi$-calculus has a certain set of properties.

Finally, in industrial practice, for example by Rolls-Royce in avionics, reactive systems are often specified in UML. Here, composite structure diagrams "glue" together state machines, which communicate data specified in OCL (a first order logic). In this context, the same fundamental questions arise that we treated in an exemplary way in this thesis. Namely: what kind of structuring operations are "compositional"? Is there an appropriate development relation available, for example, relating UML diagrams? Can one reason in a modular way over the specifications? The semantic setup given in this thesis might provide pointers for solving these questions in the UML context.

# Part IV

# Appendices

*Appendix A*

# Deferred Proofs

This appendix contains full proofs which were deferred from earlier in the thesis.

## A.1 From Chapter 7

**Lemma 7.6** $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ are valid functors, that is, they preserve identity morphisms and functional composition.

*Proof.* We prove that identity morphisms and functional composition are preserved separately for each CSP semantics $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$.

**$\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$ preserve identity morphisms** This follows from the definitions, the healthiness conditions on the domains and the fact functions $\alpha^{\checkmark}$, $\alpha^{*\checkmark}$ and $\alpha^{\mathcal{P}\checkmark}$ preserve identity.

**$\mathcal{T}$ preserves composition of functions** Let $\alpha_1 : A \to B$ and $\alpha_2 : B \to C$ be two alphabet translations and let $T \in \mathcal{T}(A)$.

$$\begin{array}{lcll} (\alpha_2^{\mathcal{T}} \circ \alpha_1^{\mathcal{T}})(T) & = & \alpha_2^{\mathcal{T}}(\{\alpha_1^{*\checkmark}(t) \mid t \in T\}) & \text{By definition of } \alpha_1^{\mathcal{T}} \\ & = & \{(\alpha_2^{*\checkmark} \circ \alpha_1^{*\checkmark})(t) \mid t \in T\} & \text{By definition of } \alpha_2^{\mathcal{T}} \\ & = & \{(\alpha_2 \circ \alpha_1)^{*\checkmark}(t) \mid t \in T\} & \text{By Lemma 7.1} \\ & = & (\alpha_2 \circ \alpha_1)^{\mathcal{T}}(T) & \text{By definition of } (\alpha_2 \circ \alpha_1)^{\mathcal{T}} \end{array}$$

**$\mathcal{F}$ preserves composition of functions** Let $\alpha_1 : A \to B$ and $\alpha_2 : B \to C$ be two alphabet translations and let $(T, F) \in \mathcal{F}(A)$. We show:

1. $\mathit{fst}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F)) = \mathit{fst}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$, and
2. $\mathit{snd}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F)) = \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$.

The first point follows directly from the traces proof of this lemma. We now show the second point via subset inclusions.

**'$\subseteq$' direction** that is, $\mathit{snd}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F)) \subseteq \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$. Let $(s'', X'') \in \mathit{snd}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F))$. We show must $(s'', X'') \in \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$, that

is, we must find a failure $(t, Y) \in F$ such that $(\alpha_2 \circ \alpha_1)^{*\checkmark}(t) = s''$ and $\forall x'' \in C^\checkmark \bullet x'' \in X'' \Longrightarrow ((\alpha_2 \circ \alpha_1)^\checkmark)^-(x'') \subseteq Y$.

As $(s'', X'') \in \mathit{snd}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F))$, we know there exists failures $(s', X') \in \mathit{snd}(\alpha_1^{\mathcal{F}}(T, F))$ and $(s, X) \in F$ such that the following hold

- $\alpha_2^{*\checkmark}(s') = s''$,
- $\alpha_1^{*\checkmark}(s) = s'$,
- $\forall x'' \in C^\checkmark \bullet x'' \in X'' \Longrightarrow (\alpha_2^\checkmark)^-(x'') \subseteq X'$, and
- $\forall x' \in B^\checkmark \bullet x' \in X' \Longrightarrow (\alpha_1^\checkmark)^-(x') \subseteq X$.

Choose $(t, Y) := (s, X)$, then we have $(t, Y) \in F$ and $(\alpha_2 \circ \alpha_1)^{*\checkmark}(t) = s''$ by Lemma 7.1. Left to show is $\forall x'' \in C^\checkmark \bullet x'' \in X'' \Longrightarrow ((\alpha_2 \circ \alpha_1)^\checkmark)^-(x'') \subseteq Y$. Let $x'' \in X''$ and $x \in ((\alpha_2 \circ \alpha_1)^\checkmark)^-(x'')$, then we show $x \in Y$. As $x \in ((\alpha_2 \circ \alpha_1)^\checkmark)^-(x'')$, we know $(\alpha_2 \circ \alpha_1)^\checkmark(x) = x''$ and by Lemma 7.1 we get $(\alpha_2^\checkmark(\alpha_1^\checkmark(x))) = x''$. Thus we know $\alpha_1^\checkmark(x) \in (\alpha_2^\checkmark)^-(x'')$, thus $\alpha_1^\checkmark(x) \in X'$. As $x \in (\alpha_1^\checkmark)^-(x')$, we know $x \in X$, thus $x \in Y$.

**'$\supseteq$' direction** that is, $\mathit{snd}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F)) \supseteq \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$. Let $(s'', X'') \in \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$. We must show $(s'', X'') \in \mathit{snd}((\alpha_2^{\mathcal{F}} \circ \alpha_1^{\mathcal{F}})(T, F))$, that is, we must find failures $(t', Y') \in \mathit{snd}(\alpha_1^{\mathcal{F}}(T, F))$ and $(t, Y) \in F$ such that the following hold

- $\alpha_2^{*\checkmark}(t') = s''$,
- $\alpha_1^{*\checkmark}(t) = t'$,
- $\forall x'' \in C^\checkmark \bullet x'' \in X'' \Longrightarrow (\alpha_2^\checkmark)^-(x'') \subseteq Y'$, and
- $\forall x' \in B^\checkmark \bullet x' \in Y' \Longrightarrow (\alpha_1^\checkmark)^-(x') \subseteq Y$.

As $(s'', X'') \in \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{F}}(T, F))$, we know there exists $(s, X) \in F$ such that $(\alpha_2 \circ \alpha_1)^{*\checkmark}(s) = s''$ and $\forall x'' \in C^\checkmark \bullet x'' \in X'' \Longrightarrow ((\alpha_2 \circ \alpha_1)^\checkmark)^-(x'') \subseteq X$. Choose

$$\begin{array}{lcl} (t, Y) & := & (s, X) \\ (t', Y') & := & (\alpha^{*\checkmark}(s), \{x' \in B^\checkmark \mid \alpha_2^\checkmark(x') \in X''\}) \end{array}$$

Then we have $(t, Y) \in F$, $\alpha^{*\checkmark}(t) = t'$. As we know $\alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(s)) = s''$, by Lemma 7.1 and substitution of $t'$ we get $\alpha_2^{*\checkmark}(t') = s''$. It is also the case that $\forall x'' \in C^\checkmark \bullet x'' \in X'' \Longrightarrow (\alpha_2^\checkmark)^-(x'') \subseteq Y'$ follows directly from the construction of $Y'$. Left to show is $\forall x' \in B^\checkmark \bullet x' \in Y' \Longrightarrow (\alpha_1^\checkmark)^-(x') \subseteq Y$. Let $x' \in Y'$ and $x' \in (\alpha_1^\checkmark)^-(x')$, then we know $\alpha_2^\checkmark(\alpha_1^\checkmark(x)) \in X''$. Thus by Lemma 7.1 we get $(\alpha_2 \circ \alpha_1)^\checkmark(x)) \in X''$, which gives us $x \in Y$.

**$\mathcal{N}$ preserves composition of functions** Let $\alpha_1 : A \to B$ and $\alpha_2 : B \to C$ be two alphabet translations and let $(F, D) \in \mathcal{N}(A)$. We show:

1. $\mathit{fst}((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D)) = \mathit{fst}((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$, and
2. $\mathit{snd}((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D)) = \mathit{snd}((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$.

We first show the divergences are equal (i.e., Point 1) and then we show the failures are equal (i.e., Point 2). We show both of these by showing subset inclusions.

**Divergences '$\subseteq$' direction** i.e., $snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F,D)) \subseteq snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F,D))$. Let $s'' \in snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F,D))$. We must show $s'' \in snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F,D))$, that is, we must find a trace $u \in D$ and extension $v'' \in C^{*\checkmark}$ such that

- $(\alpha_2 \circ \alpha_1)^{*\checkmark}(u) \frown v'' = s''$, and
- if $u$ ends in $\checkmark$ then $v'' = \langle\rangle$.

We know there exists traces $s' \in snd(\alpha_1^{\mathcal{N}}(F,D))$ and $s \in D$ and extensions $t'' \in C^{*\checkmark}$ and $t' \in B^{*\checkmark}$ such that the following hold

- $\alpha_2^{*\checkmark}(s') \frown t'' = s''$,
- $\alpha_1^{*\checkmark}(s) \frown t' = s'$,
- if $s'$ ends in $\checkmark$ then $t'' = \langle\rangle$, and
- if $s$ ends in $\checkmark$ then $t' = \langle\rangle$.

Choose $u := s$ and $v'' := \alpha_2^{*\checkmark}(t') \frown t''$. As $\alpha_1^{*\checkmark}$ and $\alpha_2^{*\checkmark}$ are functorial (see Lemma 7.1), we know $\alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(s)) \frown \alpha_2^{*\checkmark}(t') \frown t'' = s''$. By Lemma 7.1 and substitution of $u$ and $v''$, we get we get $(\alpha_2 \circ \alpha_1)^{*\checkmark}(u) \frown v'' = s''$. Left to show is that if $u$ ends in $\checkmark$ then $v'' = \langle\rangle$. Assume $u$ does end in $\checkmark$, then we know $t' = \langle\rangle$. As $\alpha_1^{*\checkmark}$ preserves $\checkmark$, then $s'$ also ends in $\checkmark$, thus we have $t'' = \langle\rangle$. Finally, we can conclude $v'' = \alpha_2^{*\checkmark}(t') \frown t'' = \langle\rangle \frown \langle\rangle = \langle\rangle$.

**Divergences '$\supseteq$' direction** i.e., $snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F,D)) \supseteq snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F,D))$. Let $(s'', X'') \in snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F,D))$. We must show $(s'', X'') \in snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F,D))$, that is, We must find traces $u' \in snd(\alpha_1^{\mathcal{N}}(F,D))$ and $u \in D$ and extensions $v'' \in C^{*\checkmark}$ and $v' \in B^{*\checkmark}$ such that the following hold

- $\alpha_2^{*\checkmark}(u') \frown v'' = s''$,
- $\alpha_1^{*\checkmark}(u) \frown v' = u'$,
- if $u'$ ends in $\checkmark$ then $v'' = \langle\rangle$, and
- if $u$ ends in $\checkmark$ then $v' = \langle\rangle$.

We know there exists a trace $s \in D$ and extension $t'' \in C^{*\checkmark}$ such that

- $(\alpha_2 \circ \alpha_1)^{*\checkmark}(s) \frown t'' = s''$, and
- if $s$ ends in $\checkmark$ then $t'' = \langle\rangle$.

Choose $u := s$, $u' := \alpha_1^{*\checkmark}(s)$, $v' := \langle\rangle$, and $v'' := t''$. As $u = s$ by application of $\alpha_1^{*\checkmark}$ we obtain $\alpha_1^{*\checkmark}(u) := \alpha_1^{*\checkmark}(s)$. Hence we know $\alpha_1^{*\checkmark}(u) \frown \langle\rangle = \alpha_1^{*\checkmark}(s)$. By substitution of $u'$ and $v'$ we get $\alpha_1^{*\checkmark}(u) \frown v' = u'$.

As $u' := \alpha_1^{*\checkmark}(s)$ by application of $\alpha_2^{*\checkmark}$ we obtain $\alpha_2^{*\checkmark}(u') := \alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(s))$. By Lemma 7.1 and concatenation of $t''$ we get $\alpha_2^{*\checkmark}(u') \frown t'' := (\alpha_2 \circ \alpha_1)^{*\checkmark}(s) \frown t''$. By substitution of $s''$ and $v''$ we get $\alpha_2^{*\checkmark}(u') \frown v'' := s''$.

The statement 'if $u$ ends in $\checkmark$ then $v' = \langle\rangle$' holds trivially as $v' = \langle\rangle$. Left to show is that if $u'$ ends in $\checkmark$ then $v'' = \langle\rangle$. Assume $u'$ ends in $\checkmark$, then $\alpha_1^{*\checkmark}(s)$ ends in $\checkmark$. As $\alpha_1^{*\checkmark}$ preserves $\checkmark$, we know $s$ ends in $\checkmark$. Thus $v'' = t'' = \langle\rangle$ .

**Failures '$\subseteq$' direction** that is, $fst((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D)) \subseteq fst((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$. Let $(s'', X'') \in fst((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D))$, we show $(s'', X'') \in fst((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$. We know either $(s'', X'')$ originated from a failure or a divergence.

**Case 1** $(s'', X'')$ originated from a failure, that is, we know there exists $(s', X') \in fst(\alpha_1^{\mathcal{N}}(F, D))$ such that

- $\alpha_2^{*\checkmark}(s') = s''$, and
- $\forall x'' \in C^{\checkmark} \bullet x'' \in X'' \implies (\alpha_2^{\checkmark})^{-}(x'') \subseteq X'$.

This can also happen in two cases.

**Case 1.1** $(s', X')$ originated from a failure, i.e., we know there exists $(s, X) \in F$ such that

- $\alpha_1^{*\checkmark}(s) = s'$, and
- $\forall x' \in B^{\checkmark} \bullet x' \in X' \implies (\alpha_1^{\checkmark})^{-}(x') \subseteq X$.

Then this case is identical to the Stable-Failures proof of failures subset in '$\subseteq$' direction.

**Case 1.2** $(s', X')$ originated from a divergence, that is, $s' \in snd(\alpha_1^{\mathcal{N}}(F, D))$. Then we know $s'' \in snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D))$ by construction. As we have shown divergences are equal already, we know $s'' \in snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$. Then $(s'', X'') \in fst((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$ by construction.

**Case 2** $(s'', X'')$ originated from a divergence, i.e., $s'' \in snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D))$. As we have shown divergences are equal already, we know $s'' \in snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$. Then $(s'', X'') \in fst((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$ by construction.

**Failures '$\supseteq$' direction** that is, $fst((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D)) \supseteq fst((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$. Let $(s'', X'') \in fst((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$, we show $(s'', X'') \in fst((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D))$. We know either $(s'', X'')$ originated from a failure or a divergence.

**Case 1** $(s'', X'')$ originated from a failure, that is, we know there exists $(s, X) \in F$ such that

- $(\alpha_2 \circ \alpha_1)^{*\checkmark}(s) = s''$, and
- $\forall x'' \in C^{\checkmark} \bullet x'' \in X'' \implies ((\alpha_2 \circ \alpha_1)^{\checkmark})^{-}(x'') \subseteq X$.

Then this case is identical to the Stable-Failures proof of failures subset in '$\supseteq$' direction.

**Case 2** $(s'', X'')$ originated from a divergence, i.e., $s'' \in snd((\alpha_2 \circ \alpha_1)^{\mathcal{N}}(F, D))$. As we have shown divergences are equal already, we know $s'' \in snd((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D))$. Then $(s'', X'') \in fst((\alpha_2^{\mathcal{N}} \circ \alpha_1^{\mathcal{N}})(F, D))$ by construction. □

**Lemma 7.8** If an alphabet translation $\alpha : A \to B$ is injective then the induced covariant domain translations $\alpha^{\mathcal{T}}$, $\alpha^{\mathcal{N}}$, and $\alpha^{\mathcal{F}}$ are also injective.

*Proof.* Let $\alpha : A \to B$ be an injective alphabet translation.

**Traces translation** Let $T_1$ and $T_2$ be trace sets in $\mathcal{T}(A)$ such that $\alpha^{\mathcal{T}}(T_1) = \alpha^{\mathcal{T}}(T_2)$. We show $T_1 \subseteq T_2$. Let $s$ be a trace in $T_1$. We know $\alpha^{*\checkmark}(s) \in \alpha^{\mathcal{T}}(T_1)$, therefore $\alpha^{*\checkmark}(s) \in$

$\alpha^{\mathcal{T}}(T_2)$. As $\alpha^{*\checkmark}$ is injective we know $s \in T_2$. Showing $T_2 \subseteq T_1$ is analogous. Thus $T_1 = T_2$.

**Failures-divergences translation** Let $(F_1, D_1)$ and $(F_2, D_2)$ be denotations of $\mathcal{N}(A)$ such that $\alpha^{\mathcal{N}}((F_1, D_1)) = \alpha^{\mathcal{N}}((F_2, D_2))$. We first show $D_1 = D_2$ and then $F_1 = F_2$.

We show $D_1 \subseteq D_2$. Let $s_1 \in D_1$. We know $\alpha^{*\checkmark}(s_1) \in \mathit{snd}(\alpha^{\mathcal{N}}((F_1, D_1))$ by construction, thus $\alpha^{*\checkmark}(s) \in \mathit{snd}(\alpha^{\mathcal{N}}((F_2, D_2))$. Therefore we know there exists $s_2 \in D_2$ and $t_2' \in B^{*\checkmark}$ such that $\alpha^{*\checkmark}(s_2) \frown t_2' = \alpha^{*\checkmark}(s_1)$ and if $s_2$ ends in $\checkmark$ then $t_2' = \langle\rangle$.

**Case 1** Trace $s_2$ ends in $\checkmark$. We know $\alpha^{*\checkmark}(s_2) = \alpha^{*\checkmark}(s_1)$. As $\alpha^{*\checkmark}$ is injective, we know $s_2 = s_1$, thus $s_1 \in D_2$.

**Case 2** Trace $s_2$ does not end in $\checkmark$. As $s_1$ only uses symbols from $A^{\checkmark}$ we know $\alpha^{*\checkmark}(s_1) \in (\alpha^{\mathcal{P}\checkmark}(A))^{*\checkmark}$. Furthermore, as $\alpha^{*\checkmark}(s_1) = \alpha^{*\checkmark}(s_2) \frown t_2'$, we know $\alpha^{*\checkmark}(s_2) \frown t_2' \in (\alpha^{\mathcal{P}\checkmark}(A))^{*\checkmark}$. Now as $s_2 \in D_2$ and $(F_2, D_2)$ is healthy (and satisfies condition *D1*) we know there exists divergence $u \in D_2$ such that $\alpha^{*\checkmark}(u) = \alpha^{*\checkmark}(s_2) \frown t_2' = \alpha^{*\checkmark}(s_1)$. As $\alpha^{*\checkmark}$ is injective, we know $u = s_1$, thus $s_1 \in D_2$.

Showing $D_2 \subseteq D_1$ is analogous. Thus $D_1 = D_2$.

We now prove $F_1 \subseteq F_2$. Let $(s_1, X_1) \in F_1$. As $\forall x' \in B^{\checkmark} \bullet x' \in \alpha^{\mathcal{P}\checkmark}(X_1) \implies (\alpha^{\checkmark})^{-}(x') \subseteq X_1$ we know $(\alpha^{*\checkmark}(s_1), \alpha^{\mathcal{P}\checkmark}(X_1)) \in \mathit{fst}(\alpha^{\mathcal{N}}((F_1, D_1)))$, thus we also know $(\alpha^{*\checkmark}(s_1), \alpha^{\mathcal{P}\checkmark}(X_1)) \in \mathit{fst}(\alpha^{\mathcal{N}}((F_2, D_2)))$. This can happen in two cases.

**Case 1** There exists $(s_2, X_2) \in F_2$ such that $\alpha^{*\checkmark}(s_1) = \alpha^{*\checkmark}(s_2)$ and $\forall x' \in B^{\checkmark} \bullet x' \in \alpha^{\mathcal{P}\checkmark}(X_1) \implies (\alpha^{\checkmark})^{-}(x') \subseteq X_2$. By the injectivity of $\alpha^{*\checkmark}$ we know $s_1 = s_2$. If we can show $X_1 \subseteq X_2$, then as $(F_2, D_2)$ is healthy and satisfies condition *F2* then we know $(s_1, X_1) \in F_2$. Let $x \in X_1$, then we know $\alpha^{\checkmark}(x) \in \alpha^{\mathcal{P}\checkmark}(X_1)$. Therefore we know $(\alpha^{\checkmark})^{-}(\alpha^{\checkmark}(x)) \subseteq X_2$ and as $x \in (\alpha^{\checkmark})^{-}(\alpha^{\checkmark}(x))$ we know $x \in X_2$. Thus we can conclude $(s_1, X_1) \in F_2$.

**Case 2** $\alpha^{*\checkmark}(s_1) \in \mathit{snd}(\alpha^{\mathcal{N}}((F_2, D_2))$. As $s_1 \in A^{*\checkmark}$, by the same argument that $D_1 \subseteq D_2$ above, we know $s_1 \in D_2$. As $(F_2, D_2)$ is healthy (and satisfies condition *D2*) we know $(s_1, X_1) \in F_2$.

Showing $F_2 \subseteq F_1$ is analogous. Thus $F_1 = F_2$.

**Stable-failures translation** Let $(T_1, F_1)$ and $(T_2, F_2)$ be two denotations of $\mathcal{F}(A)$ such that $\alpha^{\mathcal{F}}((T_1, F_1)) = \alpha^{\mathcal{F}}((T_2, F_2))$. We show $T_1 = T_2$ and $F_1 = F_2$. As $\alpha^{\mathcal{T}}$ is injective we know $T_1 = T_2$. The proof of $F_1 = F_2$ follows the same argument as the proof in case 1 of $F_1 = F_2$ of the Failures-divergences translation above. □

**Lemma 7.12** For all injective alphabet translations $\alpha : A \to B$, the contravariant domain translation $\hat{\alpha}^{\mathcal{D}}$ is healthy, that is, $\hat{\alpha}^{\mathcal{D}} : \mathcal{D}(B) \to \mathcal{D}(A)$ for all $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

*Proof.* As presented in [Kah10] we prove that the contravariant domain translations $\hat{\alpha}^{\mathcal{T}}$, $\hat{\alpha}^{\mathcal{N}}$, and $\hat{\alpha}^{\mathcal{F}}$ preserve the healthiness conditions of the Traces semantics $\mathcal{T}$, the Failures/Divergences semantics $\mathcal{N}$, and the Stable-Failures semantics $\mathcal{F}$, respectively.

**Traces semantics $\mathcal{T}$** We show that $\hat{\alpha}^{\mathcal{T}}$ preserves the condition *T1*. Let $T' \in \mathcal{T}(B)$ and $T = \hat{\alpha}^{\mathcal{T}}(T)$.

**T1** We show that $T$ is healthy, that is, non-empty and prefix closed. We know that $T'$ is non-empty and prefix closed, thus $\langle\rangle \in T'$. As $\alpha^{*\checkmark}(\langle\rangle) = \langle\rangle$ we know that $\langle\rangle \in T$. Let $t \in T$ and $s \leqslant t$. We know $\alpha^{*\checkmark}(t) \in T'$ by definition and $\alpha^{*\checkmark}(s) \leqslant \alpha^{*\checkmark}(t)$. As $T'$ is prefix closed we know $\alpha^{*\checkmark}(s) \in T'$, therefore $s \in T$ by construction.

**Stable-Failures semantics $\mathcal{F}$** We show that $\hat{\alpha}^{\mathcal{F}}$ preserves the conditions *T1*, *T2*, *T3*, *F2*, *F3*, and *F4*. Let $(T', F') \in \mathcal{F}(B)$ and $(T, F) = \hat{\alpha}^{\mathcal{F}}(T', F')$.

**T1** See the proof for the Traces semantics condition *T1*.

**T2** Let $(s, X) \in F$, we show $s \in T$. As $(s, X) \in F$ then we know there exists $(s', X') \in F'$ such that $\alpha^{*\checkmark}(s) = s'$. As $(T', F')$ is healthy (and satisfies condition *T2*), we know that $s' \in T'$. Thus, we know there exists $t \in T$ such that $\alpha^{*\checkmark}(t) = s' = \alpha^{*\checkmark}(s)$. By injectivity of $\alpha$ we know $t = s$, hence $s \in T$.

**T3** Let $s^\frown\langle\checkmark\rangle \in T$, we show $(s^\frown\langle\checkmark\rangle, X) \in F$ for all $X \subseteq A^\checkmark$. Let $X \subseteq A^\checkmark$. We know there exists $s'^\frown\langle\checkmark\rangle \in T'$ such that $\alpha^{*\checkmark}(s) = s'$. As $(T', F')$ is healthy (and satisfies condition *T3*), we have that $(s'^\frown\langle\checkmark\rangle, X') \in F'$ for all $X' \subseteq B^\checkmark$, thus, especially $(s'^\frown\langle\checkmark\rangle, \alpha^{\mathcal{P}\checkmark}(X)) \in F'$. Therefore $(s^\frown\langle\checkmark\rangle, X) \in F$.

**F2** Let $(s, X) \in F$ and $Y \subseteq X$, we show $(s, Y) \in F$. We know there exists $(s', X') \in F'$ such that $\alpha^{*\checkmark}(s) = s'$ and $X' \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(X)$. Let $Y' = \alpha^{\mathcal{P}\checkmark}(Y)$, then we know $Y' \subseteq X'$. As $(T', F')$ is healthy (and satisfies condition *F2*), we know that $(s', Y') \in F'$. Therefore $(s, Y) \in F$ by construction as $Y' \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(Y)$.

**F3** Let $(s, X) \in F$ and $Y \subseteq A^\checkmark$ such that $\forall y \in Y \bullet s^\frown\langle y\rangle \notin T$, we show $(s, X \cup Y) \in F$. As $(s, X) \in F$ then there exists $(s', X') \in F'$ such that $\alpha^{*\checkmark}(s) = s'$ and $X' \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(X)$. Let $Y' = \alpha^{\mathcal{P}\checkmark}(Y)$, then we know $\forall y' \in Y' \bullet s'^\frown\langle y'\rangle \notin T'$. As $(T', F')$ is healthy (and satisfies condition *F3*), we know $(s', X' \cup Y') \in F'$. Finally, as we also know $(X' \cup Y') \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(X \cup Y)$, we can conclude $(s, X \cup Y) \in F$.

**F4** Let $s^\frown\langle\checkmark\rangle \in T$, we show $(s, A) \in F$. We know there exists $s'^\frown\langle\checkmark\rangle \in T'$ such that $\alpha^{*\checkmark}(s) = s'$. As $(T', F')$ is healthy (and satisfies condition *F4*), we know $(s', B) \in F'$. Therefore $(s, A) \in F$ by construction as $B \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(A)$.

**Failures/Divergences semantics $\mathcal{N}$** We show that $\hat{\alpha}^{\mathcal{N}}$ preserves the conditions *F1*, *F2*, *F3*, *F4*, *D1*, *D2*, and *D3*. Let $(F', D') \in \mathcal{N}(B)$ and $(F, D) = \hat{\alpha}^{\mathcal{N}}(F', D')$.

**F1** We show that $tr_\perp(F, D)$ is non-empty and prefix closed. We know that $tr_\perp(F', D')$ is non-empty and prefix closed, thus $\langle\rangle \in tr_\perp(F', D')$, therefore $(\langle\rangle, \emptyset) \in F'$. By definition of $\hat{\alpha}^{\mathcal{N}}$ we know $(\langle\rangle, \emptyset) \in F$, thus $\langle\rangle \in tr_\perp(F, D)$.

Let $t \in tr_\perp(F, D)$ and $s \leqslant t$, we show $s \in tr_\perp(F, D)$. We know $(t, \emptyset) \in F$, therefore we know, by definition of $\hat{\alpha}^{\mathcal{N}}$, there exists $(t', X') \in F'$ such that $\alpha^{*\checkmark}(t) = t'$. As $(T', F')$ is healthy (and satisfies condition *F2*, we know $(t', \emptyset) \in F'$, thus

$t' \in tr_\perp(F', D')$. As $s \leqslant t$ we know $\alpha^{*\checkmark}(s) \leqslant \alpha^{*\checkmark}(t) = t'$, and by prefix closure of $tr_\perp(F', D')$, we know $\alpha^{*\checkmark}(s) \in tr_\perp(F', D')$. Therefore $(\alpha^{*\checkmark}(s), \emptyset) \in F'$, thus $(s, \emptyset) \in F$, hence we know $s \in tr_\perp(F, D)$.

**F2 and F3** The proofs are the same as for the conditions *F2* and *F3*, however in the case of *F3* we replace the trace set $T$ with $tr_\perp(F, D)$ (similar for $T'$) and use the definition of $tr_\perp$ accordingly.

**F4** Let $s \frown \langle\checkmark\rangle \in tr_\perp(F, D)$, we show $(s, A) \in F$. We know $(s \frown \langle\checkmark\rangle, \emptyset) \in F$ and therefore we know there exists $(s' \frown \langle\checkmark\rangle, X') \in F'$ such that $\alpha^{*\checkmark}(s \frown \langle\checkmark\rangle) = s' \frown \langle\checkmark\rangle$. As $(F', D')$ is healthy (and satisfies condition *F2*), we know $(s' \frown \langle\checkmark\rangle, \emptyset) \in F'$, thus $s' \frown \langle\checkmark\rangle \in tr_\perp(F', D')$. By condition *F4* we know $(s', B) \in F'$ and furthermore, by definition of $\hat{\alpha}^{\mathcal{N}}$, we can conclude $(s, A) \in F$ as $B \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(A)$.

**D1** Let $s \in D \cap A^*$ and $t \in A^{*\checkmark}$, we show $s \frown t \in D$. We know $\alpha^{*\checkmark}(s) \in D'$. As $(F', D')$ is healthy (and satisfies condition *D1*) we know $\alpha^{*\checkmark}(s) \frown \alpha^{*\checkmark}(t) \in D'$, therefore $s \frown t \in D$.

**D2** Let $s \in D$ and $X \in \mathcal{P}(A^\checkmark)$. We know by definition that $\alpha^{*\checkmark}(s) \in D'$. As $(F', D')$ is healthy (and satisfies condition *D2*) we know $(\alpha^{*\checkmark}(s), \alpha^{\mathcal{P}\checkmark}(X)) \in F'$. As $\alpha^{\mathcal{P}\checkmark}(X) \cap \alpha^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha^{\mathcal{P}\checkmark}(X)$, we can conclude $(s, X) \in F$.

**D3** Let $s \frown \langle\checkmark\rangle \in D$, we show $s \in D$. We know by definition that $\alpha^{*\checkmark}(s) \frown \langle\checkmark\rangle \in D'$. As $(F', D')$ is healthy (and satisfies condition *D3*) we know $\alpha^{*\checkmark}(s) \in D'$. Therefore $s \in D$ by construction. □

**Lemma 7.13** $\mathcal{T}^{op}$, $\mathcal{N}^{op}$, and $\mathcal{F}^{op}$ are valid functors, that is, they preserve identity morphisms and functional composition.

*Proof.* We prove that identity morphisms and functional composition are preserved separately for each CSP semantics $\mathcal{T}$, $\mathcal{N}$, and $\mathcal{F}$.

**$\hat{\alpha}^{\mathcal{T}}$, $\hat{\alpha}^{\mathcal{N}}$, and $\hat{\alpha}^{\mathcal{F}}$ preserve identity morphisms** This follows from the definitions, the healthiness conditions on the domains and the fact functions $\alpha^\checkmark$, $\alpha^{*\checkmark}$ and $\alpha^{\mathcal{P}\checkmark}$ preserve identity.

**$\hat{\alpha}^{\mathcal{T}}$ preserves composition of functions** Let $\alpha_1 : A \to B$ and $\alpha_2 : B \to C$ be two injective alphabet translations and let $T'' \in \mathcal{T}(C)$.

$$
\begin{array}{lcll}
(\hat{\alpha}_1^{\mathcal{T}} \circ \hat{\alpha}_2^{\mathcal{T}})(T'') & = & \hat{\alpha}_1^{\mathcal{T}}(\{s' \mid \alpha_2^{*\checkmark}(s') \in T''\}) & \text{By definition of } \hat{\alpha}_2^{\mathcal{T}} \\
& = & \{s \mid \alpha_1^{*\checkmark}(s) \in \{s' \mid \alpha_2^{*\checkmark}(s') \in T''\}\} & \text{By definition of } \hat{\alpha}_1^{\mathcal{T}} \\
& = & \{s \mid \alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(s)) \in T''\} & \\
& = & \{s \mid (\alpha_2 \circ \alpha_1)^{*\checkmark}(s) \in T''\} & \text{By Lemma 7.1} \\
& = & (\alpha_2 \hat{\circ} \alpha_1)^{\mathcal{T}}(T'') & \text{By definition of } (\alpha_2 \hat{\circ} \alpha_1)^{\mathcal{T}}
\end{array}
$$

**$\hat{\alpha}^{\mathcal{F}}$ preserves composition of functions** Let $\alpha_1 : A \to B$ and $\alpha_2 : B \to C$ be two injective alphabet translations and let $(T'', F'') \in \mathcal{F}(C)$. We show $(\hat{\alpha}_1^{\mathcal{F}} \circ \hat{\alpha}_2^{\mathcal{F}})(T'', F'') = (\alpha_2 \hat{\circ} \alpha_1)^{\mathcal{F}}(T'', F'')$

The proof for the traces component is the same as for the Traces semantics. For the failures component we show $snd((\hat{\alpha}_1^{\mathcal{F}} \circ \hat{\alpha}_2^{\mathcal{F}})(T'', F'')) = snd((\alpha_2 \hat{\circ} \alpha_1)^{\mathcal{F}}(T'', F''))$, by subset inclusion in both directions.

**'$\subseteq$' direction** Let $(s, X) \in snd((\hat{\alpha}_1^{\mathcal{F}} \circ \hat{\alpha}_2^{\mathcal{F}})(T'', F''))$. Thanks to the functor properties of the basic functions (Lemma 7.1) and unfolding the definitions we know there exists $(s', X') \in snd(\hat{\alpha}_2^{\mathcal{F}}(T'', F''))$ and $(s'', X'') \in F''$ such that $\alpha_1^{*\checkmark}(s) = s'$, $\alpha_2^{*\checkmark}(s') = s''$, $X' \cap \alpha_1^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha_1^{\mathcal{P}\checkmark}(X)$, and $X'' \cap \alpha_2^{\mathcal{P}\checkmark}(B^\checkmark) = \alpha_2^{\mathcal{P}\checkmark}(X')$. Thus we know $\alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(s)) = s''$. It remains to show that $X'' \cap \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(A^\checkmark)) = \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(X))$. This follows from definitions and injectivity of $\alpha_1$ and $\alpha_2$. We again establish this by showing subset inclusion in both directions.

**'$\subseteq$' direction** Let $x \in X'' \cap \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(A^\checkmark))$, then we know $x'' \in X''$ and there exists $x \in A^\checkmark$ such that $\alpha_2^\checkmark(\alpha_2^\checkmark(x)) = x''$. Therefore $x'' \in \alpha_2^{\mathcal{P}\checkmark}(B^\checkmark)$ and thus $x'' \in \alpha_2^{\mathcal{P}\checkmark}(X')$. By injectivity of $\alpha_2$ we then know $\alpha_1^\checkmark(x) \in X'$, thus $\alpha_1^\checkmark(x) \in \alpha_1^{\mathcal{P}\checkmark}(X)$. Finally allowing us to establish $x'' \in \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(X))$.

**'$\supseteq$' direction** Let $x'' \in \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(X))$, then we know there exists $x \in X$ such that $\alpha_2^\checkmark(\alpha_1^\checkmark(x)) = x''$. As we know $\alpha_1^\checkmark(x) \in \alpha_1^{\mathcal{P}\checkmark}(X)$ we also know $\alpha_1^\checkmark(x) \in X' \cap \alpha_1^{\mathcal{P}\checkmark}(A^\checkmark)$. As $\alpha_2^\checkmark(\alpha_1^\checkmark(x)) \in \alpha_2^{\mathcal{P}\checkmark}(X')$ we know $x'' \in X'' \cap \alpha_2^{\mathcal{P}\checkmark}(B^\checkmark)$. Thus we know $x'' \in X'' \cap \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(A^\checkmark))$.

**'$\supseteq$' direction** Let $(s, X) \in snd((\alpha_2 \hat{\circ} \alpha_1)^{\mathcal{F}}(T'', F''))$. Thanks to the functor properties of the basic functions (Lemma 7.1) and unfolding the definitions we know there exists $(s'', X'') \in F''$ such that $\alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(s)) = s''$ and $X'' \cap \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(A^\checkmark)) = \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(X))$. Choose $s' = \alpha_1^{*\checkmark}(s)$ and $X' = \{x' \in B^\checkmark \mid \alpha_2^{*\checkmark}(x') \in X''\}$. We must establish that $X'' \cap \alpha_2^{\mathcal{P}\checkmark}(B^\checkmark) = \alpha_2^{\mathcal{P}\checkmark}(X')$ and $X' \cap \alpha_1^{\mathcal{P}\checkmark}(A^\checkmark) = \alpha_1^{\mathcal{P}\checkmark}(X)$ hold. The first follows directly from the construction of $X'$. We establish the second by showing subset inclusion in both directions.

**'$\subseteq$' direction** Let $x' \in X' \cap \alpha_1^{\mathcal{P}\checkmark}(A^\checkmark)$. Then we know $x \in X'$ and there exists $x \in A^\checkmark$ such that $\alpha_1^{*\checkmark}(x) = x'$. By the construction of $X'$ we know $\alpha_2^\checkmark(x') \in X''$. By assumption we therefore know $\alpha_2^\checkmark(x') \in \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(X))$. Thus $x' \in \alpha_1^{\mathcal{P}\checkmark}(X)$.

**'$\supseteq$' direction** Let $x' \in \alpha_1^{\mathcal{P}\checkmark}(X)$, then there exists $x \in X$ such that $\alpha_1^{*\checkmark}(x) = x'$. Therefore we know $\alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(x)) \in \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(X))$ and thus by assumption that $\alpha_2^{*\checkmark}(\alpha_1^{*\checkmark}(x)) \in X'' \cap \alpha_2^{\mathcal{P}\checkmark}(\alpha_1^{\mathcal{P}\checkmark}(A^\checkmark))$. By the definition of $X'$ we know $\alpha_1^\checkmark(x) \in X'$, thus $x' \in X' \cap \alpha_1^{\mathcal{P}\checkmark}(A^\checkmark)$.

**$\hat{\alpha}^{\mathcal{N}}$ preserves composition of functions** Let $\alpha_1 : A \to B$ and $\alpha_2 : B \to C$ be two injective alphabet translations and let $(F'', D'') \in \mathcal{N}(C)$. We show $(\hat{\alpha}_1^{\mathcal{N}} \circ \hat{\alpha}_2^{\mathcal{N}})(F'', D'') = (\alpha_2 \hat{\circ} \alpha_1)^{\mathcal{N}}(F'', D'')$

The proof for the divergences component is the same as for the Traces semantics. The proof for the failures component is the same as for the Stable-Failures semantics. □

**Lemma 7.26** For all *ResSubPCFOL*$^{=}$ signature morphisms $\sigma : \Sigma \to \Sigma'$, and all $\Sigma'$-models $M'$, $\alpha_{\sigma,M'}$ is well defined and injective. This lemma is presented from [Kah10].

*Proof.* Let $\sigma : \Sigma \to \Sigma'$ be a restricted sub-sorted signature morphism, and $M'$ be a restricted sub-sorted $\Sigma'$-model where $S$ and $S'$ are the sort sets of the signatures $\Sigma$ and $\Sigma'$ respectively.

**Well definedness** We show that if $(s,x) \sim_{M'|_\sigma} (t,y)$ then $(\sigma(s),x) \sim_{M'} (\sigma(t),y)$ for all $s,t \in S$, $x \in (M'|_\sigma)_{\perp_s}$, and $y \in (M'|_\sigma)_{\perp_t}$ for all $s,t \in S$, $x \in (M'|_\sigma)_{\perp_s}$, and $y \in (M'|_\sigma)_{\perp_t}$.

Let $s,t \in S$, $x \in (M'|_\sigma)_{\perp_s}$, and $y \in (M'|_\sigma)_{\perp_t}$ such that $(s,x) \sim_{M'|_\sigma} (t,y)$. By the definition of $\sim_{M'|_\sigma}$ (defined in Section 7.2.1), there are two cases to consider:

**Case** $x = y = \perp$**:** As $(s,x) \sim_{M'|_\sigma} (t,y)$ holds, there exists sort $u \in S$ such that $s \leq u$ and $t \leq u$. By the property *p1* of $\sigma$ (see Section 4.3), we know $\sigma(s) \leq \sigma(u)$ and $\sigma(t) \leq \sigma(u)$. Thus $(\sigma(s),x) \sim_{M'} (\sigma(t),y)$ holds.

**Case** $x \neq \perp$ **and** $y \neq \perp$**:** We need to show that the following two conditions hold:

1. $\exists u' \in S'$ such that $\sigma(s) \leq u'$ and $\sigma(t) \leq u'$.
2. $\forall u' \in S'$ such that $\sigma(s) \leq u'$ and $\sigma(t) \leq u'$ the following holds:

$$(\mathtt{inj}_{\sigma(s),u'})_{M'}(x) = (\mathtt{inj}_{\sigma(t),u'})_{M'}(y)$$

For Condition 1, the proof is identical to the proof in Case 1. We now focus on Condition 2. Let $u' \in S'$ such that $\sigma(s) \leq u'$ and $\sigma(t) \leq u'$. By the *weak non-extension* property of $\sigma$ (see Section 4.4), we know there exists a sort $v \in S$ such that $s \leq v$, $t \leq v$, and $\sigma(v) \leq u'$. As $(s,x) \sim_{M'|_\sigma} (t,y)$ holds, we know $(\mathtt{inj}_{s,v})_{M'|_\sigma}(x) = (\mathtt{inj}_{t,v})_{M'|_\sigma}(y)$. Thus by the definition of reduct we know $(\mathtt{inj}_{\sigma(s),\sigma(v)})_{M'}(x) = (\mathtt{inj}_{\sigma(t),\sigma(v)})_{M'}(y)$. By applying the function $(\mathtt{inj}_{\sigma(v),u'})_{M'}$ to both sides of the previous equation and from the transitivity of injections (third) axiom which our sub-sorted model $M'$ respects (see Section 4.3), it follows that $(\mathtt{inj}_{\sigma(s),u})_{M'}(x) = (\mathtt{inj}_{\sigma(t),u})_{M'}(y)$.

**Injectivity** We show if $[(\sigma(s),x)]_{\sim_{M'}} = [(\sigma(t),y)]_{\sim_{M'}}$ then $[(s,x)]_{\sim_{M'|_\sigma}} = [(t,y)]_{\sim_{M'|_\sigma}}$, that is,

$$(\sigma(s),x) \sim_{M'} (\sigma(t),y) \Longrightarrow (s,x) \sim_{M'|_\sigma} (t,y)$$

for all $s,t \in S$, $x \in M'_{\perp_s}$, and $y \in M'_{\perp_t}$.

Let $s,t \in S$, $x \in M'_{\perp_s}$, and $y \in M'_{\perp_t}$ such that $(\sigma(s),x) \sim_{M'} (\sigma(t),y)$. By the definition of $\sim_{M'}$ (defined in Section 7.2.1), there are two cases to consider:

**Case** $x = y = \perp$**:** As $(\sigma(s),x) \sim_{M'} (\sigma(t),y)$ holds, there exists sort $u' \in S'$ such that $\sigma(s) \leq u'$ and $\sigma(t) \leq u'$. By the *weak non-extension* property of $\sigma$ (see Section 4.4), we know there exists sort $v \in S$ such that $s \leq v$, $t \leq v$, and $\sigma(v) \leq u'$. Thus $(s,x) \sim_{M'|_\sigma} (t,y)$ holds.

**Case $x \neq \bot$ and $y \neq \bot$:** We need to show that the following two conditions hold:

1. $\exists u \in S$ such that $s \leq u$ and $t \leq u$.
2. $\forall u \in S$ such that $s \leq u$ and $t \leq u$ the following holds:

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

For Condition 1, the proof is identical to the proof in Case 1. Regarding Condition 2, as $(\sigma(s), x) \sim_{M'} (\sigma(t), y)$, we know that for all sorts $u' \in S'$ such that $\sigma(s) \leq u'$ and $\sigma(t) \leq u'$, we have $(\mathtt{inj}_{\sigma(s),u'})_{M'}(x) = (\mathtt{inj}_{\sigma(t),u'})_{M'}(y)$. Let $u \in S$ such that $s \leq u$ and $t \leq u$. By property *p1* of $\sigma$ (see Section 4.3), we know $\sigma(s) \leq \sigma(u)$ and $\sigma(t) \leq \sigma(u)$, thus we know $(\mathtt{inj}_{\sigma(s),\sigma(u)})_{M'}(x) = (\mathtt{inj}_{\sigma(t),\sigma(u)})_{M'}(y)$. By applying the definition of reduct (see Section 4.2.2), we get $(\mathtt{inj}_{s,u})_{M'|_\sigma}(x) = (\mathtt{inj}_{t,u})_{M'|_\sigma}(y)$. □

*Appendix B*

# CSP Domain Clauses

Here, we present the clauses for the CSP Traces semantics, Failures/Divergences semantics and Stable-Failures semantics as presented in [Ros98].

## B.1 Traces Semantics

Semantic clauses for the *traces* function used in the Traces and Stable-Failures semantics.

$$
\begin{aligned}
\mathit{traces}(\text{SKIP}) &:= \{\langle\rangle, \langle\checkmark\rangle\} \\
\mathit{traces}(\text{STOP}) &:= \{\langle\rangle\} \\
\mathit{traces}(\text{DIV}) &:= \{\langle\rangle\} \\
\mathit{traces}(a \rightarrow P) &:= \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{traces}(P)\} \\
\mathit{traces}(\sqcap x :: X \rightarrow P) &:= \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{traces}(P[a/x]), a \in X\} \\
\mathit{traces}(\Box x :: X \rightarrow P) &:= \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{traces}(P[a/x]), a \in X\} \\
\mathit{traces}(P \fatsemi Q) &:= (\mathit{traces}(P) \cap A^*) \\
&\quad \cup \{s \frown t \mid s \frown \langle\checkmark\rangle \in \mathit{traces}(P), t \in \mathit{traces}(Q)\} \\
\mathit{traces}(P \sqcap Q) &:= \mathit{traces}(P) \cup \mathit{traces}(Q) \\
\mathit{traces}(P \,\Box\, Q) &:= \mathit{traces}(P) \cup \mathit{traces}(Q) \\
\mathit{traces}(P \parallel Q) &:= \mathit{traces}(P) \cap \mathit{traces}(Q) \\
\mathit{traces}(P \,|||\, Q) &:= \bigcup\{s \,|||\, t \mid s \in \mathit{traces}(P) \wedge t \in \mathit{traces}(Q)\} \\
\mathit{traces}(P \,[\![ X ]\!]\, Q) &:= \bigcup\{s \,[\![ X ]\!]\, t \mid s \in \mathit{traces}(P) \wedge t \in \mathit{traces}(Q)\} \\
\mathit{traces}(P \,[\![ X \mid Y ]\!]\, Q) &:= \{s \in (X \cup Y)^{*\checkmark} \mid s \upharpoonright X \cup \{\checkmark\} \in \mathit{traces}(P) \wedge \\
&\qquad\qquad s \upharpoonright Y \cup \{\checkmark\} \in \mathit{traces}(Q)\} \\
\mathit{traces}(P \setminus X) &:= \{s \setminus X \mid s \in \mathit{traces}(P)\} \\
\mathit{traces}(P[R]) &:= \{t \mid \exists s \in \mathit{traces}(P) \bullet s \; R^* \; t\}
\end{aligned}
$$

$$traces(\text{if } \varphi \text{ then } P \text{ else } Q) := \begin{cases} traces(P) & \text{if } \varphi \text{ evaluates to } true \\ traces(Q) & \text{if } \varphi \text{ evaluates to } false \end{cases}$$

Clauses for operations on individual traces:

1. $\forall s, t \in A^*, a, b \in A$

$$\begin{aligned} \langle\rangle \,|||\, s &= \{s\} \\ s \,|||\, \langle\rangle &= \{s\} \\ \langle a\rangle \frown s \,|||\, \langle b\rangle \frown t &= \{\langle a\rangle \frown u \mid u \in s \,|||\, \langle b\rangle \frown t\} \\ &\cup \{\langle b\rangle \frown u \mid u \in \langle a\rangle \frown s \,|||\, t\} \\ s \,|||\, t \frown \langle\checkmark\rangle &= \{\} \\ s \frown \langle\checkmark\rangle \,|||\, t &= \{\} \\ s \frown \langle\checkmark\rangle \,|||\, t \frown \langle\checkmark\rangle &= \{u \frown \langle\checkmark\rangle \mid u \in s \,|||\, t\} \end{aligned}$$

2. $\forall s, t \in A^*, x, x' \in X, y, y' \in A \setminus X$ :

$$\begin{aligned} s \,|[\,X\,]|\, t \frown \langle\checkmark\rangle &= \{\} \\ s \frown \langle\checkmark\rangle \,|[\,X\,]|\, t &= \{\} \\ s \frown \langle\checkmark\rangle \,|[\,X\,]|\, t \frown \langle\checkmark\rangle &= \{u \frown \langle\checkmark\rangle \mid u \in s \,|[\,X\,]|\, t\} \\ s \,|[\,X\,]|\, t &= t \,|[\,X\,]|\, s \\ \langle\rangle \,|[\,X\,]|\, \langle\rangle &= \{\langle\rangle\} \\ \langle\rangle \,|[\,X\,]|\, \langle x\rangle &= \{\} \\ \langle\rangle \,|[\,X\,]|\, \langle y\rangle &= \{\langle y\rangle\} \\ \langle x\rangle \frown s \,|[\,X\,]|\, \langle y\rangle \frown t &= \{\langle y\rangle \frown u \mid u \in \langle x\rangle \frown s \,|[\,X\,]|\, t\} \\ \langle x\rangle \frown s \,|[\,X\,]|\, \langle x\rangle \frown t &= \{\langle x\rangle \frown u \mid u \in s \,|[\,X\,]|\, t\} \\ \langle x\rangle \frown s \,|[\,X\,]|\, \langle x'\rangle \frown t &= \{\} \quad \text{if } x \neq x' \\ \langle y\rangle \frown s \,|[\,X\,]|\, \langle y'\rangle \frown t &= \{\langle y\rangle \frown u \mid u \in s \,|[\,X\,]|\, \langle y'\rangle \frown t\} \\ &\cup \{\langle y'\rangle \frown u \mid u \in \langle y\rangle \frown s \,|[\,X\,]|\, t\} \end{aligned}$$

3. $s \setminus X$ is defined to be $s \upharpoonright (A \setminus X)$ for any trace $s$.

4. If $s \in A^*$ and $X \subseteq A$ then $s \upharpoonright A$ means the sequence $s$ *restricted* to $X$: the sequence whose members are those of $s$ which are in $X$.
   $\langle\rangle \upharpoonright X = \langle\rangle$ and
   $(s \frown \langle a\rangle) \upharpoonright X = (s \upharpoonright X) \frown \langle a\rangle$ if $a \in X$, $s \upharpoonright X$ otherwise.

5. Definition of the Relation $s\ R^*\ t$:
   $\langle a_1, \ldots a_n\rangle\ R^*\ \langle b_1, \ldots b_m\rangle \quad \Leftrightarrow \quad n = m \wedge \forall i \leq n \bullet a_i\ R\ b_i$

## B.2 Failures/Divergences Semantics

Semantic clauses for *divergences* function used in the Failures/Divergences semantics.

$$
\begin{aligned}
\mathit{divergences}(\text{SKIP}) &:= \emptyset \\
\mathit{divergences}(\text{STOP}) &:= \emptyset \\
\mathit{divergences}(\text{DIV}) &:= A^{*\checkmark} \\
\mathit{divergences}(a \rightarrow P) &:= \{\langle a\rangle \frown s \mid s \in \mathit{divergences}(P)\} \\
\mathit{divergences}(\Box x :: X \rightarrow P) &:= \{\langle a\rangle \frown s \mid s \in \mathit{divergences}(P[a/x]) \wedge a \in X\} \\
\mathit{divergences}(P \fatsemi Q) &:= \mathit{divergences}(P) \cup \{s \frown t \mid s \frown \langle\checkmark\rangle \in tr_\perp(P) \\
&\qquad \wedge t \in \mathit{divergences}(Q)\} \\
\mathit{divergences}(P \sqcap Q) &:= \mathit{divergences}(P) \cup \mathit{divergences}(Q) \\
\mathit{divergences}(P \,\Box\, Q) &:= \mathit{divergences}(P) \cup \mathit{divergences}(Q) \\
\mathit{divergences}(P \,|[\, X \,]|\, Q) &:= \{u \frown v \mid \exists s \in tr_\perp(P), t \in tr_\perp(Q) \bullet \\
&\qquad u \in (s \,|[\, X \,]|\, t) \cap A^* \\
&\qquad \wedge (s \in \mathit{divergences}(P) \vee t \in \mathit{divergences}(Q))\} \\
\mathit{divergences}(P \setminus X) &:= \{(s \setminus X) \frown t \mid s \in \mathit{divergences}(P)\} \\
&\qquad \cup \{(u \setminus X) \frown t \mid u \in A^\omega \wedge (u \setminus X) \text{ is finite} \\
&\qquad \wedge \forall s < u \bullet s \in tr_\perp(P)\} \\
\mathit{divergences}(P[R]) &:= \{s' \frown t \mid \exists s \in \mathit{divergences}(P) \cap A^* \bullet s \; R^* \; s'\} \\
\mathit{divergences}(\text{if } \varphi \text{ then } P \text{ else } Q) &:= \begin{cases} \mathit{divergences}(P) & \text{if } \varphi \text{ evaluates to } true \\ \mathit{divergences}(Q) & \text{if } \varphi \text{ evaluates to } false \end{cases}
\end{aligned}
$$

Divergences of $\sqcap x :: X \rightarrow P$, $P \| Q$, $P ||| Q$, and $P \,|[\, X \mid Y \,]|\, Q$ can be calculated as standard in the literature from the above clauses.

Semantic clauses for the $\mathit{failures}_\perp$ function used in the Failures/Divergences semantics.

$$
\begin{aligned}
\mathit{failures}_\perp(\text{SKIP}) &:= \{(\langle\rangle, X) \mid X \subseteq A\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq A^\checkmark\} \\
\mathit{failures}_\perp(\text{STOP}) &:= \{(\langle\rangle, X) \mid X \subseteq A^\checkmark\} \\
\mathit{failures}_\perp(\text{DIV}) &:= A^{*\checkmark} \times \mathcal{P}(A^\checkmark) \\
\mathit{failures}_\perp(a \rightarrow P) &:= \{(\langle\rangle, X) \mid a \notin X\} \\
&\qquad \cup \{(\langle a\rangle \frown s, X) \mid (s, X) \in \mathit{failures}_\perp(P)\} \\
\mathit{failures}_\perp(\Box x :: X \rightarrow P) &:= \{(\langle\rangle, Y) \mid X \cap Y = \emptyset\} \\
&\qquad \cup \{(\langle a\rangle \frown s, Y) \mid (s, Y) \in \mathit{failures}_\perp(P[a/x]) \\
&\qquad \wedge a \in X\} \\
\mathit{failures}_\perp(P \fatsemi Q) &:= \{(s, X) \mid s \in A^* \wedge (s, X \cup \{\checkmark\}) \in \mathit{failures}_\perp(P)\} \\
&\qquad \cup \{(s \frown t, X) \mid s \frown \langle\checkmark\rangle \in tr_\perp(P) \\
&\qquad \wedge (t, X) \in \mathit{failures}_\perp(Q)\} \\
&\qquad \cup \{(s, X) \mid s \in \mathit{divergences}(P \fatsemi Q)\} \\
\mathit{failures}_\perp(P \sqcap Q) &:= \mathit{failures}_\perp(P) \cup \mathit{failures}_\perp(Q)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{failures}_\perp(P \,\Box\, Q) := {} & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}_\perp(P) \cap \mathit{failures}_\perp(Q)\} \\
& \cup \{(s, X) \mid (s, X) \in \mathit{failures}_\perp(P) \cup \mathit{failures}_\perp(Q) \\
& \wedge\, s \neq \langle\rangle\} \\
& \cup \{(\langle\rangle, X) \mid \langle\rangle \in \mathit{divergences}(P) \\
& \cup\, \mathit{divergences}(Q))\} \\
& \cup \{(\langle\rangle, X) \mid X \subseteq A \wedge \langle\checkmark\rangle \in tr_\perp(P) \cup tr_\perp(Q)\} \\
\mathit{failures}_\perp(P \,|[\, X \,]|\, Q) := {} & \{(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
& \wedge\, \exists s, t \bullet (s, Y) \in \mathit{failures}_\perp(P) \wedge (t, Z) \in \mathit{failures}_\perp(Q) \\
& \wedge\, u \in s \,|[\, X \,]|\, t\} \\
& \cup \{(u, Y) \mid u \in \mathit{divergences}(P \,|[\, X \,]|\, Q)\} \\
\mathit{failures}_\perp(P \setminus X) := {} & \{(s \setminus X, Y) \mid (s, X \cup Y) \in \mathit{failures}_\perp(P)\} \\
& \cup \{(s, Y) \mid s \in \mathit{divergences}(P \setminus X)\} \\
\mathit{failures}_\perp(P[R]) := {} & \{(s', X) \mid \exists s \bullet s\ R^*\ s' \\
& \wedge\, (s, R^{-1}(X)) \in \mathit{failures}_\perp(P)\} \\
& \cup \{(s, X) \mid s \in \mathit{divergences}(P[R])\} \\
& \text{where } R^{-1}(X) = \{a \mid \exists a' \in X \bullet (a, a') \in X\}
\end{aligned}
$$

$$
\mathit{failures}_\perp(\text{if } \varphi \text{ then } P \text{ else } Q) := \begin{cases} \mathit{failures}_\perp(P) & \text{if } \varphi \text{ evaluates to } true \\ \mathit{failures}_\perp(Q) & \text{if } \varphi \text{ evaluates to } false \end{cases}
$$

$\mathit{failures}_\perp$ of $\sqcap x :: X \to P$, $P \parallel Q$, $P \,|||\, Q$, and $P \,|[\, X \mid Y \,]|\, Q$ can be calculated as standard in the literature from the above clauses.

## B.3 Stable-Failures Semantics

Semantic clauses for the *failures* function used in the Stable-Failures semantics.

$$
\begin{aligned}
\mathit{failures}(\text{SKIP}) := {} & \{(\langle\rangle, X) \mid X \subseteq A\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq A^\checkmark\} \\
\mathit{failures}(\text{STOP}) := {} & \{(\langle\rangle, X) \mid X \subseteq A^\checkmark\} \\
\mathit{failures}(\text{DIV}) := {} & \emptyset \\
\mathit{failures}(a \to P) := {} & \{(\langle\rangle, X) \mid a \notin X\} \\
& \cup \{(\langle a \rangle^\frown s, X) \mid (s, X) \in \mathit{failures}(P)\} \\
\mathit{failures}(\Box x :: X \to P) := {} & \{(\langle\rangle, Y) \mid X \cap Y = \emptyset\} \\
& \cup \{(\langle a \rangle^\frown s, Y) \mid (s, Y) \in \mathit{failures}(P[a/x]) \wedge a \in X\} \\
\mathit{failures}(P \fatsemi Q) := {} & \{(s, X) \mid s \in A^* \wedge (s, X \cup \{\checkmark\}) \in \mathit{failures}(P)\} \\
& \cup \{(s^\frown t, X) \mid s^\frown\langle\checkmark\rangle \in \mathit{traces}(P) \\
& \wedge\, (t, X) \in \mathit{failures}(Q)\} \\
\mathit{failures}(P \sqcap Q) := {} & \mathit{failures}(P) \cup \mathit{failures}(Q) \\
\mathit{failures}(P \,\Box\, Q) := {} & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathit{failures}(P) \cap \mathit{failures}(Q)\} \\
& \cup \{(s, X) \mid (s, X) \in \mathit{failures}(P) \cup \mathit{failures}(Q) \\
& \wedge\, s \neq \langle\rangle\} \\
& \cup \{(\langle\rangle, X) \mid X \subseteq A \wedge \langle\checkmark\rangle \in \mathit{traces}(P) \cup \mathit{traces}(Q)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{failures}(P \,|[\, X \,]|\, Q) := \{&(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}) \\
&\wedge \exists s, t \bullet (s, Y) \in \mathit{failures}(P) \wedge (t, Z) \in \mathit{failures}(Q) \\
&\wedge u \in s \,|[\, X \,]|\, t\} \\
\mathit{failures}(P \setminus X) := \{&(s \setminus X, Y) \mid (s, X \cup Y) \in \mathit{failures}(P)\} \\
\mathit{failures}(P[R]) := \{&(s', X) \mid \exists s \bullet s \; R^* \; s' \\
&\wedge (s, R^{-1}(X)) \in \mathit{failures}(P)\} \\
&\text{where } R^{-1}(X) = \{a \mid \exists a' \in X \bullet (a, a') \in X\}
\end{aligned}
$$

$$
\mathit{failures}(\text{if } \varphi \text{ then } P \text{ else } Q) := \begin{cases} \mathit{failures}(P) & \text{if } \varphi \text{ evaluates to } true \\ \mathit{failures}(Q) & \text{if } \varphi \text{ evaluates to } false \end{cases}
$$

Failures of $\sqcap x :: X \to P$, $P \,||\, Q$, $P \,|||\, Q$, and $P \,|[\, X \mid Y \,]|\, Q$ can be calculated as standard in the literature from the above clauses.

*Appendix C*

# ATM Full Specifications

This appendix contains full specification of the ATM system used as an example in Section 8.4.1.

## C.1 Specifications of ATM with Shrinking Alphabet

**library** ATM_SYSTEM_SHRINKING

**from** BASIC/NUMBERS **get** INT

**logic** CASL

**spec** ARCH_COMMONDATA =
    **sorts** *Number, Decision*
**end**

**logic** CSPCASL

**spec** ARCH_USER =
    **data** ARCH_COMMONDATA
    **process** *User* : *Number, Decision*
**end**

**spec** ARCH_ATM =
    **data** ARCH_COMMONDATA
    **process** *ATM* : *Number, Decision*
**end**

**spec** ARCH_SYSTEM = ARCH_USER **and** ARCH_ATM **then**
    **process** *System* : *Number, Decision*;

```
        System = User || ATM
end

logic CASL

spec ACL_COMMONDATA = INT then
    sort Decision
end

logic CSPCASL

spec ACL_USER =
    data ACL_COMMONDATA
    process User : Int, Nat, Pos, Decision;
        User = (⊓ x :: Int → User) ⊓ (□ y :: Decision → User)
end

spec ACL_ATM =
    data ACL_COMMONDATA
    process ATM : Int, Nat, Pos, Decision;
        ATM = (□ x :: Int → ATM) ⊓ (⊓ y :: Decision → ATM)
end

spec ACL_SYSTEM = ACL_USER and ACL_ATM then
    process System : Int, Nat, Pos, Decision;
        System = User || ATM
end

logic CASL

spec CCL_COMMONDATA = INT then
    free type Decision ::= allowed | refused
end

logic CSPCASL

spec CCL_USER =
    data CCL_COMMONDATA
    process User : Pos, Decision;
        User = ⊓ amount :: Pos → □ decision :: Decision → User
end

spec CCL_ATM =
    data CCL_COMMONDATA then op    startingBalance : Int
```

```
    process ATM : Int, Nat, Pos, Decision;
            ATMAux(Int) : Int, Nat, Pos, Decision;
            ATM = ATMAux (startingBalance);
            ATMAux(balance) = □ amount :: Int → ⊓ decision :: Decision →
                              if decision = allowed
                              then ATMAux (startingBalance − amount)
                              else ATMAux (startingBalance)
end

spec CCL_SYSTEM = CCL_USER and CCL_ATM then
    process System : Int, Nat, Pos, Decision;
            System = User || ATM
end

logic CASL

view ARCH2ACL_COMMONDATA : ARCH_COMMONDATA to ACL_COMMONDATA =
    Number ↦ Int
end

view ACL2CCL_COMMONDATA : ACL_COMMONDATA to CCL_COMMONDATA

logic CSPCASL

view ARCH2ACL_USER : ARCH_USER to ACL_USER =
    Number ↦ Int
end

view ACL2CCL_USER : ACL_USER to CCL_USER =
    User : {Int, Nat, Pos, Decision} ↦ User : {Pos,Decision}
end

view ARCH2ACL_ATM : ARCH_ATM to ACL_ATM =
    Number ↦ Int
end

view ACL2CCL_ATM : ACL_ATM to CCL_ATM =
    ATM : {Int, Nat, Pos, Decision} ↦ ATM : {Int, Nat, Pos, Decision}
end

view ARCH2ACL_SYSTEM : ARCH_SYSTEM to ACL_SYSTEM =
    Number ↦ Int
end
```

**view** ACL2CCL_SYSTEM : ACL_SYSTEM **to** CCL_SYSTEM =
*User* : {*Int*, *Nat*, *Pos*, *Decision*} ↦ *User* : {*Pos*, *Decision*},
*ATM* : {*Int*, *Nat*, *Pos*, *Decision*} ↦ *ATM* : {*Int*, *Nat*, *Pos*, *Decision*},
*System* : {*Int*, *Nat*, *Pos*, *Decision*} ↦ *System* : {*Int*, *Nat*, *Pos*, *Decision*}
**end**

## C.2 Specifications of ATM without Shrinking Alphabet

Here, mainly the architectural level is changed. We show only the changed specifications and simply note that the other specifications are the same as the shrinking alphabet case (Appendix C.1) but with the imports being the alternative version.

**library** ALTERNATIVE_ATM_SYSTEM

**from** BASIC/NUMBERS **get** INT

**logic** CASL

**spec** ALT_ARCH_COMMONDATA =
**sort** *Decision*
**sort** *UserNumber* < *ATMNumber*
**end**

**logic** CSPCASL

**spec** ALT_ARCH_USER =
**data** ALT_ARCH_COMMONDATA
**process** *User* : *UserNumber*, *Decision*
**end**

**spec** ALT_ARCH_ATM =
**data** ALT_ARCH_COMMONDATA
**process** *ATM* : *ATMNumber*, *Decision*
**end**

**spec** ALT_ARCH_SYSTEM = ALT_ARCH_USER **and** ALT_ARCH_ATM **then**
**process** *System* : *ATMNumber*, *UserNumber*, *Decision*;
*System* = *User* ‖ *ATM*
**end**

**spec** ALT_ACL_USER =
**data** ALT_ACL_COMMONDATA
**process** *User* : *Pos*, *Decision*;
*User* = (⊓ *x* :: *Pos* → *User*) ⊓ (□ *y* :: *Decision* → *User*)

**end**
**logic** CASL

**view** ALT_ARCH2ACL_COMMONDATA :
ALT_ARCH_COMMONDATA **to** ALT_ACL_COMMONDATA =
*UserNumber* $\mapsto$ *Pos*, *ATMNumber* $\mapsto$ *Int*
**end**

**view** ALT_ACL2CCL_COMMONDATA :
ALT_ACL_COMMONDATA **to** ALT_CCL_COMMONDATA
**end**

**logic** CSPCASL

**view** ALT_ARCH2ACL_USER : ALT_ARCH_USER **to** ALT_ACL_USER =
*UserNumber* $\mapsto$ *Pos*, *ATMNumber* $\mapsto$ *Int*
**end**

**view** ALT_ACL2CCL_USER : ALT_ACL_USER **to** ALT_CCL_USER
**end**

**view** ALT_ARCH2ACL_ATM : ALT_ARCH_ATM **to** ALT_ACL_ATM =
*UserNumber* $\mapsto$ *Pos*, *ATMNumber* $\mapsto$ *Int*
**end**

**view** ALT_ACL2CCL_ATM : ALT_ACL_ATM **to** ALT_CCL_ATM

**view** ALT_ARCH2ACL_SYSTEM : ALT_ARCH_SYSTEM **to** ALT_ACL_SYSTEM =
*UserNumber* $\mapsto$ *Pos*, *ATMNumber* $\mapsto$ *Int*,
*System* : $\{$*ATMNumber*, *UserNumber*, *Decision*$\}$ $\mapsto$ *System* : $\{$*Int*, *Nat*, *Pos*, *Decision*$\}$
**end**

**view** ALT_ACL2CCL_SYSTEM : ALT_ACL_SYSTEM **to** ALT_CCL_SYSTEM

*Appendix D*

# Online Shop Full Specifications

This appendix contains full specification of the online shopping system presented in Chapter 10.

## D.1 Generic Shop Specification and Instantiations

**library** SHOP

**from** ARCH_COMPONENTS **get** ARCH_CUSTOMER, ARCH_WAREHOUSE,
ARCH_PAYMENTSYSTEM, ARCH_COORDINATOR

**from** ACL_COMPONENTS **get** ACL_CUSTOMER, ACL_WAREHOUSE,
ACL_PAYMENTSYSTEM, ACL_COORDINATOR

**from** CCL_COMPONENTS **get** CCL_CUSTOMER, CCL_WAREHOUSE,
CCL_PAYMENTSYSTEM, CCL_COORDINATOR

**logic** CSPCASL
**spec** GENERIC_SHOP [*RefCl*(ARCH_CUSTOMER)] [*RefCl*(ARCH_WAREHOUSE)]
[*RefCl*(ARCH_PAYMENTSYSTEM)] [*RefCl*(ARCH_COORDINATOR)] =
**process** *System* : {*C_C*, *C_W*, *C_PS*};
*System* = *Coordinator* [ *C_C*, *C_W*, *C_PS* || *C_C*, *C_W*, *C_PS* ]
(*Customer* [ *C_C* || *C_W*, *C_PS* ]
(*Warehouse* [ *C_W* || *C_PS* ] *PaymentSystem*))
**end**

**spec** ARCH_SHOP = GENERIC_SHOP [ARCH_CUSTOMER] [ARCH_WAREHOUSE]
[ARCH_PAYMENTSYSTEM] [ARCH_COORDINATOR]
**end**

```
spec ACL_SHOP = GENERIC_SHOP [ACL_CUSTOMER] [ACL_WAREHOUSE]
                             [ACL_PAYMENTSYSTEM] [ACL_COORDINATOR]
end

spec CCL_SHOP = GENERIC_SHOP [CCL_CUSTOMER] [CCL_WAREHOUSE]
                             [CCL_PAYMENTSYSTEM] [CCL_COORDINATOR]
end

view ARCH2ACL : ARCH_SHOP to ACL_SHOP
end

view ACL2CCL : ACL_SHOP to CCL_SHOP
end
```

## D.2 Architectural Components

```
library ARCH_COMPONENTS

logic CASL

spec ARCH_COMMON_DATA =
    {}
end

spec ARCH_COMM_COORDINATOR_CUSTOMER_DATA =
    ARCH_COMMON_DATA
then sorts LoginReq, LogoutReq < D_C
end

spec ARCH_COMM_COORDINATOR_WAREHOUSE_DATA =
    ARCH_COMMON_DATA
then sort D_W
end

spec ARCH_COMM_COORDINATOR_PAYMENTSYSTEM_DATA =
    ARCH_COMMON_DATA
then sort D_PS
end

spec ARCH_CUSTOMER_DATA =
    ARCH_COMM_COORDINATOR_CUSTOMER_DATA
end
```

```
spec ARCH_WAREHOUSE_DATA =
    ARCH_COMM_COORDINATOR_WAREHOUSE_DATA
end

spec ARCH_PAYMENTSYSTEM_DATA =
    ARCH_COMM_COORDINATOR_PAYMENTSYSTEM_DATA
end

spec ARCH_COORDINATOR_DATA =
    ARCH_COMM_COORDINATOR_CUSTOMER_DATA
and ARCH_COMM_COORDINATOR_WAREHOUSE_DATA
and ARCH_COMM_COORDINATOR_PAYMENTSYSTEM_DATA
end

logic CSPCASL

spec ARCH_CUSTOMER =
    data ARCH_CUSTOMER_DATA
    channel C_C : D_C
    process Customer : C_C;
            Customer_SuccessfulLogin : C_C;
            Customer_FailedLogin : C_C;
            Customer_Body : C_C;
            Customer_ViewCatalogue : C_C;
            Customer_ViewBasket : C_C;
            Customer_AddItem : C_C;
            Customer_RemoveItem : C_C;
            Customer_Checkout : C_C;
            Customer_Logout : C_C;
            Customer_SuccessfulLogout : C_C;
            Customer_FailedLogout : C_C;
            Customer = C_C ! x :: LoginReq →
                       (Customer_SuccessfulLogin ; Customer_Body
                        □ Customer_FailedLogin ; Customer);
            Customer_Logout = C_C ! x :: LogoutReq →
                              (Customer_SuccessfulLogout ; Customer
                               □ Customer_FailedLogout ; Customer_Body);
            Customer_Body = Customer_ViewCatalogue ⊓ Customer_ViewBasket
                          ⊓ Customer_AddItem ⊓ Customer_RemoveItem
                          ⊓ Customer_Checkout ⊓ Customer_Logout
end

spec ARCH_WAREHOUSE =
    data ARCH_WAREHOUSE_DATA
```

```
    channel C_W : D_W
    process Warehouse : C_W;
            Warehouse_ReserveItem : C_W;
            Warehouse_ReleaseItem : C_W;
            Warehouse_Dispatch : C_W;
            Warehouse = Warehouse_ReserveItem □ Warehouse_ReleaseItem
                      □ Warehouse_Dispatch
end

spec ARCH_PAYMENTSYSTEM =
    data ARCH_PAYMENTSYSTEM_DATA
    channel C_PS : D_PS
    process PaymentSystem : C_PS
end

spec ARCH_COORDINATOR =
    data ARCH_COORDINATOR_DATA
    channels C_C : D_C;
             C_W : D_W;
             C_PS : D_PS
    process Coordinator : {C_C, C_W, C_PS};
            Coordinator_SuccessfulLogin : C_C;
            Coordinator_FailedLogin : C_C;
            Coordinator_Body : {C_C, C_W, C_PS};
            Coordinator_ViewCatalogue : {C_C, C_W, C_PS};
            Coordinator_ViewBasket : {C_C, C_W, C_PS};
            Coordinator_AddItem : {C_C, C_W, C_PS};
            Coordinator_RemoveItem : {C_C, C_W, C_PS};
            Coordinator_Checkout : {C_C, C_W, C_PS};
            Coordinator_Logout : {C_C, C_W, C_PS};
            Coordinator_SuccessfulLogout : C_C;
            Coordinator_FailedLogout : C_C;
            Coordinator = C_C ? x :: LoginReq →
                          (Coordinator_SuccessfulLogin ; Coordinator_Body
                           ⊓ Coordinator_FailedLogin ; Coordinator);
            Coordinator_Logout = C_C ? x :: LogoutReq →
                                 (Coordinator_SuccessfulLogout ; Coordinator
                                  ⊓ Coordinator_FailedLogout ; Coordinator_Body);
            Coordinator_Body = Coordinator_ViewCatalogue □ Coordinator_ViewBasket
                             □ Coordinator_AddItem □ Coordinator_RemoveItem
                             □ Coordinator_Checkout □ Coordinator_Logout
end
```

## D.3 Abstract Component Level Components

**library** ACL_COMPONENTS

**logic** CASL

**spec** ACL_COMMON_DATA =
{}
**end**

**spec** ACL_COMM_COORDINATOR_CUSTOMER_DATA =
ACL_COMMON_DATA
**then sorts** *LoginReq*, *SuccessfulLoginRes*, *FailedLoginRes*,
*ViewCatalogueReq*, *ViewCatalogueRes*,
*ViewBasketReq*, *ViewBasketRes*, *AddItemReq*,
*AddItemRes*, *RemoveItemReq*, *RemoveItemRes*,
*CheckoutReq*, *CheckoutRes*, *CheckoutConfirmReq*,
*CheckoutConfirmRes*, *LogoutReq*, *SuccessfulLogoutRes*,
*FailedLogoutRes*
**free type**
*D_C* ::= %% Requests
*sort LoginReq*
| *sort ViewCatalogueReq*
| *sort ViewBasketReq*
| *sort AddItemReq*
| *sort RemoveItemReq*
| *sort CheckoutReq*
| *sort CheckoutConfirmReq*
| *sort LogoutReq*
%% Responses
| *sort SuccessfulLoginRes*
| *sort FailedLoginRes*
| *sort ViewCatalogueRes*
| *sort ViewBasketRes*
| *sort AddItemRes*
| *sort RemoveItemRes*
| *sort CheckoutRes*
| *sort CheckoutConfirmRes*
| *sort SuccessfulLogoutRes*
| *sort FailedLogoutRes*
**end**

**spec** ACL_COMM_COORDINATOR_WAREHOUSE_DATA =
ACL_COMMON_DATA

```
then sorts ReserveItemReq, ReserveItemRes,
           ReleaseItemReq, ReleaseItemRes, DispatchReq,
           DispatchRes
     free type
     D_W ::= sort ReserveItemReq
           | sort ReserveItemRes
           | sort ReleaseItemReq
           | sort ReleaseItemRes
           | sort DispatchReq
           | sort DispatchRes
end

spec ACL_COMM_COORDINATOR_PAYMENTSYSTEM_DATA =
     ACL_COMMON_DATA
then sorts TakePaymentReq, TakePaymentRes
     free type
     D_PS ::= sort TakePaymentReq | sort TakePaymentRes
end

spec ACL_CUSTOMER_DATA =
     ACL_COMM_COORDINATOR_CUSTOMER_DATA
end

spec ACL_WAREHOUSE_DATA =
     ACL_COMM_COORDINATOR_WAREHOUSE_DATA
end

spec ACL_PAYMENTSYSTEM_DATA =
     ACL_COMM_COORDINATOR_PAYMENTSYSTEM_DATA
end

spec ACL_COORDINATOR_DATA =
     ACL_COMM_COORDINATOR_CUSTOMER_DATA
and  ACL_COMM_COORDINATOR_WAREHOUSE_DATA
and  ACL_COMM_COORDINATOR_PAYMENTSYSTEM_DATA
end

logic CSPCASL

spec ACL_CUSTOMER =
     data ACL_CUSTOMER_DATA
     channel C_C : D_C
     process Customer : C_C;
             Customer_SuccessfulLogin : C_C;
```

*Customer_FailedLogin* : *C_C*;
*Customer_Body* : *C_C*;
*Customer_ViewCatalogue* : *C_C*;
*Customer_ViewBasket* : *C_C*;
*Customer_AddItem* : *C_C*;
*Customer_RemoveItem* : *C_C*;
*Customer_Checkout* : *C_C*;
*Customer_Logout* : *C_C*;
*Customer_SuccessfulLogout* : *C_C*;
*Customer_FailedLogout* : *C_C*;
*Customer* = *C_C* ! *x* :: *LoginReq* →
(*Customer_SuccessfulLogin* ; *Customer_Body*
□ *Customer_FailedLogin* ; *Customer*);
*Customer_SuccessfulLogin* = *C_C* ? *x* :: *SuccessfulLoginRes* → *SKIP*;
*Customer_FailedLogin* = *C_C* ? *x* :: *FailedLoginRes* → *SKIP*;
*Customer_ViewCatalogue* = *C_C* ! *x* :: *ViewCatalogueReq def* →
*C_C* ? *y* :: *ViewCatalogueRes* → *Customer_Body*;
*Customer_ViewBasket* = *C_C* ! *x* :: *ViewBasketReq def* →
*C_C* ? *y* :: *ViewBasketRes* → *Customer_Body*;
*Customer_AddItem* = *C_C* ! *x* :: *AddItemReq def* →
*C_C* ? *y* :: *AddItemRes* → *Customer_Body*;
*Customer_RemoveItem* = *C_C* ! *x* :: *RemoveItemReq def* →
*C_C* ? *y* :: *RemoveItemRes* → *Customer_Body*;
*Customer_Checkout* = *C_C* ! *x* :: *CheckoutReq def* →
*C_C* ? *y* :: *CheckoutConfirmReq* →
*C_C* ! *x* :: *CheckoutConfirmRes* →
*C_C* ? *x* :: *CheckoutRes* → *Customer_Body*;
*Customer_Logout* = *C_C* ! *x* :: *LogoutReq* →
(*Customer_SuccessfulLogout* ; *Customer*
□ *Customer_FailedLogout* ; *Customer_Body*);
*Customer_SuccessfulLogout* = *C_C* ? *x* :: *SuccessfulLogoutRes* → *SKIP*;
*Customer_FailedLogout* = *C_C* ? *x* :: *FailedLogoutRes* → *SKIP*;
*Customer_Body* = *Customer_ViewCatalogue* ⊓ *Customer_ViewBasket*
⊓ *Customer_AddItem* ⊓ *Customer_RemoveItem*
⊓ *Customer_Checkout* ⊓ *Customer_Logout*

**end**

**spec** ACL_WAREHOUSE =
**data** ACL_WAREHOUSE_DATA
**channel** *C_W* : *D_W*
**process** *Warehouse* : *C_W*;
*Warehouse_ReserveItem* : *C_W*;
*Warehouse_ReleaseItem* : *C_W*;
*Warehouse_Dispatch* : *C_W*;

$Warehouse\_ReserveItem = C\_W\ ?\ x :: ReserveItemReq \rightarrow$
$C\_W\ !\ y :: ReserveItemRes \rightarrow Warehouse;$
$Warehouse\_ReleaseItem = C\_W\ ?\ x :: ReleaseItemReq \rightarrow$
$C\_W\ !\ y :: ReleaseItemRes \rightarrow Warehouse;$
$Warehouse\_Dispatch = C\_W\ ?\ x :: DispatchReq \rightarrow$
$C\_W\ !\ y :: DispatchRes \rightarrow Warehouse;$
$Warehouse = Warehouse\_ReserveItem \Box Warehouse\_ReleaseItem$
$\Box\ Warehouse\_Dispatch$

**end**

**spec** ACL_PAYMENTSYSTEM =
**data** ACL_PAYMENTSYSTEM_DATA
**channel** $C\_PS : D\_PS$
**process** $PaymentSystem : C\_PS;$
$PaymentSystem = C\_PS\ ?\ x :: TakePaymentReq \rightarrow$
$C\_PS\ !\ y :: TakePaymentRes \rightarrow PaymentSystem$
**end**

**spec** ACL_COORDINATOR =
**data** ACL_COORDINATOR_DATA
**channels** $C\_C : D\_C;$
$C\_W : D\_W;$
$C\_PS : D\_PS$
**process** $Coordinator : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_SuccessfulLogin : C\_C;$
$Coordinator\_FailedLogin : C\_C;$
$Coordinator\_Body : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_ViewCatalogue : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_ViewBasket : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_AddItem : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_RemoveItem : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_Checkout : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_Checkout\_Cancel : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_Checkout\_Confirm : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_Checkout\_PaymentFailed : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_Checkout\_PaymentSucessful : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_Logout : \{C\_C, C\_W, C\_PS\};$
$Coordinator\_SuccessfulLogout : C\_C;$
$Coordinator\_FailedLogout : C\_C;$
$Coordinator = C\_C\ ?\ x :: LoginReq \rightarrow$
$(Coordinator\_SuccessfulLogin\ ;\ Coordinator\_Body$
$\sqcap\ Coordinator\_FailedLogin\ ;\ Coordinator);$
$Coordinator\_SuccessfulLogin = C\_C\ !\ x :: SuccessfulLoginRes\ def \rightarrow SKIP;$
$Coordinator\_FailedLogin = C\_C\ !\ x :: FailedLoginRes\ def \rightarrow SKIP;$

$Coordinator\_ViewCatalogue = C\_C\ ?\ x :: ViewCatalogueReq \rightarrow$
$C\_C\ !\ y :: ViewCatalogueRes \rightarrow$
$Coordinator\_Body;$
$Coordinator\_ViewBasket = C\_C\ ?\ x :: ViewBasketReq \rightarrow$
$C\_C\ !\ y :: ViewBasketRes \rightarrow Coordinator\_Body;$
$Coordinator\_AddItem = C\_C\ ?\ x :: AddItemReq \rightarrow$
$C\_W\ !\ y :: ReserveItemReq\ def \rightarrow$
$C\_W\ ?\ x :: ReserveItemRes \rightarrow$
$C\_C\ !\ y :: AddItemRes \rightarrow Coordinator\_Body;$

%% Branch 1: Item can be removed
%% Branch 2: Item cannot be removed
$Coordinator\_RemoveItem = C\_C\ ?\ x :: RemoveItemReq \rightarrow$
$(C\_W\ !\ x :: ReleaseItemReq\ def \rightarrow$
$C\_W\ ?\ y :: ReleaseItemRes \rightarrow$
$C\_C\ !\ x :: RemoveItemRes \rightarrow Coordinator\_Body$
$\sqcap\ C\_C\ !\ x :: RemoveItemRes \rightarrow$
$Coordinator\_Body);$
$Coordinator\_Checkout = C\_C\ ?\ x :: CheckoutReq \rightarrow$
$C\_C\ !\ y :: CheckoutConfirmReq \rightarrow$
$C\_C\ ?\ x :: CheckoutConfirmRes \rightarrow$
$(Coordinator\_Checkout\_Cancel$
$\sqcap\ Coordinator\_Checkout\_Confirm);$
$Coordinator\_Checkout\_Cancel = C\_C\ !\ x :: CheckoutRes \rightarrow Coordinator\_Body;$
$Coordinator\_Checkout\_Confirm = C\_PS\ !\ x :: TakePaymentReq \rightarrow$
$C\_PS\ ?\ y :: TakePaymentRes \rightarrow$
$(Coordinator\_Checkout\_PaymentFailed$
$\sqcap\ Coordinator\_Checkout\_PaymentSucessful);$
$Coordinator\_Checkout\_PaymentFailed = C\_C\ !\ x :: CheckoutRes \rightarrow$
$Coordinator\_Body;$
$Coordinator\_Checkout\_PaymentSucessful = C\_W\ !\ x :: DispatchReq\ def \rightarrow$
$C\_W\ ?\ y :: DispatchRes \rightarrow$
$C\_C\ !\ x :: CheckoutRes \rightarrow$
$Coordinator\_Body;$
$Coordinator\_Logout = C\_C\ ?\ x :: LogoutReq \rightarrow$
$(Coordinator\_SuccessfulLogout\ ;\ Coordinator$
$\sqcap\ Coordinator\_FailedLogout\ ;\ Coordinator\_Body);$
$Coordinator\_SuccessfulLogout = C\_C\ !\ x :: SuccessfulLogoutRes\ def \rightarrow SKIP;$
$Coordinator\_FailedLogout = C\_C\ !\ x :: FailedLogoutRes\ def \rightarrow SKIP;$
$Coordinator\_Body = Coordinator\_ViewCatalogue \Box Coordinator\_ViewBasket$
$\Box\ Coordinator\_AddItem \Box Coordinator\_RemoveItem$
$\Box\ Coordinator\_Checkout \Box Coordinator\_Logout$

**end**

# Bibliography


[ABR99] Egidio Astesiano, Manfred Broy, and Gianna Reggio. Algebraic specification of concurrent systems. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*. Springer, 1999.

[ACM10] Jean-Raymond Abrial, Dominique Cansell, and Christophe Metayer. Specification of the automatic prover P3. In Jens Bendispoto, Michael Leuschel, and Markus Roggenbach, editors, *AVoCS'10 – Proceedings of the Tenth International Workshop on Automated Verification of Critical Systems*. Heinrich-Heine-Universität Düsseldorf, 2010.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6:213–249, July 1997.

[AG09] M. Alexander and W. Gardner. *Process algebra for parallel and distributed processing*. Chapman & Hall/CRC computational science series. CRC Press, 2009.

[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 *Lecture Notes in Computer Science*. Springer, 2005.

[Bau99] Hubert Baumeister. *Relations between Abstract Datatypes modeled as Abstract Datatypes*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1999.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language Lotos. *Computer Networks*, 14(1):25–59, January 1988.

[BCH99] Michel Bidoit, Maria Victoria Cengarle, and Rolf Hennicker. Proof systems for structured specifications and their refinements. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of System Specification*. Springer, 1999.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. *Algebraic specification*, 1989.

[BHK90] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *J. ACM*, 37:335–372, April 1990.

[Bjø00] Dines Bjørner. Formal Software Techniques for Railway Systems. *CTS2000: 9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, 2000.

[Bjø09] Dines Bjørner. *DOMAIN ENGINEERING Technology Management, Research and Engineering*. Japan Advanced Institute of Science and Technology, 2009.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual: Introduction to Using the Common Algebraic Specification Language*. LNCS 2900. Springer, 2004.

[Boe88] Barry William Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[BS93] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. Lotos Specifications, their Implementations, and their Tests. In *Protocol Specification, Testing and Verification*, pages 349–360. Elsevier, 1987.

[CM97] M. Cerioli and J. Meseguer. May I borrow your logic? (Transporting logical structures along maps). *Theoretical Computer Science*, 173:311–347, 1997.

[CRS+ar] Antonio Cerone, Markus Roggenbach, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh. *Formal Methods for Software Engineering: Languages, Methods, Application Domains*. Springer, to appear.

[DGS93] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In *Logical Environments*, pages 83–130. Cambridge, 1993.

[EBK+02] Egidio, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The Common Algebraic Specification Language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer, 1985.

[EP208] EP2 Consortium. EFT/POS 2000 specification, version 4.0.0, 2008. Project Overview available at `http://www.eftpos2000.ch`.

[FDR06] *Failures-Divergence Refinement – the FDR2 User Manual.* Formal Systems (Europe) Ltd., 2006.

[Fia05] José Luiz Fiadeiro. *Categories for Software Engineering.* Springer, 2005.

[Fis97] Clemens Fischer. Combining OBJECT-Z and CSP. In Adam Wolisz, Ina Schieferdecker, and Axel Rennoch, editors, *FBT*, volume 315 of *GMD-Studien.* GMD-Forschungszentrum Informationstechnik GmbH, 1997.

[Fow09] Kim Fowler. *Mission-Critical and Safety-Critical Systems Handbook: Design and Development for Embedded Applications.* Newnes, 2009.

[Gar96] Hubert Garavel. An Overview of the Eucalyptus Toolbox. In *COST 247*, pages 76–88. University of Maribor, 1996.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[Gim08] Andy Gimblett. Tool Support for CSP-CASL. MPhil Thesis, Swansea University, 2008.

[GKOR09] Andy Gimblett, Temesghen Kahsai, Liam O'Reilly, and Markus Roggenbach. On the whereabouts of CSP-CASL – A Survey. In Bernd Gersdorf, Berthold Hoffmann, Christoph Lüth, Till Mossakowski, Sylvie Rauer, Markus Roggenbach Thomas Röfer, Lutz Schröder, and Mattias Werner Shi Hui, editors, *Specification, Transformation, Navigation – Festschrift dedicated to Bernd Krieg-Brückner*, pages 121–139, 2009.

[GP95] J. F. Grote and A. Ponse. The syntax and semantics of $\mu$CRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing. Springer, 1995.

[HHJW07] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A History of Haskell: Being Lazy With Class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hoa06] Tony Hoare. Why ever csp? *Electr. Notes Theor. Comput. Sci.*, 162:209–215, 2006.

[IR] Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.

[IR05] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. 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, pages 158–172. Springer, 2006.

[ISO89] ISO 8807. Lotos – a formal description technique based on the temporal ordering of observational behaviour, 1989.

[Jon03] S.P. Jones. *Haskell 98 language and libraries: the revised report.* Cambridge Univ Pr, 2003.

[JOW06] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified Software: A Grand Challenge. *Computer*, 39(4):93–95, 2006.

[JR11] Phillip James and Markus Roggenbach. Designing domain specific languages for verification: First steps. In Georg Struth Peter Hofner, Annabelle McIver, editor, *ATE-2011 – Proceedings of the First Workshop on Automated Theory Engineering*, volume 760 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[JTC01] JTCI/CS7/WG14. *The E-LOTOS Final Draft International Standard*, 2001.

[Kah87] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[Kah10] Temesghen Kahsai. *Property Preserving Development and Testing for* CSP-CASL. PhD thesis, Swansea University, 2010.

[KR09] Temesghen Kahsai and Markus Roggenbach. Property preserving refinement for CSP-CASL. In *WADT 2008*, LNCS 5486. Springer, 2009.

[KRS08] Temesghen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff. Specification-based testing for software product lines. In Antonio Cerone and Stefan Gruner, editors, *SEFM 2008*, pages 149–159. IEEE Computer Society, 2008.

[Lis] Bert Lisser. $\mu$-CRL homepage. `http://www.cwi.nl/~mcrl`.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[ML98] Saunders Mac Lane. *Categories for the working mathematician.* Graduate texts in mathematics. Springer, 2nd edition, 1998.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, HETS. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2007.

[MOR11] Till Mossakowski, Liam O'Reilly, and Markus Roggenbach. Compositional reasoning for processes and data. In *Proceedings of the 18th Workshop on Automated Reasoning*, volume TR-2011-327 of *Technical Report*. Department of Computing Science, University of Glasgow, 2011.

[Mos97] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97*, LNCS 1214, pages 115–137. Springer, 1997.

[Mos98] T. Mossakowski. Colimits of order-sorted specifications. In *WADT*, LNCS 1376. Springer, 1998.

[Mos00] Till Mossakowski. Specifications in an arbitrary institution with symbols. In *WADT '99: Selected papers from the 14th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 252–270, London, UK, 2000. Springer-Verlag.

[Mos02] Till Mossakowski. Relating CASL with other specification languages: The institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[Mos04] Peter D. Mosses. CASL *Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*. 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. CO-CASL at work – Modelling Process Algebra. In *Coalgebraic Methods in Computer Science*, volume 82 of *Electronic Notes Theoretical Computer Science*, 2003.

[MSRR06] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-co-algebraic specification in CO-CASL. *Journal of Logic and Algebraic Programming*, 67(1-2):146–197, 2006. Extends (Mossakowski et al. 2003).

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[MU05] Petra Malik and Mark Utting. CZT: A framework for Z tools. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2005.

[MV90] S. Mauw and G. J. Veltink. A process specification formalism. *Fundam. Inf.*, 13(2):85–139, 1990.

[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, London, UK, 1992. Springer-Verlag.

[NPW02] T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. LNCS 2283. Springer-Verlag, London, UK, 2002.

[OGC08] M. V. M. Oliveira, A. C. Gurgel, and C. G. Castro. Crefine: Support for the CIRCUS refinement calculus. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 281–290. IEEE Computer Society, 2008.

[OMR12] Liam O'Reilly, Till Mossakowski, and Markus Roggenbach. Compositional modelling and reasoning in an institution for processes and data. In Till Mossakowski and Hans-Jörg Kreowski, editors, *WADT 2010*, volume 7137 of *Lecture Notes in Computer Science*, pages 251–269. Springer, 2012.

[O'R08] Liam O'Reilly. Developing proof technology for CSP-CASL. MPhil Thesis, Swansea University, 2008.

[ORI09] Liam O'Reilly, Markus Roggenbach, and Yoshinao Isobe. CSP-CASL-Prover: A generic tool for process and data refinement. *ENTCS*, 250(2):69–84, 2009.

[Pau94] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

[Pro03] *Process Behaviour Explorer – the ProBE User Manual*. Formal Systems (Europe) Ltd., 2003.

[PSF97] PSF toolkit manual pages, 1997. `http://www.wins.uva.nl/~bobd/work/`.

[RAC00] G. Reggio, E. Astesiano, and C. 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(1):42–71, 2006.

[Ros98] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[Ros05] A. W. Roscoe. *The theory and practice of concurrency*. 2005. Revised edition. Only available online.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[RR00] G. Reggio and L. 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.

[RSR04] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Form. Asp. Comput.*, 16(4):394–411, 2004.

[SAA01] Gwen Salaün, Michel Allemand, and Christian Attiogbé. A formalism combining CCS and CASL. Technical Report 00.14, University of Nantes, 2001.

[SAA02] 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.

[Sam08] D. Gift Samuel. Implementation of the Stable Revivals Model in CSP-CASL-Prover. MPhil Thesis, Swansea University, 2008.

[Sca98] Bryan Scattergood. The Semantics and Implementation of Machine-Readable CSP, 1998. DPhil thesis, University of Oxford.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The* CSP *Approach.* Wiley, 1999.

[Smi00] Graeme Smith. *The* OBJECT-Z *Specification Language.* Kluwer Academic Publishers, 2000.

[SMT+05] Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Piotr Hoffman, and Bartek Klin. Amalgamation in the semantics of CASL. *Theoretical Computer Science*, 331(1):215–247, 2005.

[Som07] Ian Sommerville. *Software Engineering.* Addison-Wesley, 8th edition, 2007.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.

[ST88] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[Sto96] Neil R. Storey. *Safety Critical Computer Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[Sze11] Wai Leung Sze. Evaluation of a domain specific language for modelling and verifying in the railway domain. BSc Dissertation, Swansea University, 2011.

[Ver99] Alberto Verdejo. E-LOTOS: Tutorial and semantics. Master's thesis, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 1999.

[vGW96] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

[VMO00] Alberto Verdejo and Narciso Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, February 2000.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer-Verlag, July 27–30 2002.

[WC01] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In Andrew Butterfield, Glenn Strong, and Claus Pahl, editors, *IWFM*, Workshops in Computing. BCS, 2001.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of CIRCUS. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.

[WD96] Jim Woodcock and Jim Davies. *Using Z – Specification, Refinement, and Proof.* Prentice Hall, 1996.

[Zaw04] Artur Zawlocki. Architectural specifications for reactive systems. In José Luiz Fiadeiro, Peter D. Mosses, and Fernando Orejas, editors, *WADT*, volume 3423 of *Lecture Notes in Computer Science*, pages 252–269. Springer, 2004.