Developing Proof Technology For CSP-CASL

Liam O'Reilly

A thesis submitted to the University of Wales in
candidature for the degree of Master of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

June 2008

Declaration

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

Signed ..... (candidate)

Date .....

Statement 1

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

Signed ..... (candidate)

Date .....

Statement 2

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

Signed ..... (candidate)

Date .....

Abstract

Distributed applications such as flight booking systems, web services, and electronic payment systems require parallel processing of data. Such systems exhibit concurrent aspects (e.g., deadlock freedom) as well as data aspects (e.g., functional correctness). Often, these two aspects depend on each other. The language CSP-CASL is tailored to the specification and verification of such distributed systems and allows one to model data as well as processes within a single framework.

In this thesis we explore methods and techniques tailored to theorem proving for CSP-CASL. This leads to the development of an architecture for CSP-CASL-Prover which re-uses the tools HETS and CSP-Prover. We also design – up to the algorithmic level – procedures for transforming a CSP-CASL specification into Isabelle/HOL code whilst preserving the semantics. By using this translation, it is possible to perform theorem proving on CSP-CASL specifications using Isabelle/HOL.

As proof of concept we validate our tool CSP-CASL-Prover on a case study of industrial strength. Our experiment shows that CSP-CASL-Prover scales up to large systems. When using CSP-CASL-Prover reasoning about CSP-CASL specifications becomes as easy as reasoning about data and processes separately.

Part of the results presented within this thesis have been published in [OIR07, OIR08].

Acknowledgements

I wish to thank my parents and family who have always believed in me. They gave me strength and supported me in so many ways without which this project would not have come true.

I would like to thank my supervisor Dr. Markus Roggenbach for his close and continuous guidance, support, and friendship throughout this work. I am very grateful to him for advising me that continuing my education after my degree was the best choice. I also thank my second supervisor, Dr. Monika Seisenberger for her kind support during this time.

I would also like to thank my examiners, Dr. Ulrich Berger and Prof. Holger Schlingloff for their valuable feedback and advice in polishing my thesis.

I would also like to extend my appreciation to my colleges Andy Gimblett, Temesghen Kahsai, and Gift Samuel within the *Processes and Data* group for their support and encouragement. Without them the learning process would have been much tougher and more annoying.

I thank Swansea University's Department of Computer Science as a whole for providing me with the opportunity to pursue this work. I would also like to thank all the staff and postgraduates who have made this experience very enjoyable and profitable. I thank Sitsofe Wheeler for his continuous effort in the battle of keeping my workstation up and running while I inevitably find bugs and try to destroy it.

Finally, I would like to thank Erwin R. Catesbeiana (jr) for watching over my shoulder and preventing me from procrastinating too much!

Table of Contents

List of Figures

Chapter 1

Introduction

CSP-CASL [Rog06] is a relativity young specification language tailored for describing distributed systems. Examples include electronic payment systems (EP2), flight booking systems, etc. Such system have both data and process aspects which often depend on one and other. Currently there exist limited tool support for this specification language.

For dependable system design it is essential in the first place to establish properties of specifications. Once a system has been specified in a formal specification language such as CSP-CASL, interesting properties can then be verified. Such properties include (but are not limited to) safety properties and liveness properties such as deadlock freedom. Verifying such properties on specifications of large distributed systems requires machine assisted tool support.

CSP-CASL integrates the process algebra CSP [Hoa85, Ros98] with the algebraic specification language CASL [Mos04]. Its novel aspects include the combination of denotational semantics in the process part and, in particular, loose semantics for the data types covering both concepts of partiality and sub-sorting. In [GRS05] CSP-CASL has been applied to the EP2 standard. It has been demonstrated that CSP-CASL can deal with problems of industrial strength.

Here, we develop theorem proving support for CSP-CASL and show that our approach scales up to practically relevant systems such as the EP2 standard. CSP-CASL comes with a simple, but powerful notion of refinement. CSP-CASL refinement can be decomposed into first a refinement step on data only and then a refinement step on processes. Data refinement is well understood in the CASL context and has good tool support already. Thus, we focus in this thesis on process refinement and tool support. The basic idea is to re-use existing tools for the languages CASL and CSP, namely for CASL the tool HETS [MML07] and for CSP the tool CSP-Prover [IR05, IR06], both of which are based on the theorem prover Isabelle/HOL [NPW02]. This re-use is possible thanks to the definition of the CSP-CASL semantics in a two step approach: First, the data specified in CASL is translated into an alphabet of communications, which, in the second step, is used within the processes, where the standard CSP semantics are applied.

The main issue in integrating the tools HETS and CSP-Prover into a CSP-CASL-Prover is to implement – in Isabelle/HOL – CSP-CASL's construction of an alphabet of communications out of an algebraic specification of data written in CASL. The correctness of this construction relies on the fact that a certain relation turns out to be an equivalence relation. Although this has been proven

to hold under certain conditions, we chose to prove this fact for each CSP-CASL specification individually. This adds an additional layer of trust. It turns out that the alphabet construction, the formulation of the justification theorems (establishing the equivalence relation), and also the proofs of these theorems can be automatically generated.

Closely related to CSP-CASL is the specification language μCRL [GP95]. Here, data types have loose semantics and are specified in equational logic with total functions. The underlying semantics of the process algebraic part is operational. [BFG$^+$05] presents - on the fly, as the focus of the paper is on protocol verification – the prototype of a μCRL-Prover based on the interactive theorem prover PVS [ORS92]. The chosen approach is to represent the abstract μCRL data types directly by PVS types, and to give a subset of μCRL processes, namely the linear process equations, an operational semantics in terms of labelled transition systems. Thanks to μCRL's simple approach to data – neither sub-sorting nor partiality are available – there is no need for an alphabet construction – as it is also the case in CSP-CASL in the absence of sub-sorting and partiality. Concerning processes, CSP-CASL provides semantics to full CSP by re-using the implementation of various denotational CSP semantics in CSP-Prover.

Thesis outline

In Chapter 2 we introduce the functional programming language Haskell along with the concepts of monadic programming within Haskell. We then present an own example of creating a monadic parser for a subset of CSP.

Chapter 3 introduces the notions of institutions and institutions representations. We present an own example of Many-Sorted Equational Logic as institution. Within this chapter we also present several other instances of institutions, namely $FOL^{=}$, $PFOL^{=}$ which is built on top of $FOL^{=}$ and finally $SubPFOL^{=}$ which is built upon $PFOL^{=}$. The institution $SubPFOL^{=}$ is very close to the CASL institution $SubPCFOL^{=}$. We then present the category of presentations, which are the basis for specifications. Institution representations are then introduced which allows one to relate different institutions. Finally we present some institution representations which allow us to translate specification written using the CASL institution into Isabelle/HOL.

Chapter 4 discusses all the existing tools that we re-use in the constructions within this thesis. The theorem prover Isabelle/HOL is introduced and discussed. This is a generic interactive theorem prover that has been shown to be very powerful. Isabelle/HOL plays a dominant role within this thesis. The tool HETS is introduced which implements both the institutions and institutions representations that are discussed in Chapter 3. This tool allows us to translate CASL specifications into Isabelle/HOL code. Finally the tool CSP-Prover is presented which is an extension to the theorem prover Isabelle/HOL. CSP-Prover is centred around refinement proofs for the process algebra CSP.

Chapter 5 introduces the language CSP-CASL followed by several examples of the modelling of systems in CSP-CASL. Such CSP-CASL specifications consist of a data part specified in CASL and a process part where communications are specified using data from the data part. The CSP-CASL semantics are then presented along with the CSP-CASL notions of refinement.

Chapter 6 begins by discussing our architecture for a CSP-CASL-Prover. Our algorithm for the tool CSP-CASL-Prover is then presented and discussed. The algorithm re-uses both the existing

tools HETS and CSP-Prover which are introduced in Chapter 4. Our algorithm takes as its input a CSP-CASL specification and encodes it in Isabelle/HOL whilst preserving its semantics. One can then easily carry out refinement proofs on processes where communications are specified using data from the data part (written using CASL). Our algorithm produces *Integrations Theorems* which are critical in reducing the complexity of proofs of such process refinements.

Chapter 7 presents proofs which have been produced using theory files derived using our algorithm which is discussed in Chapter 6. We present our proofs of four core examples for the language CSP-CASL along with a case study of industrial strength, namely EP2, which shows our technique for proving on processes and data scales up and can meet the challenges of the industrial world.

Chapter 2

Haskell

Contents

Haskell [HHJW07, Hug89] is a general purpose functional programming language which features lazy evaluation, higher order functions, polymorphism and type classes. Haskell is based on the lambda calculus.

Within this chapter we discuss what monads [Wad93] are and how to use them to build a small CSP parser that recognises processes within a restricted CSP grammar. Monads are used within the tool HETS [MML07] extensively and also within the library Parsec [LM01] which is a monadic-parser library for Haskell used for building parsers for grammars. A parser created using Parsec will be more efficient and easier to build for large grammars than the method presented here.

2.1 Monads

As Haskell is a purely functional programming language certain useful features cause problems, these features include side effects and state. Functional programming languages do not have state like imperative programming languages do, instead everything is done via function calls. A function must also not perform any side effects such as input and output. This is because a function must always return the same result for given arguments, hence reading from the keyboard would give you different results (characters) for the same input. These problems are dealt with in Haskell by using *monads*, which provide a clean solution that fits in well with the pure functional style.

The `IO` monad allows input and output within Haskell, without destroying the functional environment of the language. For example the type `IO()` denotes an input or output event which returns the type `()` (void), hence the type `IO()` can be seen as an action where some output has taken place (or some input has taken place, where the input has been discarded as the return type is void). `IO()` is an instance of the more general type `IO` α, which denotes an input/output action of type

α. A value of type `IO` α can be passed around within parameters of functions. When a value of this type is evaluated the action is performed. Hence monads allow us to sequence computations which represent actions. Only when the values are evaluated are the actions performed.

For instance there is a built-in function

```
putChar :: Char -> IO ()
```

that takes as a parameter a character and returns an action. When this action is evaluated the character is output to standard output.

Another built in-function is

```
getChar :: IO Char
```

which reads a character from standard input. The function `getChar` returns an action. When this action is evaluated, a character will be read from standard input and returned as the result of the function. The type of this function is `IO Char`. Hence, once we have read a value with this function we have a value of type `IO Char` not a `Char`. This prevents the data read from escaping into the functional part of the language. We cannot strip off the `IO` part of the type, however we can use the character part of a `IO Char` within the *do notation* and pass it as an argument to a function expecting a char. Again, the result must be wrapped backup in an IO action.

Haskell has a class called `Monad m` [Tho99] which is based on a polymorphic type constructor `m` which declare functions `return`, `(>>=)`, `(>>)` and `fail`.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>) :: m a -> m b -> m b
  fail :: String -> m a
```

These functions control the sequencing of monads when using the *do* notation within Haskell and makes explicit the order of execution. This makes it possible to mimic imperative programming.

The `return` function creates basic values of the monad type `m a` from a value of type `a`. The `>>=` function is called `bind`, it takes two parameters: a value of the monad type `m a` and a function from the underlying monad type to a new monad type, i.e., `a -> m b`. The bind function returns as the result a value of type `m b`, i.e., the return type of the second parameter.

To instantiate this class, one must provide definitions of these functions[1] that satisfy some properties such as `return x` should simply return the value of x, `>>=` should be associative and `fails s` should correspond to a computation which fails. For instance the `IO monad` is an instance of this class that provides certain definitions of these functions.

Other examples of monads which Haskell comes equipped with are the monads `List` α and `Maybe` α.

[1]The functions `>>` and `fail` have default definitions based on `>>=` and `return` so these need not be re-defined if the default definitions are sufficient.

2.2 Do notation

It can be awkward and hard to write understandable code if the bind (>>=) function is used heavily. Haskell has an alternative syntax which is available when using monads, called the `do notation`. This is an expressive syntactic sugar to build up computations using the >>= function.

The do notation allows one to write blocks of code that resemble imperative programming with variables. The result of a monadic computation can be assigned to a variable using the <− operator. The variable can then be used in later monadic computations. The value that is assigned to the variable is of the underlying monadic type.

The following small example demonstrates the *do notation*.

```
echo :: IO ()
echo = do c <- getChar
          putChar c
```

The function `echo` reads a character from standard input and outputs it to standard output. Its result type is `IO ()`, an action of type "void". The `getChar` monad gets a character from the keyboard. It returns a value of type `IO Char`, an action which result in character. This underlying character is assigned to the variable `c` (`c` has type `Char`, not `IO Char`). The monad `putChar` then outputs the character `c` to standard output. Its type is `Char -> IO ()`. Its return type matches the return type of `echo`. This is equivalent to the following code which does not use the *do notation* but instead uses the bind function explicitly.

```
echo :: IO ()
echo = getChar >>= putChar
```

2.3 A programming exercise - Developing a CSP parser

As an exercise in using monadic programming in Haskell we study here an own example of writing a parser for a subset of the process algebra CSP. This will help with the integration of tool support for CSP-CASL with HETS as the HETS framework uses monadic programming and the algorithm described in Chapter 6 requires the implementation to interface with the HETS system.

Using monads it is possible to create a parser. We use monads for this in order to easily pass around state and order the execution of function calls. The state that we are interested in includes all possible parses for the consumed input along with the input that has yet to be consumed for each parse. We follow the approach of building a parser by Richard Bird [Bir98] but adapt the code for the process algebra CSP. The material in Section 2.3.1 to Section 2.3.4 has been taken from [Bir98].

We wish to create a parser for the following CSP grammar:-

```
CSP = SKIP      |
      STOP      |
```

```
a -> P        |
(P [] Q)      |
(P |~| Q)     |
(P || Q)
```

Here actions (the `a` symbol above) are natural numbers. Although this grammar is a subset of normal CSP grammars, it is adequate for showing how a parser may be constructed using monads.

2.3.1 The parser monad

We need to create a new monad which will represent the type of parsers. As it will be a monad, this will take care of sequencing parsers together and the order of execution.

A parser can be thought of as a function that takes a string as input and returns a tree as output. However, there is no obvious way to sequence two parsers because the first parser will only return the tree (which represents what was parsed) and not the unconsumed input. Hence the second parser will not know where to start parsing from.

We refine our definition of a parser to a function that takes a string as input and returns the pair consisting of the unconsumed suffix of the input and the tree as output. Now the first parser will return both the unconsumed input and the parse tree. The second parser is now able to work with the unconsumed input of the first parser and also build on the parse tree from the first parser.

Next we refine our idea of a parser further. It is possible for a string to be parsed in multiple ways for some grammars. Our monad should be able to deal with this situation. Hence to allow multiple parses for each input we return not a single pair, but a list of pairs of unconsumed input and the parse tree for the unconsumed input.

As we want our parser to be as general as possible, we do not want to fix the type of tree returned. For instance we could have a tree of integers or strings. Finally we refine this definition further to allow different kinds of trees by using a type variable which represent the type of the parse tree. Hence we are left with the generic parser that returns a tree of type `a`.

```
newtype Parser a = MkP(String -> [(a, String)])
```

where `MkP` is a type constructor and `a` is a type variable. An instance of this type is a function (labelled with the constructor MkP) which takes a string as input and returns a list of pairs of possible parses (which will have consumed some of the input) and the remaining unconsumed input for that particular parse. It is also possible for a parser to fail to parse a given input. This is represented by the parser returning an empty list of parses.

We now need a function `apply` which applies a parser to a string.

```
apply :: Parser a -> String -> [(a, String)]
apply (MkP f) s =  f s
```

Here the first parameter is a parser that returns a list of parses (which are pairs of values of type `a` and strings of unconsumed input) and the second is the input string. So the definition is simply apply the function `f` to the string `s` (and forget about the constructor `MkP`).

We can now create our monad based on the function `apply`. The monad will control how to sequence parsers and will also control the order of execution. Hence the monad will need to control how the unconsumed input of the first parser will be passed as the input to the second parser and also how the resulting trees will be passed along to subsequent parsers. Once this is done we can use the `do notation` within Haskell to easily form new parsers from existing parsers by combining them together.

We instantiate the `Monad` class with our type `Parser` and provide definitions of the required functions.

```
instance Monad Parser where
  return x = MkP f where f s = [(x, s)]
  p >>= q  = MkP f
             where f s = [(y, s'') | (x, s') <- apply p s,
                                     (y, s'') <- apply (q x) s']
```

It is these definitions that allow us to sequence parsers together. These definitions do obey the properties set out in Section 2.1. In the definition of `return x`, we return the parser `MkP f` (which is the function `f` labelled with the constructor `MkP`) which takes a string of unconsumed input `s` and returns a list of pairs `[(x,s)]`, where `x` is the existing parse tree to build upon. Hence this parser returns a single parse (as the list contains only one element) where no input has been consumed (as the string `s` is returned in the second component of the pair) and the existing parse tree `x` (first component). This is necessary when we want to change a parse tree into a parser that simply returns that tree with out affecting the input string.

In the definition of `>>=`, we create a new *parser* from the existing parsers `p` and `q`. When the new parser `q >>= q` is applied to the input string `s`, the parser first applies the parser `p` to the input string `s` (which results in a list pairs of parses and corresponding unconsumed input for each parse) and then applies the parser `q` to each of these parses and the corresponding unconsumed input for each parse. The result of this combination of parsers `p` and `q` is a list of parses for the parser `p >>= q`.

We can now use the *do* notation which is available for all monad types in order to sequence parsers together – see Section 2.1.

2.3.2 Basic parsers

Now that the type `Parser` is a monad we can combine parsers together using the *do notation*. First we build some generic basic parsers. The `item` parser consumes and returns the first character of the input. This parser will only fail when the input is exhausted, i.e., the input string is empty.

```
item :: Parser Char
item =  MkP f
        where f []     = []
              f(c:cs) = [(c,cs)]
```

Another basic parser that we need (which does not build upon `item`) is the parser that always fails i.e., returns an empty list no matter what the input is. Its called zero because it returns zero parses.

```
zero :: Parser a
zero = MkP f where f s = []
```

These are the most basic parsers and all the other parsers that we define are built on top of these. We now build a slightly more complex parser that recognises a character that satisfies a given property.

```
sat :: (Char -> Bool) -> Parser Char
sat p = do c <- item
           if p c then return c else zero
```

Here we consume a character using `item` and we assign this to `c`. We then test `c` with the property `p`; if `c` passes the test (i.e., `p c` is true) we return the parser for the character `c` else we return the zero parser, indicating that this parse failed.

Using the `sat` parser we can build a parser that recognises a given character.

```
char :: Char -> Parser ()
char x = do c <- sat (==x)
            return ()
```

Here the test is the function `==x` which takes a character and returns true only if it is equal to `x`. If the `sat (==x)` parser consumes a character then we return a parser that just consumes a character and returns the empty tree, as we already know the character that was consumed was `x`. If the `sat` parser fails then this parser will also fail.

Using recursion and the `char` parser we can now build a parser that recognises an entire string.

```
string :: String -> Parser ()
string []     = return ()
string (x:xs) = do char x
                   string xs
                   return ()
```

When applied to the empty string we succeed and return the empty tree. When applied to a string with at least one character, we first recognise that character, then recognise the rest of the string, then return the empty tree. Again we return the empty tree on a successful parse because we already know the string that a successful parse will consume.

Here we define parser that parses a digit. Notice that it is important to know which digit was parsed, which is why we return the actual number that was parsed (as an integer).

```
digit :: Parser Int
digit = do d <- sat isDigit
           return (ord d - ord '0')
```

2.3.3 Alternation and repetition parsers

We now need a mechanism for either applying parser p or parser q in the case that parser p fails. This is done by the orelse parser (denoted here by |||) which combines the existing parsers p and q into a new parser p ||| q.

```
(|||) :: Parser a -> Parser a -> Parser a
p ||| q = MkP f
          where f s = if null ps then apply q s else ps
                      where ps = apply p s
```

We return the parser that when applied to the string s will first try and apply parser p to the string s and if the the empty parse is returned (i.e., parser p failed) then we apply parser q to the string s. To test whether the empty parse is returned we use the null operation which tests whether a list is empty. If parser q applied to s also fails then the parser p ||| q fails. As monads take care of sequencing and a valid monad is associative we can use the orelse parser in sequence to form the parser p ||| q ||| r that tries to apply p if that fails tries to apply q and if that fails finally tries to apply r.

We now use the orelse parser to build a parser combinator that repeats a parser zero or more times.

```
many :: Parser a -> Parser [a]
many p = do x <- p
            xs <- many p
            return (x:xs)
       ||| return []
```

The many parser will take as input, a parser p which returns parse trees of type a. It will then return a parser that returns parse trees of lists of type a. For instance, if we have a parser p that recognises a digit, then by applying the many parser to the parser p we can recognise many digits. The result will be a list of recognised digits. This is why we return a list.

The many parser works by first using the parser p. Then p's result is bound to the variable x, and finally a recursively call to the parser many is made. If the parser p fails then we use the orelse parser to return the empty list. We then concatenate the results into a list which is returned. the concatenation works well with the recursion and the base case of an empty list.

We use the many parser to define another parser some.

```
some :: Parser a -> Parser [a]
some p = do x <- p
            xs <- many p
            return (x:xs)
```

The parser some repeats the parser p at least once and also uses recursion in a similar way to the many parser.

We use the many parser to build a parser for natural numbers, where we use the foldl Haskell operation to change the list of digits into the actual natural number.

```
nat :: Parser Int
nat = do ds <- some digit
         return (foldl (++++) 0 ds)
       where m ++++ n = 10 * m + n
```

We return the type `Int` for simplicity as there is no type of natural numbers in Haskell.

2.3.4 White space parsers

We now need parsers to discard white spaces that might appear in syntax. First we define a parser that recognises many single spaces

```
space :: Parser ()
space = many (sat isSpace) >> return ()
```

The function `isSpace` is a built in Haskell function which tests a character as white space.

Next we use the `space` parser to define a parser `token p` that skips white space around a parser `p`.

```
token :: Parser a -> Parser a
token p = do space
             x <- p
             space
             return x
```

We first recognise some white spaces using the `space` parser, then we apply the parser `p` and bind the result to `x`. We then recognise some more white spaces. Finally, we return the parser `return x`, which is the parser that returns the tree `x` without consuming any further input.

We now build our last generic parser that recognises a string surrounded by space.

```
symbol :: String -> Parser ()
symbol xs = token (string xs)
```

2.3.5 Constructing the CSP parser

We finally use the previously defined parsers to create a parser for the following CSP grammar:-

```
CSP = SKIP          |
      STOP          |
      a -> P        |
      (P [] Q)      |
      (P |~| Q)     |
      (P || Q)
```

where actions are natural numbers (but will be represented by integers in Haskell). The parenthesis here are needed for the grammar to be unambiguous. The parenthesis could be omitted by prioritising the CSP operators. However, for simplicity, we force the use of parenthesis.

First we define a recursive abstract data type to store our parsed tree which reflects our grammar.

```
data CspTree = Skip | Stop | Prefix Int CspTree |
               ExternalChoice CspTree CspTree   |
               InternalChoice CspTree CspTree   |
               SyncParallel   CspTree CspTree
```

This type will be returned by our CSP parser. We define our top level parser as

```
cspParser :: Parser CspTree
cspParser =  skip ||| stop ||| prefix ||| externalChoice |||
             internalChoice ||| syncParallel
```

which tries to recognise each sub-CSP parser in turn using the `orelse` parser (`|||`). Each of these sub-CSP parsers are mutually exclusive thanks to the parenthesis of our grammar. We could use the order of the `orelse` parser to encode priorities of the operators, in order to reduce the use of parenthesis. We have chosen to use a simple grammar with forced parenthesis with means we have no need for priorities.

We define our `skip` and `stop` parsers as

```
skip :: Parser CspTree
skip =  do symbol "SKIP"
           return Skip

stop :: Parser CspTree
stop =  do symbol "STOP"
           return Stop
```

which recognise a string of "SKIP" or "STOP" which may be surrounded by space. We return a tree representing the primitive process `SKIP` and `STOP`, respectively.

Next we define the `prefix` parser which will recognise an action as a natural number using infix notation.

```
prefix :: Parser CspTree
prefix =  do n <- token nat
             symbol "->"
             p <- cspParser
             return (Prefix n p)
```

Here a natural number is recognised sounded by space, then an arrow symbol and finally the rest of the CSP process. The result of this is the tree `Prefix n p` where `n` is the action (natural number) of the prefix action operator and `p` is the tree that is a result of parsing the rest of the CSP process.

Finally the other parsers for the remaining CSP operators are defined in a similar way. Such parsers work by recognising a process followed by an operator symbol followed by another processes. The parsers then return the tree representing the parsed process. Parenthesis are forced to stop ambiguity within the grammar.

```
externalChoice :: Parser CspTree
externalChoice = do symbol "("
                    p <- cspParser
                    symbol "[]"
                    q <- cspParser
                    symbol ")"
                    return (ExternalChoice p q)

internalChoice :: Parser CspTree
internalChoice = do symbol "("
                    p <- cspParser
                    symbol "|~|"
                    q <- cspParser
                    symbol ")"
                    return (InternalChoice p q)

syncParallel :: Parser CspTree
syncParallel = do symbol "("
                  p <- cspParser
                  symbol "||"
                  q <- cspParser
                  symbol ")"
                  return (SyncParallel p q)
```

The parser `cspParser` will now recognise the above grammar. An improved parser would allow you to change the type of action from natural numbers to other types and also not require unnecessary parenthesis around some expressions.

2.3.6 Some utility functions

In order to output a CSP parse tree (a value of type `CspTree`) to standard output we need to first be able to convert the parse tree to a string. To do this we must instantiate the Haskell class `show` with the data type `CspTree`. This will require us to provide a definition for the function `show` for each constructor of the data type `CspTree`. This the following code instantiate the class `show` with the data type `CspTree` and provides suitable definitions for the function show.

```
instance Show CspTree where
  show Skip                 = "SKIP"
  show Stop                 = "STOP"
  show (Prefix n p)         = show n ++ " -> " ++ show p
  show (ExternalChoice p q) = "(" ++ show p ++ " [] " ++
```

```
                                  show q ++ ")"
show (InternalChoice p q) = "(" ++ show p ++ " |~| " ++
                                  show q ++ ")"
show (SyncParallel p q)   = "(" ++ show p ++ " || " ++
                                  show q ++ ")"
```

The definitions of the function `show` simply convert a value of type `CspTree` into a string in the obvious way where ++ is string concatenation. We have used here, the string concatination operator as it serves our purpose well. The running time of this code is currently not an issue, however it is quadratic. There is a standard way of reducing this to a linear runtime if we need the code to be faster later on. However, it currently suits our purposes and needs no further improvement.

Finally before we can run the parser properly we need a few small utility functions. Firstly we are only interested in full parses i.e., when all the input has been consumed. We also only need a single full parse as our grammar does not allow multiple ambiguous parses. We use the `Maybe` monad here that allows us to simulate partial functions by returning either `Just` a value or `Nothing`. The following function will return the first full parse in a list of parses.

```
findValidParse :: [(a, String)] -> Maybe a
findValidParse []                       = Nothing
findValidParse ((tree, "")     :others) = Just tree
findValidParse ((tree, (x:xs)):others) = findValidParse others
```

If the list is empty we return `Nothing` indicating that no parse was found. In the second case of there being at the start of a list, a parse which consumed all input then we simply return this tree with the added constructor `Just`. In the third case of finding at the start of the list a parse that has not consumed all the input, we just recurse on the rest of the list.

We finally create a user friendly wrapper function to call a parser.

```
parse   :: Parser a -> String -> Maybe a
parse p s = findValidParse(apply p s)
```

We can now run the CSP parser `cspParser` on a string `s` with the code

```
parse cspParser s
```

This will either return `Nothing` indicating no parse was found or `Just tree` where tree is a successful parse of type `CspTree`.

2.3.7 An example run

We conclude this chapter with two runs of our CSP parser. First we try a positive run where we expect a valid parse to be found. We issue the following command Haskell command

```
parse cspParser "(((1 -> STOP [] 2 -> 3 -> SKIP) |~| STOP)
                 || 4 -> SKIP)"
```

This will attempt to parse a CSP string from our grammar. We have used every operator within our grammar as part of this test. The output of Haskell is a valid parse which when shown (output to standard output) yields

```
Just (((1 -> STOP [] 2 -> 3 -> SKIP) |~| STOP) || 4 -> SKIP)
```

The `Just` is the shown string of the `Just` constructor returned by our wrapper function.

We now try a negative test where we expect the parser to fail. We try to parse a non valid CSP expression with the code

```
parse cspParser "4 -> STOP -> STOP"
```

The string `4 -> STOP -> STOP` is not a valid CSP expression from our grammar. The test produces the following output when shown.

```
Nothing
```

This indicates that no parse was possible. Both tests have indicated that our CSP parser runs as expected.

Chapter 3

Dealing with logics

Contents

Institutions capture the nature of logic systems and are the language used to define CASL and CSP-CASL. The tool HETS [MML07] (see Section 4.2) implements various institutions centred around the CASL institution. HETS also uses institution representations in order to translate specifications from one logic to another.

Section 3.1 introduces the notions of institutions. An own example of the institution *many-sorted equational logic* is then presented in Section 3.2. Further examples of institutions necessary for the constructions within this thesis are presented in Section 3.3. The category of presentations is briefly introduced in Section 3.4. Institutions representations are then introduced in Section 3.5, which allow one to related different institutions. Finally in Section 3.6 we present two instances of institution representations which are used within HETS and play a central role within this thesis.

3.1 Institutions

Institutions were introduced by Joseph Goguen and Rod Burstall in the late 1970's in order to deal with the large volume of logical systems being used and developed in computer science. Institutions capture the very essence of what a logical system is. By using institutions it is possible to create specification languages, proof calculi and tools which are independent of the underlying logical system. Institution representations allow one to relate and translate institutions with other institutions. Goguen and Burstall sum up the idea of an institution in the following slogan taken from [GB92]

"Truth is invariant under change of notation".

This slogan expresses that truth is independent of which symbols we use to represent our functions, relations and variables. Hence if we have a logical statement and we swap all occurrences of our variable symbols, function symbols and relation symbols with different symbols then our statement means the same as the original statement.

The formal definition of institutions and institution representations rely on Category Theory. 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 roughly means that if one translates a sentence under a signature with a signature morphism, then the satisfaction of a translated sentence and a model is preserved. The satisfaction condition is the formal notion of the above slogan, where the signature morphism is the function that swaps our symbols.

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

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

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

holds for every $\varphi \in \mathbf{sen}^I(\Sigma)$ and for every $M' \in |\mathbf{mod}^I(\Sigma')|$. Figure 3.1 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 3.1 depicts the category **SIGN** with two signatures Σ and Σ' along with a signature morphism σ between them.

The functor $\mathbf{sen}^I : \mathbf{SIGN}^I \to \mathbf{SET}$ gives for each signature $\Sigma : \mathbf{SIGN}^I$, the set of sentences $\mathbf{sen}^I(\Sigma)$ over the signature Σ, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the map $\mathbf{sen}^I(\sigma) : \mathbf{sen}^I(\Sigma) \to \mathbf{sen}^I(\Sigma')$ which translates sentences built over Σ to sentences built over Σ'. The category **SET** is the category where the objects are sets, the morphisms are total functions between sets, composition is functional composition and the identity function assigns to every set the identity function on that set [Fia04]. The top right of Figure 3.1 depicts the category **SET** with two sets of signatures $\mathbf{sen}(\Sigma)$ 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 $\Sigma : \mathbf{SIGN}^I$, the category of models for that signature $\mathbf{mod}^I(\Sigma)$, and for each signature morphism $\sigma : \Sigma \to \Sigma'$, the reduct functor $\mathbf{mod}^I(\sigma) : \mathbf{mod}^I(\Sigma') \to \mathbf{mod}^I(\Sigma)$ which reduces models over the signature Σ' to models over the signature Σ. The category $\mathbf{SIGN}^{op}$ is the category with the same objects and morphisms as **SIGN**, but where the morphisms have been reversed, e.g., if $\sigma : \Sigma \to \Sigma'$ is a morphism in **SIGN**, then $\sigma : \Sigma' \to \Sigma$ is a morphism in $\mathbf{SIGN}^{op}$. Morphism composition is also reversed within this

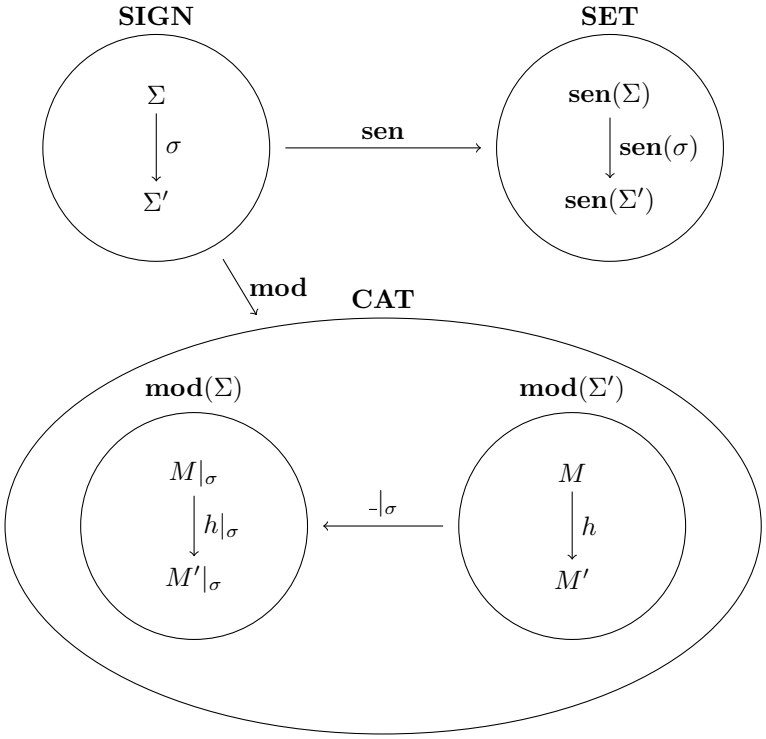

Figure 3.1: Diagram of the notion of an institution.

category. The category **CAT** is the category of categories, where the objects are categories and the morphisms are functors[1].

The lower half of Figure 3.1 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 and M' 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 $m|_\sigma$, $m'|_\sigma$ and $h|_\sigma$ respectively.

As $\mathbf{mod}^I$ is a functor into the category **CAT**, $\mathbf{mod}^I(\Sigma)$ is the category where objects are models over the signature Σ and morphisms are the homomorphisms between the models. 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. Hence $|\mathbf{mod}^I(\Sigma)|$ is the class of models of Σ within the institution I.

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].

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

Thus for each $\sigma : \Sigma \to \Sigma'$ in **SIGN**, the satisfaction condition becomes

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

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

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

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

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

3.2 Many-sorted equational logic as an institution

We illustrate the concept of an institution with an own example. Here, we define `many-sorted equational logic` as an institution.

We first define what our signatures and signature morphisms are (the category **SIGN**) – Section 3.2.1, where we prove that **SIGN** is a valid category. We then define the functor **sen** in Section 3.2.2, where we prove that **sen** is a valid functor. Next we define the functor **mod** in Section 3.2.3, where we prove that $\mathbf{mod}(\Sigma)$ forms a category for each signature Σ and that the reduct functor $\mathbf{mod}(\sigma)$ is a functor for each signature morphism $\sigma : \Sigma \to \Sigma'$. We also prove that **mod** itself is a functor. Finally we define the satisfaction relation $\models$ and prove that the satisfaction condition holds in Section 3.2.4.

3.2.1 Many-sorted equation logic - Category SIGN

One particular notion of a signature as defined by [LEW97] is a pair of sets $\Sigma = (S, \Omega)$, where

- S is a set of sorts.
- Ω is a set of total functions symbols, of the form $n : s_1 \times \ldots \times s_k \to s$ with $s_1, \ldots, s_k, s \in S$ and $k \geq 0$.

Given two signatures $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$, a signature morphism $\sigma : \Sigma \to \Sigma'$ is a pair of functions $(\sigma_S, \sigma_\Omega)$ where

- $\sigma_S : S \to S'$
- $\sigma_\Omega : \Omega \to \Omega'$

such that for each function symbol $n : s_1 \times \ldots \times s_k \to s \in \Omega$, $k \geq 0$, there exists a function name m with $\sigma_\Omega(n : s_1 \times \ldots \times s_k \to s) = (m : \sigma_S(s_1) \times \ldots \times \sigma_S(s_k) \to \sigma_S(s))$, such that $m : \sigma_S(s_1) \times \ldots \times \sigma_S(s_k) \to \sigma_S(s) \in \Omega'$. This forms our category **SIGN**.

Given two signature morphisms $\sigma : \Sigma \to \Sigma'$ and $\sigma' : \Sigma' \to \Sigma''$ morphism composition is defined as:

$$\sigma' \circ \sigma = (\sigma'_S, \sigma'_\Omega) \circ (\sigma_S, \sigma_\Omega) = (\sigma'_S \circ \sigma_S, \sigma'_\Omega \circ \sigma_\Omega,)$$

Let $Id_\Sigma = (Id_{\Sigma_S}, Id_{\Sigma_\Omega}) : \Sigma \to \Sigma'$ denote the identity morphism of $\Sigma = (S, \Omega)$ where $Id_{\Sigma_S} : S \to S$ and $Id_{\Sigma_\Omega} : \Omega \to \Omega$ are defined as

- $Id_S(s) = s$ and
- $Id_\Omega(\omega) = \omega$

for any $s \in S$ and $\omega \in \Omega$.

Theorem 3.1: SIGN is a category.

To prove that **SIGN** is a category, we must prove that identity morphisms exist (see Lemma 3.2) and that morphism composition is associative (see Lemma 3.3).

Lemma 3.2: Identity morphisms exist in **SIGN** i.e., for all signature morphisms $\sigma : \Sigma \to \Sigma'$ there exist morphisms Id_Σ and $Id_{\Sigma'}$ such that

$$\sigma \circ Id_\Sigma = \sigma \text{ and } Id_{\Sigma'} \circ \sigma = \sigma$$

Proof. Let $\sigma : \Sigma \to \Sigma'$ be an arbitrary signature morphism where $\Sigma = (S, \Omega)$. We must prove that $\sigma \circ Id_\Sigma = \sigma$.

$\sigma \circ Id_\Sigma$	$=$	$(\sigma_S, \sigma_\Omega) \circ (Id_{\Sigma_S}, Id_{\Sigma_\Omega})$	Unfolding the definition of σ and Id_Σ.
	$=$	$(\sigma_S \circ Id_{\Sigma_S}, \sigma_\Omega \circ Id_{\Sigma_\Omega})$	Definition of morphism composition.
	$=$	$(\sigma_S, \sigma_\Omega)$	Definition of identity morphism.
	$=$	σ	The definition of σ.

Since σ is an arbitrary signature morphism, we can conclude that $\sigma \circ Id_\Sigma = \sigma$ for all signature morphisms $\sigma : \Sigma \to \Sigma'$. The same line of thought holds for the proof of $Id_{\Sigma'} \circ \sigma = \sigma$. □

Lemma 3.3: Morphism composition is associative in **SIGN**, i.e., for all signature morphisms $\sigma : \Sigma \to \Sigma'$, $\sigma' : \Sigma' \to \Sigma''$ and $\sigma'' : \Sigma'' \to \Sigma'''$, where Σ, Σ', Σ'', and Σ''' are signatures in **SIGN**,

$$\sigma'' \circ (\sigma' \circ \sigma) = (\sigma'' \circ \sigma') \circ \sigma$$

Proof. Let $\sigma : \Sigma \to \Sigma'$, $\sigma' : \Sigma' \to \Sigma''$ and $\sigma'' : \Sigma'' \to \Sigma'''$ be arbitrary signature morphisms where $\Sigma =$, Σ', Σ'', and Σ''' are signatures in **SIGN**. We must prove that $\sigma'' \circ (\sigma' \circ \sigma) = (\sigma'' \circ \sigma') \circ \sigma$.

$\sigma'' \circ (\sigma' \circ \sigma)$	$=$	$(\sigma''_S, \sigma''_\Omega) \circ ((\sigma'_S, \sigma'_\Omega) \circ (\sigma_S, \sigma_\Omega))$	Unfolding the morphism definition.
	$=$	$(\sigma''_S, \sigma''_\Omega) \circ ((\sigma'_S \circ \sigma_S), (\sigma'_\Omega \circ \sigma_\Omega))$	Definition of morphism composition.
	$=$	$(\sigma''_S \circ (\sigma'_S \circ \sigma_S), \sigma''_\Omega \circ (\sigma'_\Omega \circ \sigma_\Omega))$	Definition of morphism composition.
	$=$	$((\sigma''_S \circ \sigma'_S) \circ \sigma_S), (\sigma''_\Omega \circ \sigma'_\Omega) \circ \sigma_\Omega)$	Associativity of functions.
	$=$	$((\sigma''_S \circ \sigma'_S), (\sigma''_\Omega \circ \sigma'_\Omega)) \circ (\sigma_S, \sigma_\Omega)$	Definition of morphism composition.
	$=$	$((\sigma''_S, \sigma''_\Omega) \circ (\sigma'_S, \sigma'_\Omega)) \circ (\sigma_S, \sigma_\Omega)$	Definition of morphism composition.
	$=$	$(\sigma'' \circ \sigma') \circ \sigma$	Folding the morphism definition.

Since σ, σ', and σ'' are arbitrary signature morphisms, we can conclude that $\sigma'' \circ (\sigma' \circ \sigma) = (\sigma'' \circ \sigma') \circ \sigma$ for all signature morphisms $\sigma : \Sigma \to \Sigma'$, $\sigma' : \Sigma' \to \Sigma''$, and $\sigma'' : \Sigma'' \to \Sigma'''$. □

This completes the proof obligations that **SIGN** is indeed a valid category.

3.2.2 Many-sorted equation logic - Functor sen

Given a signature $\Sigma = (S, \Omega)$ and a family of variables $X = (X_s)_{s \in S}$ of disjoint sets (which may or may not be infinite), the terms of sort s over Σ using the family of variables X is denoted as $T_{\Sigma(X),s}$ defined by:

1. $X_s \subseteq T_{\Sigma(X),s}$.
2. if $n :\rightarrow s$ is an operation of Ω then $n \in T_{\Sigma(X),s}$.
3. if $n : s_1 \times \ldots \times s_k \rightarrow s$, $k \geq 1$ is an operation of Ω and if $t_i \in T_{\Sigma(X),s_i}$, for $1 \leq i \leq k$, then $n(t_1, \ldots, t_k) \in T_{\Sigma(X),s}$.

Now that we have terms we can build formulae. Formulae in many-sorted equational logic are very simple, they are just equations quantified over all variables, that compare two terms of the same sort. For each signature Σ the set of sentences is

$$\mathbf{sen}(\Sigma) = \{\forall X.t = u \mid t, u \in T_{\Sigma(X),s}\}$$

In order to translate sentences we must first be able to translate variables and terms. Given a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ and a family of disjoint sets $(X_s)_{s \in S}$ the variable translation is defined as

$$\sigma((X_s)_{s \in S}) = ((\bigcup_{\sigma(s)=s'} X_s)_{s' \in S'}).$$

Theorem 3.4: $\sigma(X)$ is a family of disjoint sets for all signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$ and for all families of disjoint sets $(X_s)_{s \in S}$.

Proof. Let $\sigma : \Sigma \rightarrow \Sigma'$ be an arbitrary signature morphism and $(X_s)_{s \in S}$ be an arbitrary family of disjoint sets, where $\Sigma = (S, \Omega)$. We must show that $\sigma(X)$ is a family of disjoint sets.

Assume $\sigma(X)$ is family of sets which is not disjoint. This means that there exists $s'_1, s'_2 \in S'$ with $s'_1 \neq s'_2$ such there exists $x' \in \sigma(X)_{s'_1}$ and $x' \in \sigma(X)_{s'_2}$. This can only be the case if there exists some $s_1, s_2 \in S$ with $s_1 \neq s_2$ and $\sigma(s_1) = s'_1$, $\sigma(s_2) = s'_2$ such that there exists $x \in X_{s_1}$ and $x \in X_{s_2}$. Since as X is a family of disjoint sets, we know that for all $s_a, s_b \in S$ such that $s_a \neq s_b$ it is the case that $X_{s_a} \cap X_{s_b} = \emptyset$, thus we have a contradiction. We can conclude that $\sigma(X)$ is a family of disjoint sets.

As σ and X are arbitrary, we can conclude that $\sigma(X)$ is a family of disjoint sets for all signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$ and for all families of disjoint infinite sets $(X_s)_{s \in S}$. □

Given a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ and a family of variables X for the signature Σ, where $\Sigma = (S, \Omega)$, then term translation is defined by:

1. $\sigma_T(x) = \sigma(x)$ for any $x \in X$.
2. $\sigma_T(n) = m$ for any $n :\rightarrow s \in \Omega$ with $\sigma(n :\rightarrow s) = (m :\rightarrow \sigma(s))$.
3. $\sigma_T(n(t_1, \ldots, t_k)) = m(\sigma_T(t_1), \ldots, \sigma_T(t_k)$ for any $(n : s_1 \times \ldots \times s_k \rightarrow s \in \Omega)$, $k \geqslant 1$, with $\sigma(n : s_1 \times \ldots \times s_k \rightarrow s) = (m : \sigma(s_1) \times \ldots \times \sigma(s_k) \rightarrow \sigma(s))$.

The subscript T on the term translation may be omitted where no confusion arises.

We now define the translation of sentences, given a signature morphism $\sigma : \Sigma \to \Sigma'$, sentence translation $\mathbf{sen}(\sigma)$ is defined as

$$\mathbf{sen}(\sigma)(\forall X.t = u) = \forall\, \sigma(X).\sigma(t) = \sigma(u).$$

Theorem 3.5: **sen** is a functor.

To prove **sen** is a functor we must prove that **sen** preserves identity morphisms (see Lemma 3.6) and that **sen** preserves composition of morphisms (see Lemma 3.7).

Lemma 3.6: **sen** preserves identity morphisms, i.e., for all Σ in **SIGN**

$$\mathbf{sen}(Id_\Sigma) = Id_{\mathbf{sen}(\Sigma)}$$

where $Id_{\mathbf{sen}(\Sigma)}$ denotes the identity morphism (which in this case is the identity function) for the set $\mathbf{sen}(\Sigma)$.

Proof. Let $\Sigma = (S, \Omega)$ be an arbitrary signature in **SIGN**. Let φ be an arbitrary sentence in $\mathbf{sen}(\Sigma)$. We know that φ has the form $\forall X.t = u$ where $t, u \in T_{\Sigma(X),s}$ for some $s \in S$. We must prove that $\mathbf{sen}(Id_\Sigma)(\varphi) = Id_{\mathbf{sen}(\Sigma)}(\varphi)$.

$\mathbf{sen}(Id_\Sigma)(\varphi)$	$=$	$\mathbf{sen}(Id_\Sigma)(\forall X.t = u)$	Definition of φ.
	$=$	$\forall\, Id_\Sigma(X).Id_\Sigma(t) = Id_\Sigma(u))$	Definition of **sen**.
	$=$	$\forall X.t = u$	Definition of identities, variable translation, and term translation.
	$=$	$Id_{\mathbf{sen}(\Sigma)}(\forall X.t = u)$	Definition of $Id_{\mathbf{sen}(\Sigma)}$.
	$=$	$Id_{\mathbf{sen}(\Sigma)}(\varphi)$	Definition of φ.

Since φ and Σ are arbitrary, we can conclude $\mathbf{sen}(Id_\Sigma) = Id_{\mathbf{sen}(\Sigma)}$ for all signatures Σ in **SIGN**. □

Lemma 3.7: **sen** preserves composition of morphisms, i.e., for all signature morphisms $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$

$$\mathbf{sen}(\rho \circ \sigma) = \mathbf{sen}(\rho) \circ \mathbf{sen}(\sigma).$$

Proof. Let $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$ be arbitrary signature morphisms, where $\Sigma = (S, \Omega)$. Let φ be an arbitrary sentence in $\mathbf{sen}(\Sigma)$. We know that φ has the form $\forall X.t = u$ where $t, u \in T_{\Sigma(X),s}$ for some $s \in S$. We must prove that $\mathbf{sen}(\rho \circ \sigma)(\varphi) = (\mathbf{sen}(\rho) \circ \mathbf{sen}(\sigma))(\varphi)$.

L.H.S	$=$	$\mathbf{sen}(\rho \circ \sigma)(\varphi)$	
	$=$	$\mathbf{sen}(\rho \circ \sigma)(\forall X.t = u)$	Definition of φ.
	$=$	$\forall(\rho \circ \sigma)(X).(\rho \circ \sigma)(t) = (\rho \circ \sigma)(u)$	Definition of **sen**.
	$=$	$\forall(\rho(\sigma(X))).(\rho(\sigma(t))) = (\rho(\sigma(u)))$	Definition of morphism composition.
R.H.S	$=$	$(\mathbf{sen}(\rho) \circ \mathbf{sen}(\sigma))(\varphi)$	
	$=$	$(\mathbf{sen}(\rho) \circ \mathbf{sen}(\sigma))(\forall X.t = u)$	Definition of φ.
	$=$	$\mathbf{sen}(\rho)(\mathbf{sen}(\sigma)(\forall X.t = u))$	Definition of morphism composition .
	$=$	$\mathbf{sen}(\rho)(\forall\, \sigma(X).\sigma(t) = \sigma(u))$	Definition of **sen**.
	$=$	$\forall(\rho(\sigma(X))).(\rho(\sigma(t))) = (\rho(\sigma(u)))$	Definition of **sen**.

Since σ, ρ and φ are arbitrary, we can conclude that $\mathbf{sen}(\rho \circ \sigma) = \mathbf{sen}(\rho) \circ \mathbf{sen}(\sigma)$ for all signature morphisms $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$. □

This completes the proof obligations that **sen** is indeed a valid functor.

3.2.3 Many-sorted equation logic - Functor mod

The functor $\mathbf{mod} : (\mathbf{SIGN})^{op} \to \mathbf{CAT}$ maps into the category of categories. Hence, for any signature Σ in the category **SIGN**, $\mathbf{mod}(\Sigma)$ must be a category. We define $\mathbf{mod}(\Sigma)$ to be the category of total algebras for the signature Σ (our models), and the morphisms in this category to be homomorphisms between the total algebras (model morphisms).

Given a signature $\Sigma = (S, \Omega)$, a model (a total algebra) for Σ assigns:

- A carrier set $A(s)$ to each sort symbol $s \in S$.
- A total function $A(n : s_1 \times \ldots \times s_k \to s) : A(s_1) \times \ldots \times A(s_k) \to A(s)$ to each function symbol $(n : s_1 \times \ldots \times s_k \to s) \in \Omega, k \geq 0$.

Given two models A, B in $\mathbf{mod}(\Sigma)$, a model morphism (homomorphism) $h : A \to B$ is a family $(h_s)_{s \in S}$ of functions $h_s : A(s) \to B(s)$ such that for any function symbol $\omega \in \Omega$, say $\omega = (n : s_1 \times \ldots \times s_k \to s), k \geq 0$, the following condition holds:

$$h_s(A(\omega)(a_1, \ldots, a_k)) = B(\omega)(h_{s_1}(a_1), \ldots, h_{s_k}(a_k))$$

for all $(a_1, \ldots, a_k) \in A(s_1) \times \ldots \times A(s_k)$.

Within the category $\mathbf{mod}(\Sigma)$ for a given signature Σ, composition of morphisms is homomorphism composition and the identity morphism for an algebra A (denoted as Id_A) is the identity homomorphism for A.

Theorem 3.8: $\mathbf{mod}(\Sigma)$ is a category for every signature Σ in **SIGN**.

To prove that $\mathbf{mod}(\Sigma)$ is a category for every signature Σ in **SIGN**, we must prove that identity morphisms exist in $\mathbf{mod}(\Sigma)$ (see Lemma 3.9) and that morphism composition is associative in $\mathbf{mod}(\Sigma)$ (see Lemma 3.10).

Lemma 3.9: Given a signature $\Sigma = (S, \Omega)$ in **SIGN**, identity morphisms exist in $\mathbf{mod}(\Sigma)$, i.e., for all homomorphisms $h : A \to B$ in $\mathbf{mod}(\Sigma)$, there exist homomorphisms Id_A and Id_B such that

$$h \circ Id_A = h \text{ and } Id_B \circ h = h.$$

Proof. Let $h : A \to B$ be an arbitrary homomorphism in $\mathbf{mod}(\Sigma)$, where A and B are models in $\mathbf{mod}(\Sigma)$. We must prove that $h \circ Id_A = h$ i.e., $h_s \circ Id_{As} = h_s$ for all $s \in S$.

Let $s \in S$ and $x \in A(s)$.

$$\begin{array}{lcll} (h_s \circ Id_{As})(x) & = & h_s(Id_{As}(x)) & \text{Homomorphism composition.} \\ & = & h_s(x) & \text{Definition of the identity homomorphism } Id_A. \end{array}$$

Since x, s, and h are all arbitrary, we can conclude that $h \circ Id_A = h$ holds for all homomorphisms $h : A \to B$ in $\mathbf{mod}(\Sigma)$. The same line of thought holds for the proof of $Id_B \circ h = h$. □

Lemma 3.10: Given a signature $\Sigma = (S, \Omega)$ in **SIGN**, morphism composition is associative in $\mathbf{mod}(\Sigma)$, i.e., for all homomorphisms $h : A \to B$, $h' : B \to C$, and $h'' : C \to D$ in $\mathbf{mod}(\Sigma)$, where A, B, C, and D are models in $\mathbf{mod}(\Sigma)$,

$$h'' \circ (h' \circ h) = (h'' \circ h') \circ h.$$

Proof. Let $h : A \to B$, $h' : B \to C$, and $h'' : C \to D$ be arbitrary homomorphisms in $\mathbf{mod}(\Sigma)$ where A, B, C, and D are models in $\mathbf{mod}(\Sigma)$. We must prove that $h'' \circ (h' \circ h) = (h'' \circ h') \circ h$, i.e., $h''_s \circ (h'_s \circ h_s) = (h''_s \circ h'_s) \circ h_s$ for all $s \in S$.

Let $s \in S$ and $x \in A(s)$.

$$\begin{array}{lcll} (h''_s \circ (h'_s \circ h_s))(x) & = & (h''_s(h'_s(h_s(x)))) & \text{Homomorphism composition.} \\ & = & ((h''_s \circ h'_s) \circ h_s)(x) & \text{Homomorphism composition.} \end{array}$$

Since x, s, h, h', and h'' are all arbitrary we can conclude that $h'' \circ (h' \circ h) = (h'' \circ h') \circ h$ for all homomorphisms $h : A \to B$, $h' : B \to C$, and $h'' : C \to D$ in $\mathbf{mod}(\Sigma)$. □

This completes the proof obligations that $\mathbf{mod}(\Sigma)$ is indeed a valid category for every signature Σ in **SIGN**.

We have now defined $\mathbf{mod}(\Sigma)$ for a signature Σ in **SIGN**. The objects are total algebras and the morphisms are the homomorphisms between them.

Given two signatures $\Sigma = (S, \Omega)$, $\Sigma' = (S', \Omega')$, and a signature morphism $\sigma : \Sigma \to \Sigma'$ the σ-reduct of A' for an algebra A' in $\mathbf{mod}(\Sigma')$ is defined as:

- $(A' \mid_\sigma)(s) = A'(\sigma(s))$ for all $s \in S$.
- $(A' \mid_\sigma)(\omega) = A'(\sigma(\omega))$ for all $\omega \in \Omega$.

We now define the functor **mod** on signature morphisms. Given a signature morphism $\sigma : \Sigma \to \Sigma'$, $\mathbf{mod}(\sigma) : \mathbf{mod}(\Sigma') \to \mathbf{mod}(\Sigma)$ is a functor.

For each total algebra A' in $\mathbf{mod}(\Sigma')$, we define $\mathbf{mod}(\sigma)(A')$ as $A'|_\sigma$ (the σ-reduct of A').

For each homomorphism $h' : A' \to B'$ in $\mathbf{mod}(\Sigma')$, we define $\mathbf{mod}(\sigma)(h')$ to be the homomorphism reduct $h'|_\sigma : A'|_\sigma \to B'|_\sigma$ where

$$(h'|_\sigma)_s = h'_{\sigma(s)} \text{ for each } s \in S.$$

We have now defined our functor **mod**. We have defined its behaviour on objects and morphisms. It maps into the category **CAT**, where objects are categories, morphisms are functors, morphism composition is functor composition and the identity morphisms are the identity functors. We denote the identity functor within the category **CAT** as $Id_{\mathbf{mod}(\Sigma)}$ for the category $\mathbf{mod}(\Sigma)$.

Theorem 3.11: $\mathbf{mod}(\sigma)$ is a functor for every signature morphism $\sigma : \Sigma \to \Sigma'$.

To prove $\mathbf{mod}(\sigma)$ is a functor for every signature morphism $\sigma : \Sigma \to \Sigma'$, we must prove that $\mathbf{mod}(\sigma)$ preserves identity morphisms (see Lemma 3.12) and that $\mathbf{mod}(\sigma)$ preserves composition of morphisms (see Lemma 3.13).

Lemma 3.12: Given a signature morphism $\sigma : \Sigma \to \Sigma'$ where $\Sigma = (S, \Omega)$, $\mathbf{mod}(\sigma)$ preserves identity morphisms, i.e., for all models A' in $\mathbf{mod}(\Sigma')$

$$\mathbf{mod}(\sigma)(Id_{A'}) = Id_{\mathbf{mod}(\sigma)(A')}.$$

Proof. Let A' be an arbitrary model in $\mathbf{mod}(\Sigma')$. We must prove that $\mathbf{mod}(\sigma)(Id_{A'}) = Id_{\mathbf{mod}(\sigma)(A')}$, i.e., $(\mathbf{mod}(\sigma)(Id_{A'}))_s = (Id_{\mathbf{mod}(\sigma)(A')})_s$ for all sorts $s \in S$.

Let $s \in S$ and $x \in (A'|_\sigma)(s)$.

L.H.S	$=$	$(\mathbf{mod}(\sigma)(Id_{A'}))_s(x)$	
	$=$	$((Id_{A'})|_\sigma)_s(x)$	Definition of $\mathbf{mod}(\sigma)$.
	$=$	$(Id_{A'})_{\sigma(s)}(x)$	Definition of homomorphism reduct.
	$=$	x	Definition of homomorphism identity.
R.H.S	$=$	$(Id_{\mathbf{mod}(\sigma)(A')})_s(x)$	
	$=$	$(Id_{(A'|_\sigma)})_s(x)$	Definition of $\mathbf{mod}(\sigma)$.
	$=$	x	Definition of homomorphism identity.

Since s, x and A' are all arbitrary we can conclude that $\mathbf{mod}(\sigma)(Id_{A'}) = Id_{\mathbf{mod}(\sigma)(A')}$ for all models A' in $\mathbf{mod}(\Sigma')$. □

Lemma 3.13: Given a signature morphism $\sigma : \Sigma \to \Sigma'$ where $\Sigma = (S, \Omega)$, $\mathbf{mod}(\sigma)$ preserves composition of morphisms, i.e., for all morphisms $h' : A' \to B'$ and $g' : B' \to C'$ in $\mathbf{mod}(\Sigma')$

$$\mathbf{mod}(\sigma)(g' \circ h') = \mathbf{mod}(\sigma)(g') \circ \mathbf{mod}(\sigma)(h').$$

Proof. Let $h' : A' \to B'$ and $g' : B' \to C'$ be arbitrary morphisms in $\mathbf{mod}(\Sigma')$. We must prove that $\mathbf{mod}(\sigma)(g' \circ h') = \mathbf{mod}(\sigma)(g') \circ \mathbf{mod}(\sigma)(h')$, i.e., $(\mathbf{mod}(\sigma)(g' \circ h'))_s = (\mathbf{mod}(\sigma)(g'))_s \circ (\mathbf{mod}(\sigma)(h'))_s$ for all sorts $s \in S$.

Let $s \in S$.

$(\mathbf{mod}(\sigma)(g' \circ h'))_s$	$=$	$((g' \circ h')|_\sigma)_s$	Definition of $\mathbf{mod}(\sigma)$.
	$=$	$(g' \circ h')_{\sigma(s)}$	Definition of homomorphism reduct.
	$=$	$g'_{\sigma(s)} \circ h'_{\sigma(s)}$	Homomorphism composition.
	$=$	$(g'|_\sigma)_s \circ (h'|_\sigma)_s$	Definition of homomorphism reduct.
	$=$	$(\mathbf{mod}(\sigma)(g'))_s \circ (\mathbf{mod}(\sigma)(h'))_s$	Definition of $\mathbf{mod}(\sigma)$.

Since s is an arbitrary sort in S and the morphisms h' and g' are arbitrary, we can conclude that $\mathbf{mod}(\sigma)(g' \circ h') = \mathbf{mod}(\sigma)(g') \circ \mathbf{mod}(\sigma)(h')$ for all morphisms $h' : A' \to B'$ and $g' : B' \to C'$ in $\mathbf{mod}(\Sigma')$. □

This completes the proof obligations that $\mathbf{mod}(\sigma)$ is indeed a valid functor for every signature morphism $\sigma : \Sigma \to \Sigma'$.

Theorem 3.14: **mod** is a functor.

To prove **mod** is a functor we must prove that **mod** preserves identity morphisms (see Lemma 3.15) and that **mod** preserves composition of morphisms (see Lemma 3.16).

Lemma 3.15: **mod** preserves identity morphisms, i.e., for all signatures Σ in **SIGN**

$$\mathbf{mod}(Id_\Sigma) = Id_{\mathbf{mod}(\Sigma)}$$

Proof. Let $\Sigma = (S, \Omega)$ be an arbitrary signature in **SIGN**. Let A be an arbitrary model in $\mathbf{mod}(\Sigma)$. We must prove that $\mathbf{mod}(Id_\Sigma)(A) = Id_{\mathbf{mod}(\Sigma)}(A)$.

L.H.S $\mathbf{mod}(Id_\Sigma)(A) = A|_{Id_\Sigma}$ which means

- for each $s \in S$ that $(A|_{Id_\Sigma})(s) = A(Id_\Sigma(s)) = A(s)$.
- for each $\omega \in \Omega$ that $(A|_{Id_\Sigma})(A)(\omega) = A(Id_\Sigma(\omega)) = A(\omega)$.

by definition of **mod** and the definition of the identity morphisms.

R.H.S $Id_{\mathbf{mod}(\Sigma)}(A) = A$ by the definition of identity functors in the category **CAT** and the functor **mod**. Hence, by the definition of a total algebra we have a set $A(s)$ for each sort $s \in S$ and a total function $A(\omega)$ for each function symbol $\omega \in \Omega$.

Since A and Σ are arbitrary, we can conclude $\mathbf{mod}(Id_\Sigma) = Id_{\mathbf{mod}(\Sigma)}$ for all signatures Σ : **SIGN**. □

Lemma 3.16: **mod** preserves composition of morphisms, i.e., for all signature morphisms $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$

$$\mathbf{mod}(\rho \circ \sigma) = \mathbf{mod}(\sigma) \circ \mathbf{mod}(\rho)$$

Proof. Let $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$ be arbitrary signature morphisms, where $\Sigma = (S, \Omega)$. Let A'' be an arbitrary model in $\mathbf{mod}(\Sigma'')$.

We must prove that $\mathbf{mod}(\rho \circ \sigma)(A'') = (\mathbf{mod}(\sigma) \circ \mathbf{mod}(\rho))(A'')$.

L.H.S $\mathbf{mod}(\rho \circ \sigma)(A'')$ is a model over the signature Σ, hence

- for each $s \in S$ we have $\mathbf{mod}(\rho \circ \sigma)(A'')(s) = A''(\rho \circ \sigma)(s) = A''(\rho(\sigma(s)))$ and
- for each $\omega \in \Omega$ we have $\mathbf{mod}(\rho \circ \sigma)(A'')(\omega) = A''(\rho \circ \sigma)(\omega) = A''(\rho(\sigma(\omega)))$.

R.H.S $(\mathbf{mod}(\sigma) \circ \mathbf{mod}(\rho))(A'') = \mathbf{mod}(\sigma)(\mathbf{mod}(\rho)(A''))$ is a model over the signature Σ, hence

- for each $s \in S$ we have $\mathbf{mod}(\sigma)(\mathbf{mod}(\rho)(A''))(s) = \mathbf{mod}(\rho)(A'')(\sigma(s)) = A''(\rho(\sigma(s)))$ and
- for each $\omega \in \Omega$ we have $\mathbf{mod}(\sigma)(\mathbf{mod}(\rho)(A''))(\omega) = \mathbf{mod}(\rho)(A'')(\sigma(\omega)) = A''(\rho(\sigma(\omega)))$.

since σ, ρ and A'' are arbitrary, we can conclude that $\mathbf{mod}(\rho \circ \sigma) = \mathbf{mod}(\sigma) \circ \mathbf{mod}(\rho)$ for all signature morphisms $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$. □

This completes the proof obligations that **mod** is indeed a valid functor.

3.2.4 Many-sorted equation logic - Satisfaction relation

We now define the satisfaction relation between our models and sentences. For this we will need the notions of assignments and term evaluation.

Given a signature $\Sigma = (S, \Omega)$, an algebra A in $\mathbf{mod}(\Sigma)$ and a family of variables X for the signature Σ, an assignment of X for A is a family $\alpha = (\alpha_s)_{s \in S}$ of functions $\alpha_s : X_s \to A(s)$. We normally drop the subscript s (where no confusion arises) and just write $\alpha : X \to A$ for such an assignment.

We now define term evaluation. Given a signature $\Sigma = (S, \Omega)$, an Algebra A in $\mathbf{mod}(\Sigma)$, a family of variables X, a term $t \in T_{\Sigma(X)}$ and an assignment $\alpha : X \to A$ then $A(\alpha)(t)$ is defined as:

1. $A(\alpha)(t) = \alpha_s(x)$ if $t = x$ with $x \in X_s$, $s \in S$,
2. $A(\alpha)(t) = A(\omega)$ if $t = n$ and $\omega = (n :\to s) \in \Omega$,
3. $A(\alpha)(t) = A(\omega)(A(\alpha)(t_1), \ldots, A(\alpha)(t_k))$ if $t = n(t_1, \ldots, t_k)$,
   $\omega = (n : s_1 \times \ldots \times s_k \to s) \in \Omega$,
   $k \geq 1$ and
   $t_i \in T_{\Sigma(X),s_i}$, $1 \leq i \leq k$.

Finally we define satisfaction. Given a signature Σ, a formula $\forall X.t = u$ in $\mathbf{sen}(\Sigma)$, and an algebra A in $\mathbf{mod}(\Sigma)$,

$$A \models_\Sigma \forall X.t = u$$

iff

$$\text{for all assignments } \alpha : X \to A, A(\alpha)(t) = A(\alpha)(u)$$

We now introduce the *Reduct Theorem* (Theorem 3.17) which is needed in order to prove that the satisfaction condition holds. The reduct theorem and it's proof along with the proof of the satisfaction condition has been lifted from [LEW97] and applied to the notions of an institution and Category Theory.

Theorem 3.17 (Reduct Theorem)**:** Let Σ and Σ' be signatures in **SIGN** where $\Sigma = (S, \Omega)$. Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism in **SIGN**. Let X be a family of variables for Σ and $t \in T_{\Sigma(X)}$ be a term. Finally, let M' be a model in $\mathbf{mod}(\Sigma')$ and $\alpha' : \sigma(X) \to M'$ be an assignment for M'. Then

$$(M'|_\sigma)(\alpha'|_\sigma)(t) = M'(\alpha')(\sigma(t))$$

where $(\alpha'|_\sigma) : X \to (M'|_\sigma)$ is the assignment for $(M'|_\sigma)$ defined by:

$$(\alpha'|_\sigma)_s(x) = \alpha'_{\sigma(s)}(\sigma(x))$$

for all $x \in X_s$, $s \in S$.

Proof. We prove the reduct theorem using induction on the structure of t. For each case we must prove $(M'|_\sigma)(\alpha'|_\sigma)(t) = M'(\alpha')(\sigma(t))$.

if $t = x$**,** $x \in X_s$:

$$\begin{array}{lcll} (M'|_\sigma)(\alpha'|_\sigma)(x) & = & (\alpha'|_\sigma)_s(x) & \text{Definition of term evaluation.} \\ & = & \alpha'_{\sigma(s)}(\sigma(x)) & \text{Definition of } (\alpha'|_\sigma). \\ & = & M'(\alpha')(\sigma(x)) & \text{Definition of term evaluation.} \end{array}$$

if $t = n$**,** $\omega = (n :\rightarrow s) \in \Omega$:

$$\begin{array}{lcll} (M'|_\sigma)(\alpha'|_\sigma)(n) & = & (M'|_\sigma)(\omega) & \text{Definition of term evaluation.} \\ & = & M'(\sigma(\omega)) & \text{Definition of a reduct.} \\ & = & M'(\alpha')(\sigma(n)) & \text{Definitions of term evaluation and term translation.} \end{array}$$

if $t = n(t_1, \ldots, t_k)$**,** $\omega = (n : s_1 \times \ldots \times s_k \rightarrow s) \in \Omega$**,** $k \geqslant 1$:

$$\begin{array}{ll} (M'|_\sigma)(\alpha'|_\sigma)(n(t_1, \ldots, t_k)) & \\ = (M'|_\sigma)(\omega)((M'|_\sigma)(\alpha'|_\sigma)(t_1), \ldots, (M'|_\sigma)(\alpha'|_\sigma)(t_k)) & \text{Definition of term evaluation.} \\ = (M'|_\sigma)(\omega)(M'(\alpha')(\sigma(t_1)), \ldots, M'(\alpha')(\sigma(t_k))) & \text{Induction hypothesis.} \\ = M'(\sigma(\omega))(M'(\alpha')(\sigma(t_1)), \ldots, M'(\alpha')(\sigma(t_k))) & \text{Definition a reduct.} \\ = M'(\alpha')(\sigma(n(t_1, \ldots, t_k))) & \text{Definitions of term evaluation} \\ & \text{and term translation.} \end{array}$$

□

Theorem 3.18: The satisfaction condition holds, i.e., for all signature morphisms $\sigma : \Sigma \rightarrow \Sigma'$ in **SIGN**, all models M' in $\mathbf{mod}(\Sigma')$ and all sentences φ in $\mathbf{sen}(\Sigma)$,

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

Proof. Let Σ, Σ' be signatures and $\sigma : \Sigma \rightarrow \Sigma'$ be a signature morphism in **SIGN**. Let $\forall X.t = u$ be an arbitrary sentence in $\mathbf{sen}(\Sigma)$ and M' be a model in $\mathbf{mod}(\Sigma)$. We must prove that:

$$(M'|_\sigma) \models_\Sigma \forall X.t = u \text{ iff } M' \models_{\Sigma'} \sigma(\forall X.t = u)$$

i.e.,

$$(M'|_\sigma) \models_\Sigma \forall X.t = u \text{ iff } M' \models_{\Sigma'} \forall \sigma(X).\sigma(t) = \sigma(u)$$

We prove each implication separately.

Case 1: $\Rightarrow$ Let $(M'|_\sigma) \models_\Sigma \forall X.t = u$ hold, by the definition of the satisfaction relation, we know that $(M'|_\sigma)(\alpha)(t) = (M'|_\sigma)(\alpha)(u)$ holds for all assignments $\alpha : X \rightarrow (M'|_\sigma)$. Let $\alpha' : \sigma(X) \rightarrow M'$ be an assignment for M'. We have to prove $M'(\alpha')(\sigma(t)) = M'(\alpha')(\sigma(u))$.

$$\begin{array}{lcll} M'(\alpha')(\sigma(t)) & = & (M'|_\sigma)(\alpha'|_\sigma)(t) & \textit{Reduct Theorem.} \\ & = & (M'|_\sigma)(\alpha'|_\sigma)(u) & \text{As } (\alpha'|_\sigma) \text{ is an assignment from } X \text{ into } (M'|_\sigma). \\ & = & M'(\alpha')(\sigma(u)) & \textit{Reduct Theorem.} \end{array}$$

Case 2: $\Leftarrow$ Let $M' \models_{\Sigma'} \forall \sigma(X).\sigma(t) = \sigma(u)$, by the definition of the satisfaction relation, we know that $M'(\alpha')(\sigma(t)) = M'(\alpha')(\sigma(u))$ for all assignments $\alpha' : \sigma(X) \to M'$. Let $\alpha : X \to (M'|_\sigma)$ be an assignment for $(M'|_\sigma)$. We have to prove $(M'|_\sigma)(\alpha)(t) = (M'|_\sigma)(\alpha)(u)$.

Now consider the assignment $\beta' : \sigma(X) \to M'$ defined by:

$$\beta'_{\sigma(s)}(\sigma(x)) = \alpha_s(x)$$

for any $x \in X_s, s \in S$.

$$\begin{array}{lcll} (M'|_\sigma)(\alpha)(t) & = & (M'|_\sigma)(\beta'|_\sigma)(t) & \text{As } (\beta'|_\sigma) = \alpha. \\ & = & M'(\beta')(\sigma(t)) & \textit{Reduct Theorem.} \\ & = & M'(\beta')(\sigma(u)) & \text{As } \beta' \text{ is an assignment from } \sigma(X) \text{ into } M'. \\ & = & (M'|_\sigma)(\beta'|_\sigma)(u) & \textit{Reduct Theorem.} \\ & = & (M'|_\sigma)(\alpha)(u) & \text{As } (\beta'|_\sigma) = \alpha. \end{array}$$

Since the signatures Σ, Σ', the signature morphism σ, the model M', and the sentence $\forall X.t = u$ are all arbitrary, we can conclude that the satisfaction condition holds. □

We have now defined the components of the institution `many-sorted equational logic` and proven the relevant properties that establish this as a valid institution.

3.3 Other examples of institutions

`Many-sorted equational logic` is a small example of a logic that forms an institution. Many other institutions exist, for example CSP forms various institutions [MR07].

We now outline the institutions necessary for the constructions within this thesis. For simplicity we disregard sort generation constraints in our discussions, as they currently play no role in our CSP-CASL encoding. This is justified by an implementation issue within HETS. In Sections 3.3.1 and 3.3.2 we outline the institution $PFOL^=$ and $FOL^=$, respectively. These both play a prominent role in the translation of CASL specifications into Isabelle/HOL code. We present finally the institution $SubPFOL^=$ (Section 3.3.3) which is build on top of $PFOL^=$ and is close to the institution underlying CASL ($SubPCFOL^=$).

3.3.1 $PFOL^=$

We present here some parts of the institution $PFOL^=$ (partial first order logic with equality) as it is defined in [Mos02]. We present only the category of signatures, the models, the set of sentences for a signature and an informal definition of the satisfaction relation. We do not present every definition of some shorthand notations along with the definitions of terms, variables and term evaluation. We do however use these notations. Such definitions can be found in [Mos02].

Signatures. A many-sorted signature $\Sigma = (S, TF, PF, P)$ in $PFOL^=$ consists of:

- a set of sorts symbols S,

- two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of total function symbols and partial function symbols, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each $(w,s) \in S^* \times S$ (constants are treated as functions with no arguments), and
- a family $P = (P_w)_{w \in S^*}$ of predicate symbols.

We use the following short hand notations within this chapter as they are defined in [Mos02]. We write $f : w \to s \in TF$ for $f \in TF_{w,s}$, $f : w \to ? s \in PF$ for $f \in PF_{w,s}$ and $p : w \in P$ for $p \in P_w$. Given a function $f : A \to B$, let $f^* : A^* \to B^*$ be its extension to finite strings. Given a finite string $w = s_1 \ldots s_n$ 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 signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, a 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 $\sigma^P_w : P_w \to P'_{\sigma^{S^*}(w)}$ for each $w \in S^*$.

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

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

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

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

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

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

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

Identities and composition are defined in the obvious way.

Concerning reducts, if $\sigma : \Sigma \to \Sigma'$ is a signature morphism with $\Sigma = (S, TF, PF, P)$, and M' is a Σ'-model, then $M'|_\sigma$ is the Σ-model M with:

- $M_s := M'_{\sigma^S(s)}$ for each sort symbol $s \in S$,
- $(f_{w,s})_M := (\sigma^F_{w,s}(f))_{M'}$ for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$,

- $(p_w)_M := (\sigma^P_w(p))_{M'}$ for each predicate symbol $p \in Pw$.

This is well defined since $\sigma^F_{w,s}$ preserves totality.

Given a Σ'-homomorphism $h' : M'_1 \to M'_2$, its reduct $h|_\sigma : M'_1|_\sigma \to M'_2|_\sigma$ is the homomorphism defined by

$$(h'|_\sigma)_s := h'_{\sigma^S(s)} \text{ for each sort symbol } s \in S.$$

It is easy to see that the reduct w.r.t. the identity is the identity, and that the reduct w.r.t. a composition is the composition of the reducts w.r.t. the signature morphisms that are composed. Thus, **mod** is a functor.

Sentences. A many-sorted atomic Σ-formula with variables in X is either (1) an application of a qualified predicate symbol to terms of appropriate sorts, (2) an existential equation between terms of the same sort, (3) a strong equation between terms of the same sort, or (4) an assertion about the definedness of a term:

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

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

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

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

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 $\neg \forall x : s \bullet \neg\varphi$.

Now a Σ-sentence is a closed many-sorted first-order Σ-formula (i.e., a many-sorted first-order Σ-formula in the empty set of variables). For further details see [Mos02].

Satisfaction relation. Even though the evaluation of a term w.r.t. a variable assignment may be undefined, the evaluation of a formula is always defined (and is either true or false). That is, we have a two-valued logic.

The application of a predicate symbol p to a sequence of argument terms holds w.r.t. a valuation $\nu : X \to M$ iff the values of all the terms are defined under $\nu^\#$ and give a tuple belonging to p_M. A definedness assertion concerning a term holds iff the value of the term is defined. An existential

equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined; thus both notions of equation coincide for defined terms.

A formula φ is satisfied in a model M (written $M \models \varphi$) iff it is satisfied w.r.t. all variable valuations into M.

This concludes the presentation of $PFOL^=$, for further details see [Mos02].

3.3.2 $FOL^=$

$FOL^=$ (first order logic with equality) is the sub-logic of $PFOL^=$ with the restriction that the set of partial functions symbols within the signature is empty.

3.3.3 $SubPFOL^=$

We present here some parts of the institution $SubPFOL^=$ (sub-sorted partial first order logic with equality) as it is defined in [Mos02]. We present only the signatures, models and the sentences.

Signatures. A *sub-sorted signature* $\Sigma = (S, TF, PF, P, \leqslant_S)$ consists of a many-sorted signature (S, TF, PF, P) (as defined in $PFOL^=$) together with a reflexive and transitive *sub-sort relation* $\leqslant_S \subseteq S \times S$. The relation $\leqslant_S$ extends point-wise to sequences of sorts. We drop the subscript S when it is obvious from the context. For a *sub-sorted signature* $\Sigma = (S, TF, PF, P, \leqslant_S)$ 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 $w \leqslant w_1$, $w \leqslant w_2$, $s_1 \leqslant s$, and $s_2 \leqslant s$. 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 $w \leqslant w_1$ and $w \leqslant w_2$.

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

We assume that the symbols used for injection, projection and membership are not used otherwise in Σ. In formulae, we also write $t \in s$ instead of $\in^s_{s'}(t)$ if s' is clear from the context.

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

1. $\mathtt{inj}_{s,s}(x) \stackrel{e}{=} x$ for $s \in S$ (identity),
2. $\mathtt{inj}_{s,s'}(x) \stackrel{e}{=} \mathtt{inj}_{s,s'}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leqslant_S s'$ (embedding-injectivity),
3. $\mathtt{inj}_{s',s''}(\mathtt{inj}_{s,s'}(x)) \stackrel{e}{=} \mathtt{inj}_{s,s''}(x)$ for $s \leqslant_S s' \leqslant_S s''$ (transitivity),

4. $\mathtt{pr}_{s',s}((\mathtt{inj}_{s,s'}(x)) \stackrel{e}{=} x$ for $s \leqslant_S s'$ (projection),
5. $\mathtt{pr}_{s',s}(x) \stackrel{e}{=} \mathtt{pr}_{s',s}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leqslant_S s'$ (projection-injectivity),
6. $\in^s_{s'}(x) \Leftrightarrow$ *def* $\mathtt{pr}_{s',s}(x)$ for $s \leqslant_S s'$ (membership),
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 $w \leqslant_S w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, $w'' = s''_1 \ldots s''_n$, $s', s'' \leqslant_S s$ (function-monotonicity),
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 $w \leqslant_S w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, $w'' = s''_1 \ldots s''_n$ (predicate-monotonicity).

Sentences. *Sub-sorted Σ-sentences* are ordinary *many-sorted $\hat{\Sigma}$-sentences*.

For further details of this institution see [Mos02].

3.4 Presentations

We present here the notions of the category of theories and the category of presentations as they are defined in [Mos02]. Let us fix an arbitrary institution $I = (\mathbf{SIGN}, \mathbf{sen}, \mathbf{mod}, \models)$. The simplest specifications over an arbitrary institution are just *theories* $T = \langle \Sigma, \Gamma \rangle$, where Σ is a signature in **SIGN** and $\Gamma \subseteq \mathbf{Sen}(\Sigma)$ is a set of sentences. We set $\mathbf{Sig}(T) = \Sigma$ and $\mathbf{Ax}(T) = \Gamma$. *Theory morphisms* $\sigma\langle \Sigma, \Gamma \rangle \to \langle \Sigma', \Gamma' \rangle$ are those signature morphisms $\sigma : \Sigma \to \Sigma'$ for which $\Gamma' \models_{\Sigma'} \sigma(\Gamma)$, that is, axioms are mapped to logical consequences. By inheriting composition and identities from **SIGN**, we obtain a category **TH** of theories. It is easy to extend **sen** and **mod** to start from **TH** by putting $\mathbf{sen}(\langle \Sigma, \Gamma \rangle) = \mathit{Sen}(\Sigma)$ and letting $\mathbf{mod}(\langle \Sigma, \Gamma \rangle)$ be the full sub-category of $\mathbf{mod}(\Sigma)$ induced by the class of those models M satisfying Γ. The category **PRES** of presentations (also called flat specifications) is the full sub-category of theories having finite sets of axioms.

3.5 Institutions representations

In order to relate different institutions, we use the notion of institution representations (also called institutions comorphisms). These allow us to encode one institution within another institution and are implemented within the tool HETS [MML07] (see Section 4.2).

An institution representation from an institution I to an institution J is a triple which consists of three components: a functor Φ, a natural transformation α, and a natural transformation β.

The functor Φ translates I-signatures into J-presentations. This means that the signatures of I are translated into signatures of J plus a finite set of axioms. For example in the translation of $PFOL^=$(partial first order logic with equality) to $FOL^=$(first order logic with equality), partial functions can be encoded as total functions provided an extra undefined element is added to each set of sort symbols and also a definedness function on each sort is added. Axioms are then needed

to control how the definedness functions behave (e.g., definedness should hold for every element of the sort expect the undefined element). This is why *J*-presentations are required and not just *J*-signatures.

The natural transformation α translates *I*-sentences to *J*-sentences. This is similar to the functor **sen** from institutions. The natural transformation β translates *J*-models to *I*-models, which again is similar to the functor **mod** from institutions.

Formally, given institutions $I = (\mathbf{SIGN}^I, \mathbf{sen}^I, \mathbf{mod}^I, \models^I)$ and $J = (\mathbf{SIGN}^J, \mathbf{sen}^J, \mathbf{mod}^J, \models^J)$, a simple institution representation $\mu = (\Phi, \alpha, \beta) : I \to J$ consists of

- a functor $\Phi : \mathbf{SIGN}^I \to \mathbf{PRES}^J$,
- a natural transformation $\alpha : \mathbf{sen}^I \to \mathbf{sen}^J \circ \Phi$,
- a natural transformation $\beta : \mathbf{mod}^J \circ \Phi^{op} \to \mathbf{mod}^I$,

such that the representation condition is satisfied: for every signature Σ in $\mathbf{SIGN}^I$, model M' in $\mathbf{mod}^J(\Phi(\Sigma))$ and sentence φ in $\mathbf{sen}^I(\Sigma)$ the following holds:

$$M' \models^J_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi) \Leftrightarrow \beta_\Sigma(M') \models^I_\Sigma \varphi$$

The above means that for each signature Σ in $\mathbf{SIGN}^I$, Φ maps Σ to a *J*-presentation $\Phi(\Sigma)$ in $\mathbf{PRES}^J$. For every signature morphism $\sigma : \Sigma \to \Sigma'$ in $\mathbf{SIGN}^I$, Φ maps σ to a presentation morphism $\Phi(\sigma) : \Phi(\Sigma) \to \Phi(\Sigma')$ in $\mathbf{PRES}^J$, which translates *J*-presentations over Σ into presentations over Σ'.

For every signature Σ in $\mathbf{SIGN}^I$, there is a sentence translation $\alpha_\Sigma : \mathbf{sen}^I(\Sigma) \to \mathbf{sen}^J(\Phi(\Sigma))$ which translates *I*-sentences to *J*-sentences.

For every signature Σ in $\mathbf{SIGN}^I$, there is a model translation $\beta_\Sigma : \mathbf{mod}^J(\Phi(\Sigma)) \to \mathbf{mod}^I(\Sigma)$ which translates *J*-models to *I*-models. β is a contravariant model translation, similar to that of **mod** of an institution.

The representation condition has the same idea as the satisfaction condition of Institutions. This ensures that satisfaction with respect to the satisfaction relation, is preserved across translation of sentences from institution *I* to institution *J* and translation of models from institution *J* to institution *I*.

As α is a natural transformation, any signature morphism $\sigma : \Sigma \to \Sigma'$ in $\mathbf{SIGN}^I$ makes the following diagram commute:

$$\begin{array}{ccc} \mathbf{sen}^I(\Sigma) & \xrightarrow{\alpha_\Sigma} & \mathbf{sen}^J(\Phi(\Sigma)) \\ \downarrow{\scriptstyle \mathbf{sen}^I(\sigma)} & & \downarrow{\scriptstyle \mathbf{sen}^J(\Phi(\sigma))} \\ \mathbf{sen}^I(\Sigma') & \xrightarrow{\alpha_{\Sigma'}} & \mathbf{sen}^J(\Phi(\Sigma')) \end{array}$$

Similarly, as β is a natural transformation, any signature morphism $\sigma : \Sigma \to \Sigma'$ in $\mathbf{SIGN}^I$ makes the following diagram commute:

$$\begin{array}{ccc} \mathbf{mod}^I(\Sigma) & \xleftarrow{\beta_\Sigma} & \mathbf{mod}^J(\Phi(\Sigma)) \\ \uparrow{\scriptstyle \mathbf{mod}^I(\sigma)} & & \uparrow{\scriptstyle \mathbf{mod}^J(\Phi(\sigma))} \\ \mathbf{mod}^I(\Sigma') & \xleftarrow{\beta_{\Sigma'}} & \mathbf{mod}^J(\Phi(\Sigma')) \end{array}$$

We have presented here the notion of institutions representations that allows one to encode one institution within another institution. Two institution representations from institutions *I* to *J* and *J* to *K* can be composed together to form an institution representation from institution *I* to institution *K*. This allows encodings of one institution within another institution where no direct representations are available.

3.6 Examples of institution representations

We now present the institutions representations used in Section 4.2 that allows HETS to encode CASL specifications within the theorem prover Isabelle/HOL. First we present the institution representation that allows one to encode *SubPFOL*$^=$ in *PFOL*$^=$ – Section 3.6.1. Followed by the institution representation for encoding *PFOL*$^=$ in *FOL*$^=$ – Section 3.6.2. Both these institution representations can be composed together to form an encoding from CASL to Isabelle/HOL[2].

[2]For simplicity we disregard sort generation constraints in our discussions, as they currently play no role in our CSP-CASL encoding. This is justified by an implementation issue within HETS.

3.6.1 Representing $SubPFOL^{=}$ in $PFOL^{=}$

We present here the translation of the institution $SubPFOL^{=}$ to $PFOL^{=}$ as it is defined in [Mos02]. The translation is trivial, since the institution $SubPFOL^{=}$ is defined in terms of the institution $PFOL^{=}$ (see Section 3.3.3):

- *Signatures.* A signature Σ is mapped to the presentation $(\hat{\Sigma}, \hat{J}(\Sigma))$.
- *Models.* Model translation is the identity.
- *Sentences.* Sentence translation is the identity.
- *Satisfaction.* The representation condition follows immediately.

3.6.2 Representing $PFOL^{=}$ in $FOL^{=}$

We present here some parts of the translation of the institution $PFOL^{=}$ to $FOL^{=}$ as it is defined in [Mos02]. Specifically, we present the signature translation and the sentence translation.

The main idea is to use a definedness predicate to divide each carrier set into defined and undefined elements. The defined elements represent ordinary values, while the undefined elements all represent the undefined. Partial functions thus can be totalised: they possibly yield an undefined element. We specify that there is at least one undefined element $\bot$, however, it may be not the only one.

Signatures. A $PFOL^{=}$-signature $\Sigma = (S, TF, PF, P)$ is translated to a $FOL^{=}$-presentation having the signature

$$Sig(\Phi(\Sigma)) = (S, TF \uplus PF \uplus \{\bot\colon s \mid s \in S\}, P \uplus \{D : s \mid s \in S\})$$

and the set of axioms $Ax(\Phi(\Sigma))$:

1. $\exists\, x : s \bullet D_s(x)$ for each $s \in S$,
2. $\neg D_s(\bot_s)$ for each $s \in S$,
3. $D_s(f(x_1, \ldots, x_n)) \Leftrightarrow \bigwedge_{i=1\ldots n} D_{s_i}(x_i)$ for each function symbol $f : s_1 \ldots s_n \to s \in TF$,
4. $D_s(g(x_1, \ldots, x_n)) \Rightarrow \bigwedge_{i=1\ldots n} D_{s_i}(x_i)$ for each partial function symbol $g : s_1 \ldots s_n \to ? s \in PF$,
5. $p(x_1, \ldots, x_n) \Rightarrow \bigwedge_{i=1\ldots n} D_{s_i}(x_i)$ for each predicate symbol $p : s_1 \ldots s_n \in P$.

D plays the role of a definedness predicate: the elements inside D are called defined, those outside D are called undefined. The axioms in the signature translation state that there is at least one defined element (1), that $\bot$ is an undefined element (2), total functions are indeed total (3) and all functions ((3), (4)) and predicates (5) are strict.

As we want only a single undefined element in each sort, we add the extra axiom $\neg(x = \bot_s) \Rightarrow D_s(x)$ for each sort $s \in S$ as defined in Section 4.1.6 of [Mos02]. This guarantees that the $\bot$ element

is the only undefined element in each sort. This extra axiom along with axiom (2) simplifies to a new axiom $\neg(D_s(x)) \Leftrightarrow (x = \bot_s)$ for each sort $s \in S$.

Theorem 3.19: Given a Signature $\Sigma = (S, TF, PF, P)$

$$\neg D_s(\bot_s) \wedge (\neg(x = \bot_s) \Rightarrow D_s(x)) \text{ iff } \neg(D_s(x)) \Leftrightarrow (x = \bot_s)$$

for each $x \in s$, $s \in S$.

Proof. We have proven this using the theorem prover SPASS [WBH+02]. □

Sentences. The sentence translation keeps the structure of the sentences and maps strong and existential equality to appropriate circumscriptions using the definedness predicate D. Definedness is mapped to D, and quantifiers are relativised to the set of all defined elements [Mos02].

Formally, a Σ-sentence φ (in $PFOL^{=}$) is translated to the $\Phi(\Sigma)$-sentence $\alpha_\Sigma(\varphi)$:

- $\alpha_\Sigma(def\,(t)) = D_s(t)$,
- $\alpha_\Sigma(t_1 = t_2) = ((D_s(t_1) \vee D_s(t_2)) \Rightarrow t_1 = t_2)$,
- $\alpha_\Sigma(\varphi \wedge \psi) = \alpha_\Sigma(\varphi) \wedge \alpha_\Sigma(\psi)$,
- $\alpha_\Sigma(t_1 \stackrel{e}{=} t_2) = t_1 = t_2 \wedge D_s(t_1)$,
- $\alpha_\Sigma(p(t_1, \ldots, t_n)) = p(t_1, \ldots, t_n)$,
- $\alpha_\Sigma(F) = F$,
- $\alpha_\Sigma(\varphi \Rightarrow \psi) = \alpha_\Sigma(\varphi) \Rightarrow \alpha_\Sigma(\psi)$,
- $\alpha_\Sigma(\forall x : s \bullet \varphi) = \forall x : s \bullet D_s(x) \Rightarrow \alpha_\Sigma(\varphi)$,

For further details of this institution representation see [Mos02].

Chapter 4

Tools involved

Contents

CSP-CASL-Prover re-uses existing technology and tools where appropriate. In this section we will explain what these tools are and what they do.

4.1 Isabelle/HOL

Isabelle [NPW02, Pau94] is a widely used, generic interactive theorem prover implemented in ML [MTH90]. It is developed at Cambridge University and the Technical University of Munich. Isabelle is a command line tool which is closely integrated with *Proof General* [Asp00, Pro] to provide a powerful graphical user interface. Isabelle has many variants, each one instantiated with a different logic. The most popular being Isabelle/HOL, the Isabelle variant that has been instantiated with Higher Order Logic (HOL). During this thesis we use the variant Isabelle/HOL.

Isabelle has an input language similar to ML which consists of *commands* and *proof commands*. *Commands* allow one to extend the logic in various ways. For example, by adding new types, data structures and functions. While *proof commands* allow one to interface with Isabelle in order to prove lemmas and theorems.

Isabelle has an outer syntax and an inner syntax. The outer syntax is where the *commands* and *proof commands* live, whereas the inner syntax is used within such *commands* and *proof commands* to specify formulae of the particular logic that Isabelle has been instantiated with. Hence the inner syntax changes depending on the logic in use and is surrounded by double quotes (these can be omitted if they are quoting a single term – which is often the case).

Using the techniques described here the tool CSP-Prover [IR05, IR06, IR] (see Section 4.3) provides a deep encoding of the process algebra CSP in Isabelle/HOL. HETS [MML07] on the other hand produces a shallow encoding of CASL in Isabelle/HOL (see Section 4.2).

4.1.1 Theorems and proofs

The purpose of Isabelle is to aid the user in interactively proving theorems on mathematical formulae of a particular domain.

Theorems are entered into Isabelle via the *command*

```
theorem name [opt]: "formula"
```

where `name` is an optional argument used to later reference the theorem, `opt` are some options which control how Isabelle uses the theorem once it has been established and `formula` is the mathematical formula to be established as a theorem. This theorem then needs to be proven in Isabelle. Lemmas can also be introduced in a similar way by replacing the keyword `theorem` by the keyword `lemma`. Isabelle displays the proof goals which need to be discharged in order for that theorem to be proven. For example the command

```
theorem T1: "a+b = b+a"
```

creates a new theorem with the name `T1` and the formula `a+b = b+a`. Here we can see that no options have been specified and that `a+b = b+a` is a formula of the inner syntax of the particular logic in use. Isabelle displays the following proof goal once the above theorem command has been issued.

```
proof (prove): step 0

fixed variables: a, b

goal (lemma (T1), 1 subgoal):
 1. a + b = b + a
```

Here the goal only has a single sub-goal which is to prove `a+b = b+a`. The variables `a` and `b` have been fixed so that we can work with them, whereas in the theorem they where arbitrary. The display of the goals can be more complex where essentially the sub-goals are prefixed by a list of local assumptions which can be used to prove the goal.

To prove such a theorem, *proof commands* are issued which modify the proof state by transforming goals into other goals (or possibly many sub-goals). A goal is discharged if it is transformed into the truth value `True`. A theorem is proven when all of its proof obligations are discharged. Previously established theorems can be used within further proofs as new rules (see Section 4.1.4). Proof commands can be combined in various ways to form tactics (see Section 4.1.5), which can ease the burden of discharging proof goals.

Isabelle sometimes uses schematic variables inside goals where they usually can be considered as links between various goals. A *schematic variable* or *unknown* always has a `?` as its first character. Logically, an *unknown* is a free variable. But it may be instantiated by another term during the proof process. For example, the mathematical theorem $x = x$ is represented in Isabelle as $?x = ?x$, which means that Isabelle can instantiate it arbitrarily. This is in contrast to ordinary variables, which remain fixed [NPW02]. We normally do not use schematic variables explicitly, however the proof machinery uses the extensively.

4.1.2 Theory files

Isabelle uses theory files which are scripts of Isabelle commands and proof commands. Such theory files can use theorems, proofs, data structures and functions written in other theory files. This brings in a concept of modularity and code re-use to Isabelle. Isabelle/HOL comes equipped with an extensive library of existing theory files which the user can make use of. For example the theory file `nat.thy` contains the formalisation of the natural numbers and operations on them in Isabelle/HOL.

A theory file has the general structure

```
theory T
imports B1 ... Bn
begin
  declarations, definitions, and proofs
end
```

where `T` is the theory's name, `B1 ... Bn` are the names of existing theory files upon which this theory file is based and `declarations, definitions, and proofs` are the newly introduced types, functions and proofs.

4.1.3 Commands

Commands allow one to extended the logic and can be seen as the functional programming language of Isabelle/HOL. For example, one can add new data structures, types and function definitions to Isabelle/HOL. This allows one to accommodate for the particular area of interest. Here we will give an overview of some of the most frequently used commands in the project.

New types can be introduced in various ways depending on what command is used. To introduce a new data type in Isabelle/HOL the command

```
datatype (a1, ... , an) t =
  C1 T11 ... T1k1 | ... | Cm Tm1 ... Cm TmKm
```

may be issued where `t` is the type name of the new data type, `ai` are type variables (parameters for the type), `Ci` are distinct constructors and `Tij` are types. Recursive types are allowed, i.e., when the type `t` is used on the R.H.S as some of the types `Tij`. Polymorphic data types are created by using the type variables `ai` on the R.H.S in place of some of the types `Tij`. The parenthesis around the type parameters can be omitted in the case of zero or one parameter. For example, to add a new data type with the name `Num`, the command

```
datatype Num = N nat | I int
```

can be issued. This will create a new type which is the sum of natural numbers and integers. `N` and `I` are user chosen type constructors while `nat` and `int` are the built in types of natural numbers and integers respectively. Such a datatype declaration comes with built in induction tactics within Isabelle/HOL, see section 4.1.5 for details.

Functions whose parameter is defined by a `datatype` command may be defined using primitive recursion via the `primrec` command. This allows on to define functions which are based on the recursive nature of types defined via the `datatype` command.

We can now define addition on our new type `Num` by creating a new function `plus:: Num => Num => Num` such that a natural number will be returned only if both arguments are natural numbers, else an integer will be returned. This can be defined by primitive recursion as

```
consts plus:: "Num => Num => Num"
primrec "plus (N n) x = (case x of (N m) => N (n + m)
                                 | (I m) => I ((int n) + m))"
        "plus (I n) x = (case x of (N m) => I (n + (int m))
                                 | (I m) => I (n + m))"
```

where a number of other commands (namely, `consts` and `case`) have also been used. These commands will be explained in the next paragraph. When using primitive recursion (the command `primrec`) in Isabelle/HOL, it is required that all functions terminate. Non-terminating functions can cause inconsistencies in Isabelle/HOL. Due to the construction of Isabelle/HOL it is impossible to write a definition using the `primrec` command which Isabelle/HOL considers to be valid that is actually a definition of a non-terminating function. Isabelle does this by requiring that one parameter has been declared via a `datatype` command and that there is a single equation for each constructor of the datatype. This parameter must be called in a way that is structurally smaller. This with a few other restrictions ensures that the function terminates in all cases.

The `consts` command extends the signature by a new symbol. Here we declare the function `plus` which extends the signature with a new function. This function takes as parameters two values of type `Num` and returns a value of type `Num`. The `case` command allows for a definition based on the analysis of the constructors of a data type. We use this here to apply case analysis based on the constructor of the second parameter. The function `int` is a function that comes with Isabelle/HOL and converts a natural number (type `nat`) into an integer (type `int`).

4.1.4 Proof commands

After a theorem or lemma command (see Section 4.1.1) has been issued, Isabelle changes into its proof mode. In this mode, proof obligations may be discharged by the application of proof commands.

Whilst developing a proof, the proof command `sorry` allows one to explore the proof structure. The proof command `sorry` instantly dismisses all goals and completes the proof. The theorem can be used later on in the script as if it has been proven. This can be very useful for top down development, where lower lemmas and theorems can be assumed to hold using the proof command `sorry` and later completed. All `sorry` proof commands must be replaced by real proofs in order to be sure that the theorem actually holds.

Another useful proof command is `defer`, which moves the first sub-goal to the end of the list of sub-goals. This is useful when the user wants to prove the most complex sub-goal first. For example, when the user is sure that the first sub-goal holds, but is unsure whether others hold. The

hardest sub-goals can be investigated first, if this sub-goal cannot be proven then there is no point in continuing the rest of the proof. The proof command `prefer n` is similar and moves the nth sub-goal to the start of the list.

So far we have looked at proof commands that "organise" proofs, i.e., proof commands that re-order the sub-goal list and allows one to explore the proof structure. We now discuss proof commands that actually transform proof goals and allows one to make progress within proofs.

One of the most frequently used proof command is the `apply` command which is used to apply many different proof methods to the proof state. One such popular use of this is

```
apply(auto)
```

which tries to automatically prove all sub-goals. The `auto` proof method mainly tries to simplify all sub-goals by invoking the simplifier on each sub-goal in turn. This is a powerful method and one of the main proof tools of Isabelle/HOL.

Isabelle/HOL has a large collection of simplification rules called the *simpset*. When the simplifier is invoked, the rules in the *simpset* are applied (almost blindly) from left to right to the first sub-goal and the sub-goal's assumptions. These rules are used as rewrite rules during the term rewriting process. This is a powerful method and can automatically prove many goals. The user can invoke the simplifier via the `auto` method via the proof command

```
apply(simp modifier list)
```

where the user may specify modifiers that control how the simplifier behaves. For example the user can specify a list of rules to be temporarily added the *simpset* via the proof command

```
apply(simp add: rules)
```

where `rules` are a list of the names of previously established lemmas and theorems to be temporally added to the *simpset*.

Rules can also be temporarily deleted from the *simpset* in a similar manner via the modifier `del`.

The simplifier applies rules almost blindly from left to right, as a result the user can easily add new rules that cause non-termination of the simplifier. For example, If the formula `x=y ==> y=x` was somehow added to the *simpset*, maybe because this formula was a result of another formula which had been re-written using a third rewrite rule, then this would easily lead to non-termination. It is useful in these cases to remove the offending rules from the `simpset`.

When a lemma or theorem is created the user can specify several options (see Section 4.1.1). For instance, one such option is the *simp* option that causes the lemma or theorem to be added to the *simpset* from that point onward.

When a lemma or theorem has been successfully proven (when all goals have been discharged), the proof command `done` should be issued which causes Isabelle to exit the proof mode and return to normal behaviour where *commands* can be issued again.

4.1.5 Tactics

Tactics allow the user to apply many proof steps in complex ways by issuing a single proof command. Tactics work by collecting together proof commands and applying them in certain combinations. Isabelle/HOL also comes with some built in tactics. One such tactic is induction over types declared using the `datatype` command. Its name is `induct_tac`. It is applied by the command `apply(induct_tac x)` where `x` is the variable to apply induction on. When applied this *proof command* will transform the first sub-goal into many sub-goals, one for each constructor of the data type.

The following Isabelle/HOL code proves our new function `plus` (see Section 4.1.3) to be commutative. This uses the induction tactic and the `auto` proof method to easily prove the theorem.

```
1 theorem comm: "plus a b = plus b a"
2 apply(induct_tac a)
3 apply(induct_tac b)
4 prefer 3
5 apply(induct_tac b)
6 apply(auto)
7 done
```

The interesting steps here are the induction steps on the variables `a` and `b`. After the fourth proof command (Line 5) has been issued, the proof obligation has been reduced to a finite case distinction over the forms of the variables `a` and `b`. Each variable has one of two forms, either the constructor `Nat` followed by a natural number or the constructor `Int` followed by an integer. Hence there are only four possible cases. Hence we obtain after applying the final induction tactic (Line 5) the following four sub-goals, where `int`, `nat`, `inta` and `nata` can be regarded as local variables (where Isabelle/HOL has generated the names):

```
goal (theorem (comm), 4 subgoals):
 1. !!int nat. plus (I int) (N nat) = plus (N nat) (I int)
 2. !!int inta. plus (I int) (I inta) = plus (I inta) (I int)
 3. !!nat nata. plus (N nat) (N nata) = plus (N nata) (N nat)
 4. !!nat int. plus (N nat) (I int) = plus (I int) (N nat)
```

The `auto` proof method is now capable of solving all 4 sub-goals by applying the definitions of function the `plus` and simplifying the results – Line 6. Line 7 completes the proof with the *proof command* `done`.

One can also combine several proof steps to form tactics. For instance proof methods can be combined sequentially to form a tactic by separating them with a comma. This has been used in the proof of commutativity of `plus` above, where `apply(induct_tac b, auto)` is the proof command that applies the tactic `(induct_tac b, auto)` to the proof state. This *proof command* causes Isabelle to first apply induction on the first sub-goal followed by an attempt to automatically solve the sub-goals. We could have used the symbol | instead of the comma, this would have caused Isabelle to first try and apply induction and only if this fails does Isabelle try to automatically solve the sub-goals. However, this would not be a good idea in this case as once

we have performed induction we would like Isabelle to try and automatically solve the sub-goals, hence sequencing of proof commands is the correct choice.

Another useful way of creating tactics is to append a + at the end of the *proof command* `apply` . For instance the proof command

```
apply(simp)+
```

invokes the simplifier which solve the first sub-goal and then repeatedly invokes the simplifier upon the remaining sub-goals until the application of the simplifier fails at which point the application of the tactic will be complete. This can be very useful in combination with the tactic formed using a comma to repeatedly apply a sequential list of proof commands. This can result in a significantly shorter proof script.

4.1.6 Philosophy of Isabelle

Isabelle is based on a small core of axioms, all other functionality and theorems are derived from this small core. Hence as long as the core is consistent then everything which is derived from it will also be consistent. There are a few commands for which this doesn't follow. Thus, these commands must be used with caution, for example the command

```
  axioms A1: "P"
```

states that the formula `P` always holds. `A1` is a label so we can later reference the axiom. This can be used to state that false formulae hold, for instance the command

```
  axioms False_Axiom: "False"
```

states that the formulae `False` holds, this axiom has the name `False_Axiom`. It is now possible to prove every formulae true in Isabelle/HOL. This is done by proving true formulae true and when one has a false formulae, first reduce it to the sub-goal `False` and then use the axiom `False_Axiom` to prove it holds.

4.2 HETS

The Heterogeneous Tool Set (HETS) [MML07] is a parsing, static analysis and proof management tool for various specification languages centred around CASL. HETS is an established tool and is developed mainly at Bremen University where work continues to add new features. HETS is an interactive system with a graphical user interface, recently command line support has been called for.

HETS is a system which keeps track of open proof goals (which are caused by theorems which have not yet been proven) and closed proof goals (which are caused by theorems which have been proven or disproven). HETS reads a specification text possibly including open proof goals, parses it, and then performs static analysis on it. After this, a graph of the structure of the specification is displayed in it's user interface. Within this graph, the user can see which goals are open and which

are closed. The user can also perform various operations on each node in the graph, for instance requesting the theory of such a node, HETS will then display the relevant information.

HETS can interface with different theorem provers, including Isabelle and SPASS [WBH+02]. HETS can call theorem provers with proof obligations and axioms given by specifications in order to discharge open proof goals. This allows the user to pass control over to a theorem prover and use that theorem prove to discharge open proof obligations. Figure 4.1 shows a screen-shot of HETS running on a specification containing open proof goals. There are two windows shown, one of which is the proof window that allows one to interface with various theorem provers. Various open proof goals can be seen in the top left of the proof window, a list of theorem provers can be seen in the centre right and various axioms and theorems which can be pass to the theorem prover can be seen in the lower half. Once the proof obligation is discharged, control returns to HETS and HETS records what was proven by changing the colours of the nodes on the graph to green.

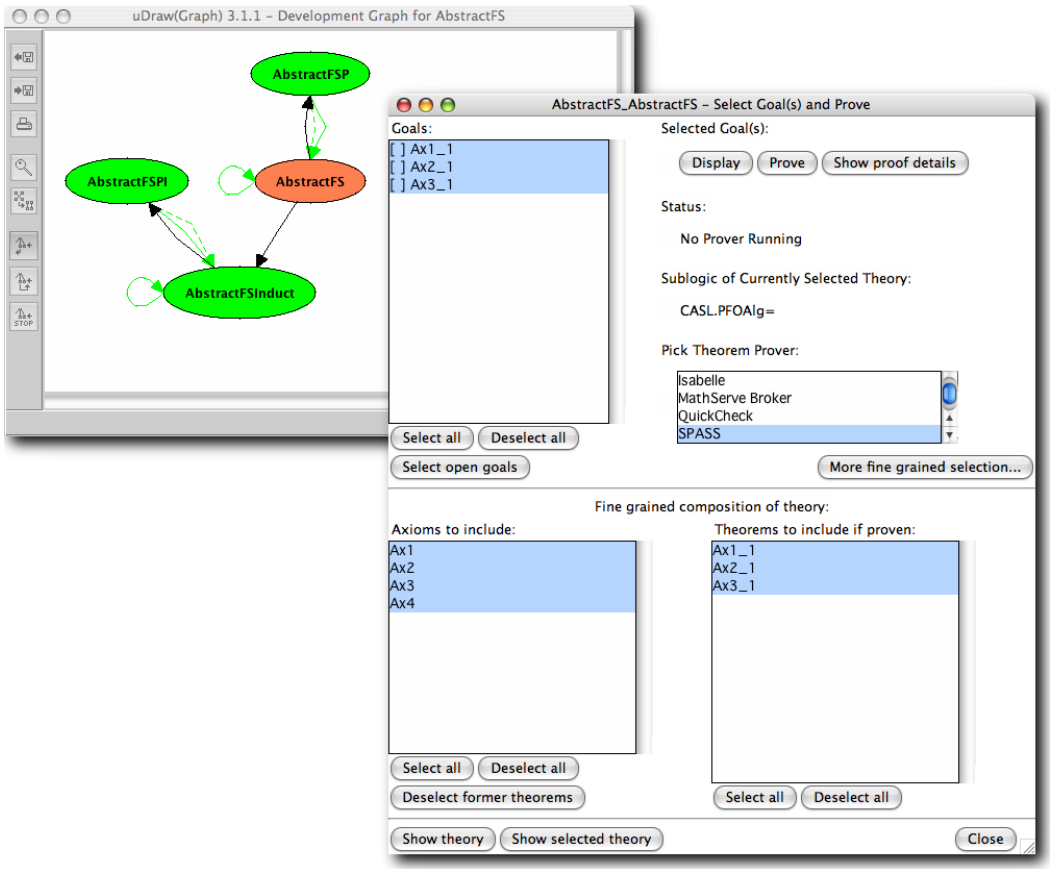

Figure 4.1: Screen-shot of HETS.

HETS is also capable of taking a specification written in one specification language and then transforming it into a specification written in another language, whilst preserving the semantics of the original specification. An important instance of this is the ability to transform CASL specifications into suitable code for use in the theorem prover Isabelle/HOL. We mainly use HETS as an input/out-

put tool, loading specifications written in CASL and encoding them into Isabelle/HOL theory files. This is a non-trivial encoding and CSP-CASL-Prover exploits this functionality heavily.

Figure 4.2 shows the most important sub-logics and comorphisms which are currently supported by HETS. Each of these sub-logics is implemented within HETS as an institution (see Section 3.1). The arrows are the institution representations (comorphisms) which allows one to transform specifications from one institution into specifications from another institution (see Section 3.5). Arrow can be composed thus forming many transformations between various sub-logics. We use the transformation along the path "CASL2PCFOL" → "PCFOL2CFOL" → "CFOL2IsabelleHOL" to translate our CASL specifications (the data part of our CSP-CASL specifications) into Isabelle/HOL code (see Section 6.4).

4.2.1 The Isabelle encoding of CASL

HETS is able to automatically transform a CASL specification into an Isabelle/HOL theory file whilst preserving the semantics of the original CASL specification. The logic of CASL is *Sub-PCFOL*$^{=}$ (sub-sorted partial first order logic with sort generation constraints and equality) and the logic of Isabelle/HOL is obviously *HOL* (a higher order logic). HETS provides us with the following automatic translation from CASL to Isabelle/HOL: CASL to *PCFOL*$^{=}$, *PCFOL*$^{=}$ to *CFOL*$^{=}$, *CFOL*$^{=}$ to Isabelle/HOL.

There are other translations available, but we have chosen to use this translation as the axioms and functions which its provides are the most appropriate for use with our construction.

We describe the encoding using a running example [1] performed on the CASL specification shown in Figure 4.3. Here we have a specification called "SP" which has two sorts `S` and `T`. Sort `S` is a sub-sort of sort `T`. There is also a partial function from sort `S` to sort `T`.

The first translation from CASL to *PCFOL*$^{=}$ translates specifications written using the logic *Sub-PCFOL*$^{=}$ into specifications using the logic *PCFOL*$^{=}$. This translation removes sub-sorting from the specification, this is done by encoding the sub-sorting using injection and projection functions (see Section 3.6.1) for details.

Figure 4.4 shows the resulting specification after the specification in Figure 4.3 has been translated from CASL to *PCFOL*$^{=}$. We still have two sorts `S` and `T`, however they are no longer in a sub-sort relation. We still have our partial function `f` from `S` to `T`. Additionally we have two new functions: an injection function gn_inj_S_T from `S` to `T` and a partial projection function gn_proj_T_S from `T` to `S`. We also have new axioms that govern how the injection and projections functions behave. These new functions and axioms are exactly the same functions and axioms discussed in Section 3.6.1.

The second translation *PCFOL*$^{=}$ to *CFOL*$^{=}$ translates specifications written using the logic *PCFOL*$^{=}$ into specification using the logic *CFOL*$^{=}$. This translation encoding a special element in each sort called the undefined element (bottom) and totalises partial functions using such undefined elements.

[1] For the running example we have use HETS version 0.8, for the rest of the thesis we have used HETS version 0.6. Both versions are identical for our purposes up to the naming scheme of the functions, predicates and axioms.

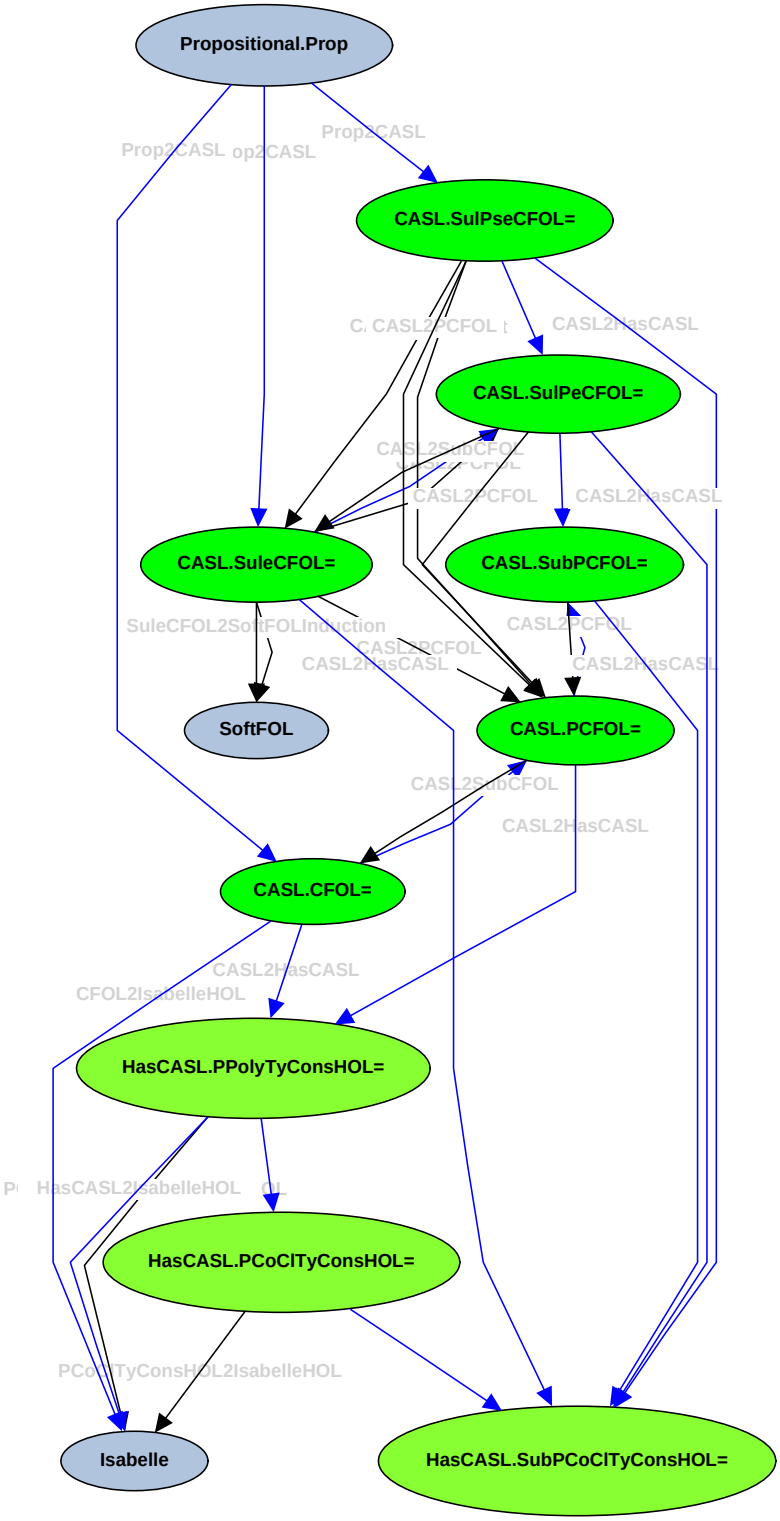

Figure 4.2: Graph of the most important sub-logics currently supported by HETS, together with their comorphisms [Mos06].

```
spec SP =
  sort S < T
  op f: S ->? T
end
```

Figure 4.3: A small example of a CASL specification (with no formulae) in the logic *SubPFOL*$^{=}$.

```
sorts S, T
op f : S ->? T
op gn_inj_S_T : S -> T
op gn_proj_T_S : T ->? S

forall x, y : S . gn_inj_S_T(x) =e= gn_inj_S_T(y) => x =e= y
                                        %(ga_embedding_injectivity)%

forall x, y : T . gn_proj_T_S(x) =e= gn_proj_T_S(y) => x =e= y
                                       %(ga_projection_injectivity)%

forall x : S . gn_proj_T_S(gn_inj_S_T(x)) =e= x %(ga_projection)%
```

Figure 4.4: Specification after encoding of sub-sorting in the logic *PFOL*$^{=}$.

Figure 4.5 shows the resulting specification after the *PCFOL*$^{=}$ to *CFOL*$^{=}$ translation has been applied to the specification in Figure 4.4. We still have two sorts (`S` and `T`) and a pair of injection and projection functions, however the projection function is now a total function. There are new undefined elements `gn_bottom_S` and `gn_bottom_T` along with two definedness predicates `gn_defined` for sorts `S` and `T`. Extra axioms have also been added, that govern how the predicates and undefined elements behave. The axioms `ga_embedding_injectivity`, `ga_projection_injectivity` and `ga_projection` have been translated to work with the new undefined elements and predicates. The new predicates, undefined elements, axioms and translated axioms are the same as in Section 3.6.2.

The final translation *CFOL*$^{=}$ to Isabelle/HOL translates the syntax of *CFOL*$^{=}$ into the correct syntax for Isabelle/HOL. This translation also encodes sort generation constraints into *FOL*$^{=}$ (although, currently, this plays no role as we do not use sort generation constraints). As Isabelle/HOL has (to the best of our knowledge) not yet been formalised in the framework of an institution, this translation lacks the mathematical rigour of the previous translations.

Figure 4.6 shows the resulting Isabelle/HOL theory file [2] after the *CFOL*$^{=}$ to Isabelle/HOL translation has been applied to the specification in Figure 4.5. This is basically just a syntactic translation from the logic *CFOL*$^{=}$ to Isabelle/HOL code. For instance the sort declarations have been changed into `typedecl` commands for Isabelle/HOL.

This translation has produced a shallow encoding of CASL inside Isabelle/HOL (no deep-encoding

[2] A header and a line stating the point between the signature extension and the axioms needs to be added to this code in order for it to be a valid Isabelle/HOL theory file.

```
sorts S, T
op f : S -> T
op gn_bottom_S : S
op gn_bottom_T : T
op gn_inj_S_T : S -> T
op gn_proj_T_S : T -> S
pred gn_defined : S
pred gn_defined : T

. exists x : S . gn_defined(x) %(ga_nonEmpty)%

forall x : S . not gn_defined(x) <=> x = gn_bottom_S
                                          %(ga_notDefBottom)%

. exists x : T . gn_defined(x) %(ga_nonEmpty_1)%

forall x : T . not gn_defined(x) <=> x = gn_bottom_T
                                        %(ga_notDefBottom_1)%

forall x_1 : S . gn_defined(gn_inj_S_T(x_1)) <=> gn_defined(x_1)
                                            %(ga_totality)%

forall x_1 : S . gn_defined(f(x_1)) => gn_defined(x_1)
                                          %(ga_strictness)%

forall x_1 : T . gn_defined(gn_proj_T_S(x_1)) => gn_defined(x_1)
                                          %(ga_strictness_1)%

forall x, y : S . gn_defined(x) & gn_defined(y)
  => gn_inj_S_T(x) = gn_inj_S_T(y) & gn_defined(gn_inj_S_T(x))
     => x = y & gn_defined(x)        %(ga_embedding_injectivity)%

forall x, y : T . gn_defined(x) & gn_defined(y)
  => gn_proj_T_S(x) = gn_proj_T_S(y) & gn_defined(gn_proj_T_S(x))
     => x = y & gn_defined(x)       %(ga_projection_injectivity)%

forall x : S . gn_defined(x)   => gn_proj_T_S(gn_inj_S_T(x)) = x
     & gn_defined(gn_proj_T_S(gn_inj_S_T(x)))  %(ga_projection)%
```

Figure 4.5: Specification after encoding of partial functions the in logic $FOL^{=}$.

```
typedecl S
typedecl T

consts
X_f :: "S => T" ("f/'(_')" [3] 999)
X_gn_bottom_S :: "S" ("gn'_bottom'_S")
X_gn_bottom_T :: "T" ("gn'_bottom'_T")
X_gn_inj_S_T :: "S => T" ("gn'_inj'_S'_T/'(_')" [3] 999)
X_gn_proj_T_S :: "T => S" ("gn'_proj'_T'_S/'(_')" [3] 999)
gn_definedX1 :: "S => bool" ("gn'_defined''/'(_')" [3] 999)
gn_definedX2 :: "T => bool" ("gn'_defined''''/'(_')" [3] 999)

instance S:: type ..
instance T:: type ..

ga_nonEmpty [rule_format] : "EX x. gn_defined'(x)"

ga_notDefBottom [rule_format] :
"ALL x. (~ gn_defined'(x)) = (x = gn_bottom_S)"

ga_nonEmpty_1 [rule_format] : "EX x. gn_defined''(x)"

ga_notDefBottom_1 [rule_format] :
"ALL x. (~ gn_defined''(x)) = (x = gn_bottom_T)"

ga_totality [rule_format] :
"ALL x_1. gn_defined''(gn_inj_S_T(x_1)) = gn_defined'(x_1)"

ga_strictness [rule_format] :
"ALL x_1. gn_defined''(f(x_1)) --> gn_defined'(x_1)"

ga_strictness_1 [rule_format] :
"ALL x_1. gn_defined'(gn_proj_T_S(x_1)) --> gn_defined''(x_1)"

ga_embedding_injectivity [rule_format] : "ALL x. ALL y.
 gn_defined'(x) & gn_defined'(y) -->
 gn_inj_S_T(x) = gn_inj_S_T(y) & gn_defined''(gn_inj_S_T(x)) -->
 x = y & gn_defined'(x)"

ga_projection_injectivity [rule_format] : "ALL x.  ALL y.
 gn_defined''(x) & gn_defined''(y) -->
 gn_proj_T_S(x) = gn_proj_T_S(y) & gn_defined'(gn_proj_T_S(x))
  --> x = y & gn_defined''(x)"

ga_projection [rule_format] : "ALL x. gn_defined'(x) -->
 gn_proj_T_S(gn_inj_S_T(x)) = x &
 gn_defined'(gn_proj_T_S(gn_inj_S_T(x)))"
```

Figure 4.6: Specification encoded within Isabelle/HOL.

of CASL in Isabelle/HOL exists). This means that the semantic objects are encoded directly in Isabelle/HOL without any syntax (see [NvOP00] for further details of shallow and deep encodings). The `typedecl` commands that have been produced extend the signature with loosely specified types that Isabelle/HOL requires types to be non-empty. This matches the fact that CASL also requires sorts to be non-empty.

Using this encoding we can now translate any CASL specification into an Isabelle/HOL theory file with the same semantics. We use this feature for part of the translation of CSP-CASL specifications. In particular we encode the data part of a CSP-CASL specification into Isabelle/HOL using the exact method described here.

4.3 CSP-Prover

CSP-Prover [IR05, IR06, IR] is an interactive theorem prover built upon Isabelle/HOL. CSP-Prover is dedicated to refinement proofs within the process algebra CSP [Hoa85, Ros98]. It is generic in the models of CSP that can be used. Currently the trace model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$ are available, while implementations of the models $\mathcal{R}$ and $\mathcal{N}$ are well underway. The stable failures model has been shown to be complete [IR06].

CSP-Prover provides a deep-encoding of CSP within Isabelle/HOL (i.e., the syntax and semantics of CSP processes have been encoded within Isabelle/HOL). Consequently, it offers a CSP process type `'a proc`, where `'a` is an Isabelle/HOL type variable. This type variable can be instantiated with any Isabelle/HOL type. For instance the type `int proc` is the type of CSP processes where communications are integer values (values of type int).

CSP-Prover supports two methods for allowing the user to prove process refinements, namely syntactical and semantical proofs.

Semantical proofs evaluate the denotational semantics of CSP processes and compare the denotations. For example, when working in the traces model, it is common to be proving equality between sets of traces of CSP processes as part of a process refinement proof. Semantical proofs tend to be more challenging than syntactical proofs.

Syntactical proofs transform the syntax of CSP processes into equivalent CSP processes via CSP laws, until syntactical identity is reached. CSP-Prover provides a large collection of such CSP laws and tactics. Such tactics combine CSP laws to provide powerful proof principles. These CSP laws have been proven to be correct with respect to the semantics. Hence one can use them without looking into their semantics, which means that proofs of process refinements can remain purely syntactical.

During this project we have only used syntactic proofs.

4.3.1 The encoding

CSP-Prover provides a deep encoding of CSP in Isabelle/HOL. As part of this encoding there is a recursive data type `'a proc` along with other types representing the semantic domains for each model. For instance `'a domT` is the type representing the semantic domain of the traces model

$\mathcal{T}$, which is a set of traces over the type `'a` which satisfy a healthiness condition. Similarly there is a type `'a domF` which represents the semantic domain of the stable failures model $\mathcal{F}$. As the semantic domain of the stable failures model consists of pairs of traces and failures (which satisfy some healthiness conditions), the definition of the type `'a domF` uses the type of traces `'a domT`. Hence the deep encoding reflects the relationship between the CSP traces model $\mathcal{T}$ and the stable failures model $\mathcal{F}$.

Finally, semantic maps have also been encoded into Isabelle/HOL. These map processes to the semantic domain of each CSP model. For instance, there is a semantic map $[\![\_]\!]_{\mathcal{T}}$: `'a proc` $\Rightarrow$ `'a domT` which maps processes to the semantic domain of the traces model $\mathcal{T}$. Similarly, there is a semantic map for the stable failures model $\mathcal{F}$, $[\![\_]\!]_{\mathcal{F}}$: `'a proc` $\Rightarrow$ `'a domF` which maps processes to the semantic domain of the stable failures model.

4.3.2 CSP process refinements

Each CSP model defines a notion of refinement between two CSP processes. The simplest notion of refinement is within the traces model $\mathcal{T}$, where a process Q is a refinement of a process P if and only if all the possible sequences of communications that Q can perform are also possible for P. This is formally written as

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

where the function $traces(P)$ is defined as the set of all possible sequences which the process P can engage in. The process `STOP` is a traces refinement of every process as it communicates no events. Trace refinements can be written within CSP-Prover as

```
theorem "P <=T Q"
```

in order to create a new theorem that states that `Q` is a traces refinement of `P`.

Refinement within the stable failures model ($\mathcal{F}$) is defined as

$$P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P)$$

where $failures(P)$ is the set of all P's failures. This can be written similarly in CSP-Prover as

```
theorem "P <=F Q"
```

in order to setup a stable failures refinement theorem.

In Csp, deadlock is represented by the process `Stop`. A process is deadlock- free, if it never reaches a state equivalent to `Stop` [IRar]. Stable failures refinement preserves deadlock freedom i.e., if $P \sqsubseteq_{\mathcal{F}} Q$ and P is deadlock free then Q is also deadlock free. We use this fact to prove (see Section 7.2) deadlock freedom of an electronic payment system standard, namely EP2 [ep202].

4.3.3 Syntactic proofs without tactics

We give no examples of semantic proofs in CSP-Prover as we only use syntactical proofs within this project. The CSP-Prover User Guide [IR07] provides detailed examples of how to use both semantic proofs and syntactical proofs.

To perform a *syntactic proof* (without using tactics) one must apply CSP laws which CSP-Prover provides to discharge the proof obligations. The CSP-Prover User Manual calls this *manual syntactical proofs*. All the CSP laws that are provided have been proved correct at the semantic level, which means we can use them without looking in to their semantics.

First we briefly present an example from [IR07] for syntactic proofs, whilst explaining the CSP laws that are used. We wish to prove that the processes `(a -> P) |[{a}]| (a -> Q)` and `a -> (P |[{a}]| Q)` are equivalent within the stable failures model.

The first process `(a -> P) |[{a}]| (a -> Q)` is the process which has two sub processes, namely, a process that communicates an `a` action and then behaves like the process `P` and a processes that communicates an `a` action and then behaves like the process `Q`. Both these sub-processes synchronise over the set $\{a\}$. This means that the process `(a -> P) |[{a}]| (a -> Q)` can only perform the action `a` if both sub-processes are ready to engage in the event `a`.

The second process `a -> (P |[{a}]| Q)` is similar in that it communicates the action `a` and then behaves like the process `(P |[{a}]| Q)`, which is the process formed when the sub-processes `P` and `Q` synchronise over the set $\{a\}$.

In the following code, lines 1 and 2 create the theorem while the rest prove it.

```
1  lemma syntactical_proof:
2    "(a -> P) |[{a}]| (a -> Q) =F a -> (P |[{a}]| Q)"
3  apply (rule cspF_rw_left)
4  apply (rule cspF_decompo)
5  apply (simp)
6  apply (rule cspF_step)
7  apply (rule cspF_step)
8
9  apply (rule cspF_rw_left)
10 apply (rule cspF_step)
11
12 apply (rule cspF_rw_right)
13 apply (rule cspF_step)
14
15 apply (rule cspF_decompo)
16 apply (simp)
17 apply (simp)
18 done
```

The above proof works by carefully controlling which CSP laws are applied to specific sub-expression at the correct time. This involves decomposing the process to the correct level, then applying the CSP step laws and finally applying some simplification.

After lines 1 and 2 are loaded, we have the following goal[3]:

```
goal (lemma (syntactical_proof), 1 subgoal):
 1. (a -> P) |[{a}]| (a -> Q) =F a -> (P |[{a}]| Q)
```

[3] Some of the header information which Isabelle displays with the goals have been omitted.

The main idea for discharging this goal is to:

1. re-formulate the sub-processes in terms of the *Action Prefix* operator using step laws,
2. apply the *Action Prefix* step law, and
3. perform simplification of the remaining goal.

We cannot apply certain useful CSP laws including step laws until we have isolated one of the sides of the equation. We do this by applying the rules `cspF_rw_left` or `cspF_rw_right` to rewrite either the left of right side of the goal, respectively. This will cause the goal to be split into two sub-goals linked with a schematic variable, hence isolating one of the sides of the equation.

After Line 3 is loaded we have the following sub-goals:

```
goal (lemma (syntactical_proof), 2 subgoals):
 1. (a -> P) |[{a}]| (a -> Q) =F ?P2.0
 2. ?P2.0 =F a -> (P |[{a}]| Q)
```

where `?P2.0` is a schematic variable used to link both sides of the equation. We would now like to transform the process `a -> P` into the equivalent process `? x:{a} -> P` (and similarly for the process `a -> Q`).

First, we have to decompose the generalised parallel operator in order to access the left and right operands of the generalised parallel operator. We do this in line 4 by using the `cspF_decompo` law which results in the following sub-goals:

```
goal (lemma (syntactical_proof), 4 subgoals):
 1. {a} = ?Y1
 2. a -> P =F ?Q1.1
 3. a -> Q =F ?Q2.1
 4. ?Q1.1 |[?Y1]| ?Q2.1 =F a -> (P |[{a}]| Q)
```

Here sub-goals 2 and 3 are our left and right operands of the generalised parallel operator, which are now isolated.

Next we use the CSP step laws to transform our two processes `a -> P` and `a -> Q` into the processes `? x:{a} -> P` and `? x:{a} -> Q`, respectively. After lines 5-7 are loaded we have the following goal:

```
goal (lemma (syntactical_proof), 1 subgoal):
1. (? x:{a} -> P) |[{a}]| (? x:{a} -> Q) =F a -> (P |[{a}]| Q)
```

Here the process on the L.H.S is now ready to have the parallel operator reduced. this can only be done when the two sub-processes have a prefix choice operator at their heads. We now have this situation which we did not have at the start of this proof.

Now we can reduce the action prefixes with the outer parallel operator (lines 9-10). First by isolating the left hand side of the equation and then applying the CSP step laws. We now turn our attention to the right hand side of the equation. We need to transform the process `a -> (P |[{a}]| Q)` into the equivalent process `? x:{a} -> (P |[{a}]| Q)`. This is done with lines 12-13.

Finally some simplification within the sub-expressions completes the proof, lines 15-18. Full details can be found in [IR07].

This proof is quite intimately linked with the structure of the CSP processes in the theorem. The user spends most of the time decomposing the processes in the correct way in order to apply the CSP step laws. If the process changes slightly the proof will have to be modified accordingly. We can (usually) improve on this by using *tactics* in syntactical proofs.

4.3.4 Syntactic proofs with tactics

By using tactics we can automate much of the *syntactic proofs*, the CSP-Prover User Manual calls this *semi-automatic syntactical proofs*. Here we prove the same theorem but using tactics which are provided by CSP-Prover. This example is also covered in detail within the CSP-Prover User Guide [IR07].

```
1 lemma tactical_proof:
2   "(a -> P) |[{a}]| (a -> Q) =F a -> (P |[{a}]| Q)"
3 apply (tactic {* cspF_hsf_tac 1 *})
4 apply (auto)
5 done
```

Here lines 1 and 2 state the theorem while line 3 does most of the work, by using a powerful tactic. Line 4 performs some simplification via the `auto` proof method. Finally, Line 5 completes the proof.

The tactic `cspF_hsf_tac` is one of the most useful tactics which CSP-Prover provides. This tactic works by first applying the tactics `cspF_hsf_left_tac` and `cspF_hsf_right_tac` in sequence. The tactic `cspF_hsf_left_tac` is used to sequentialise processes on the left hand side of equations. This is achieved by applying certain CSP laws to the processes much like in the previous example in Section 4.3.3. The net result of this tactic is that it is usually capable of decomposing processes and applying the CSP step laws at relevant places.

We use the tactic `cspF_hsf_tac` in Line 3 which first tackles the left hand side followed by the right hand side. The sub-processes `a -> P` and `a -> Q` are transformed into the equivalent processes `? x:{a} -> P` and `? x:{a} -> Q`, respectively. Then the generalised parallel operator is transformed with the CSP step laws. Then the tactic moves to the right hand side where the process `a -> (P |[{a}]| Q)` is transformed into the process `? x:{a} -> (P |[{a}]| Q)`. Finally some simplification via the `auto` proof method completes the proof (line 4).

Here we have seen that the tactics provided by CSP prover can automate much of the proof script. This results in elimination of a lot of the monotonous proof steps and also completes the proof in a much simpler manner where the high level proof structure is usually visible.

Chapter 5

CSP-CASL

Contents

CSP-CASL [Rog06] is a comprehensive language which combines *processes* written in CSP [Hoa85, Ros98] with the specification of *data types* in CASL [Mos04, BM04]. The general idea is to describe reactive systems in the form of processes based on CSP operators, where the communications of these processes are the values of data types, which are loosely specified in CASL. All standard CSP operators are available, such as multiple prefix, the various parallel operators, operators for non-deterministic choice, communication over channels. Concerning CASL features, the full language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and sub-sorting. Furthermore, the various CASL structuring constructs are included, where the structured `free` construct adds the possibility to specify data types with initial semantics. CSP-CASL specifications can be organised in libraries. This allows to specify a complex system in a modular way.

Syntactically, a CSP-CASL specification with name *N* consists of a data part *Sp*, which is a structured CASL specification, an (optional) channel part *Ch* to declare channels, which are typed according to the data part, and a process part *P* written in CSP, within which CASL terms are used as communications, CASL sorts denote sets of communications, relational renaming is described by a binary CASL predicate, and the CSP conditional construct uses CASL formulae as conditions – see Figure 5.1 for an instance of this scheme:

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

5.1 Modelling systems in CSP-CASL

Here we discuss the modelling of systems in CSP-CASL. We first present four core examples taken from [Rog06], which capture the central challenges of theorem proving for CSP-CASL–

Section 5.1.1. We then present specifications of a remote control unit [Kah07] – Section 5.1.2. Finally, we present the CSP-CASL specification of a dialog within the EP2 system – Section 5.1.3.

5.1.1 The four core examples

We present here four specifications taken from [Rog06], which represent the four core cases of CSP-CASL specifications. All other more complex specifications are just various instances and combinations of these four core cases.

```
ccspec tcs1 =
data sort S, T
     ops c: S;
     ops d: T;
process
     tcs1 = c -> SKIP || d -> SKIP
end
```

Figure 5.1: Core CSP-CASL example 1: No sub-sorting or partial functions.

The first core example is shown in Figure 5.1. The data-part has two sorts `S` and `T` which are not related in any way and two constants `c` and `d` of sorts `S` and `T`, respectively. The process part consists of a single process where two sub-processes synchronise over the parallel CSP operator. The first process communicates the constant `c` then terminates while the second process communicates the constant `d` and then terminates. The two sub-processes can only synchronise if they communicate the same event, i.e., if $c = d$ in all models. Analysis of the CSP-CASL semantics leads us to the fact that this is not the case and `c` is not equal to `d` in all models. Hence this process deadlocks and is equivalent to the process `STOP` within the stable failures model $\mathcal{F}$. A prover for CSP-CASL should be able to prove this process equivalence.

```
ccspec tcs2 =
data sorts S < T
     ops c: S; d: T;
     .   c = d;
process
     tcs2 = c -> SKIP || d -> SKIP
end
```

Figure 5.2: Core CSP-CASL example 2: Sub-sorting without partial functions.

The second core example (Figure 5.2) is a modification of the first (Figure 5.1), where the data-part has an axiom added. The axiom states that the constants `c` and `d` are equal. This has the result that the sub-processes do synchronise, as $c = d$ in all models. Hence the process is equal to the process `c -> SKIP` within the stable failures model $\mathcal{F}$. A prover for CSP-CASL should also be able to prove this process equivalence.

```
ccspec tcs3 =
data sorts S, T
     ops f: S ->? T
     . forall x: S . not def f(x);
process
     tcs3 = ? x: S -> f(x) -> SKIP [| T |]
            ? y: T -> (if def y then P else Q)
end
```

Figure 5.3: Core CSP-CASL example 3: Partial functions without sub-sorting.

The third core example (Figure 5.3) starts fresh and introduced undefined terms. The data-part specifies that there are two sorts `S` and `T` which are not related. There is also a partial function from sort `S` to sort `T` and a single axiom that states that the result of the function `f` is undefined for all inputs of sort `S`. The process part again consists of a single process which is the synchronisation of two sub processes over sort `T`. The first sub-process receives an element `x` of sort `S` then communicates the event `f(x)` and finally terminates. The second sub-process receives a value `y` of sort `T` and if this value is defined behaves like the process `P` else it behaves like the process `Q`.

As these two sub-processes synchronise over the sort `T`, the undefined result of `f(x)` is received and bound to the variable `y`. This raises the question whether `y` is defined in all models. The result of this question dictates the remaining behaviour of the second sub-process. The variable `y` is undefined in all models according to the CSP-CASL semantics. Hence the entire process is equivalent to the process `? x:S -> f(x) -> (SKIP [| T |] Q)` within the stable failures model $\mathcal{F}$. Once again, a prover for CSP-CASL should be able to prove this process equivalence.

```
ccspec tcs4 =
data sorts A, B, C < S
     ops a: A;
         b1, b2: B;
         c: C;

         f: A ->? A;
         g: C ->? C

     .   a = b1 . b2 = c
     .   forall x: A . not def f(x)
     .   forall x: C . not def g(x);
process
     tcs4 = f(a) -> SKIP || g(c) -> SKIP
end
```

Figure 5.4: Core CSP-CASL example 4: Sub-sorting with partial functions.

The final core example (Figure 5.4) is a more complex CSP-CASL specification than the previous

three examples, containing both sub-sorting and partial functions. There are four sorts A, B, C and S with sorts A, B and C all being sub-sorts of sort S. There are several constants, namely: a of sort A, b1 and b2 of sort B and c of sort C. There are two partial functions, namely, f from sort A to sort A and g from sort C to sort C. The data-part concludes with two axioms that state that the results of the functions f and g are undefined for all inputs. The process-part consists of a single processes composed of two sub-processes synchronising over the parallel CSP operator. The first sub-process communicates the value f(a) and terminates while the second sub-process communicates the value g(c) and terminates.

As sorts A and C have a common super-sort S and f(a) and g(c) both result in undefined elements in all models, the sub-processes successfully synchronise. Hence the entire process is equivalent to the process g(c) -> SKIP within the stable failures model $\mathcal{F}$. Once again, a prover for CSP-CASL should be able to prove this process equivalence.

If these four core examples can be dealt with in a prover for CSP-CASL, then the prover should be able to handle all possible CSP-CASL refinement proofs. In Chapter 7 we show that our approach for CSP-CASL-Prover can indeed deal with these four core examples.

5.1.2 Case study - Remote control unit

The following presents a specification of a remote control unit. These results are from personal communication with Temesghen Kahsai, Markus Roggenbach, and Holger Schlingloff [Kah07].

We specify a remote control unit (*RCU*) which, for example, could interface and control a television via the user pressing various buttons on the remote control. When a button is pressed the remote control unit sends (usually using infrared light) a signal to another device, for instance a television.

On an abstract level, a remote control unit can be described as having a number of buttons and a light emitting diode (*LED*), which is capable of transmitting keycodes (for instance bitvectors of length 16). The buttons can only be pressed one at a time. Internally, the remote control unit stores a table which maps buttons to keycodes. Whenever a button is pressed, the remote control unit sends the corresponding keycode via the LED. This is captured by the CSP-CASL specification in Figure 5.5.

```
ccspec AbstractRCU=
data sorts Button, Signal
     ops codeOf: Button -> Signal;
process
     AbstractRCU = ?x:Button -> codeOf(x) -> Skip
end
```

Figure 5.5: CSP-CASL specification for an abstract remote control unit.

In the 1970's a typical basic remote control unit (*BRCU*) had at least 11 buttons ($b_0 \ldots b_9, b_{OnOff}$). An agreed standard for remote control units defines the signals that are emitted by the LED. Signals are to be bitvectors of length 16 with the following structure: the first 4 bits identify the company,

the next 5 bits represent the device type (e.g., TV, DVD, etc.), while the last 7 bits identify which button was pressed. This can be captured by the CSP-CASL specification in Figure 5.6.

```
ccspec BRCU=
data sorts Button, Signal
  ops b0, b1, ... , b9, bOnOff: Button;
    free type Bit ::= 0 | 1
      then List[sort Bit]
        then
          sort Signal = {l : List[Bit] . #l = 16 }
          op codeOf: Button -> Signal;
             prefix: List[Bit] = [0000]++[01010]
           axioms
             codeOf(b0) = prefix ++ [0000000];
             ...
             codeOf(b9) = prefix ++ [0001001];
             codeOf(bOnOff) =  prefix ++ [1111111];
             forall b: Button . exists l: List[Bit] .
                                  codeOf(b) = prefix++l
process
  BRCU = ? x: Button -> codeOf(x) -> Skip
end
```

Figure 5.6: CSP-CASL specification for a basic remote control unit.

The specification of the basic remote control unit is a refinement according to to the CSP-CASL refinement notions of the specification of the abstract remote control unit [Kah07], i.e.,

$$\texttt{AbstractRCU} \overset{\text{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} \texttt{BRCU}.$$

A prover for CSP-CASL should also be able to prove this refinement between the specification of the abstract remote control unit and the specification of the basic remote control unit.

Soon after this first generation of basic remote control units, the market demanded improved remote control units with more functionality and, thus, more buttons. In particular, buttons b_{volup} and b_{voldn} for controlling the volume and b_{chup} and b_{chdn} for cycling though channels were introduced.

5.1.3 Case study - EP2

As a running example and case study, we choose a dialog of the EP2 system [ep202]. This dialog has been modelled as a specification in CSP-CASL– see [GRS05] for further details of the modelling approach. The data-part of the CSP-CASL specification for EP2 can been seen in Figure 5.7, while the process-part can be seen in Figure 5.8.

In this dialog, the credit card terminal and another component, the so-called acquirer, are supposed to exchange initialisation information over the channel `C_SI_Init`. The messages on this channel can be classified into the following types:

- `SessionStart` and `SessionEnd`,
- `ConfigDataRequest` and `ConfigDataResponse`,
- `D_SI_Init_ConfigDataNotification` and `D_SI_Init_ConfigDataAcknowledge`,
- `D_SI_Init_RemoveConfigDataNotification` and `D_SI_Init_RemoveConfigDataAcknowledge`,
- `D_SI_Init_ActivateConfigDataNotification` and `D_SI_Init_ActivateConfigDataAcknowledge`.

Each of these message types are declared as sorts (Line 4 to Line 13 of Figure 5.7) in the specification.

In order to model the dialog properly, we need to ensure that certain message groups do not overlap, i.e., messages of type `SessionEnd`, `ConfigDataRequest`, `D_SI_Init_ConfigDataNotification`, `D_SI_Init_RemoveConfigDataNotification`, and `D_SI_Init_ActivateConfigDataAcknowledge` are never equal to each other (Lines 15 to 37 of Figure 5.7).

Finally, we create constants of certain sorts (Line 39 to 43 of Figure 5.7) which are used in the process part as communications (see Figure 5.7).

The terminal initiates the dialog by sending a message of type `SessionStart`, see the process `Ter_Init` in Figure 5.8. The acquirer receives this message, see the process `Acq_Init`.
In `Acq_ConfigManagement`, the acquirer then takes the internal decision either to end the dialog by sending the message `seM` of type `SessionEnd` or to start one of four data exchanges with the terminal. The terminal, on the other side, waits in the process `Ter_ConfigManagement` for a message from the acquirer. Depending on the type of this message, the terminal ends the dialog with `SKIP`, engages in a data exchange, or executes the deadlock process `STOP`. The system consists of the parallel composition of terminal and acquirer. Should one of these two components be in a deadlock, the whole system will be in deadlock.

5.2 CSP-CASL semantics and refinement

Semantically, a CSP-CASL specification is a family of process denotations for a CSP process, where each model of the data part *Sp* gives rise to one process denotation. The definition of the language CSP-CASL is generic in the choice of specific CSP semantics. For example, all denotational CSP models mentioned in [Ros98] are possible parameters, these include the *traces model* $\mathcal{T}$, the *stable failures model* $\mathcal{F}$, the *failures / divergences model* $\mathcal{N}$ and, the newly defined *stable revivals model* $\mathcal{R}$ [Ros05].

The semantics of CSP-CASL are defined in a two-step approach[1], see Figure 5.9. Given a CSP-CASL specification (Sp, P), in the first step we construct for each data model M of Sp a CSP process

[1] We omit the syntactic encoding of channels into the data part.

```
1  library EP2
2
3  spec D_ACL_GetInitinitialisation =
4      sorts D_SI_Init_SessionStart,
5            D_SI_Init_SessionEnd,
6            D_SI_Init_ConfigDataRequest,
7            D_SI_Init_ConfigDataResponse,
8            D_SI_Init_ConfigDataNotification,
9            D_SI_Init_ConfigDataAcknowledge,
10           D_SI_Init_RemoveConfigDataNotification,
11           D_SI_Init_RemoveConfigDataAcknowledge,
12           D_SI_Init_ActivateConfigDataNotification,
13           D_SI_Init_ActivateConfigDataAcknowledge < D_SI_Init
14
15     forall x:D_SI_Init_SessionEnd;
16       y:D_SI_Init_ConfigDataRequest . not (x=y)
17     forall x:D_SI_Init_SessionEnd;
18       y:D_SI_Init_ConfigDataNotification . not (x=y)
19     forall x:D_SI_Init_SessionEnd;
20       y:D_SI_Init_RemoveConfigDataNotification . not (x=y)
21     forall x:D_SI_Init_SessionEnd;
22       y:D_SI_Init_ActivateConfigDataNotification . not (x=y)
23
24     forall x:D_SI_Init_ConfigDataRequest;
25       y:D_SI_Init_ConfigDataNotification . not (x=y)
26     forall x:D_SI_Init_ConfigDataRequest;
27       y:D_SI_Init_RemoveConfigDataNotification . not (x=y)
28     forall x:D_SI_Init_ConfigDataRequest;
29       y:D_SI_Init_ActivateConfigDataNotification . not (x=y)
30
31     forall x:D_SI_Init_ConfigDataNotification;
32       y:D_SI_Init_RemoveConfigDataNotification . not (x=y)
33     forall x:D_SI_Init_ConfigDataNotification;
34       y:D_SI_Init_ActivateConfigDataNotification . not (x=y)
35
36     forall x:D_SI_Init_RemoveConfigDataNotification;
37       y:D_SI_Init_ActivateConfigDataNotification . not (x=y)
38
39     ops seM: D_SI_Init_SessionEnd;
40         cdrM: D_SI_Init_ConfigDataRequest;
41         cdnM: D_SI_Init_ConfigDataNotification;
42         rcdnM: D_SI_Init_RemoveConfigDataNotification;
43         acdnM: D_SI_Init_ActivateConfigDataNotification
44 end
```

Figure 5.7: CASL specification of the data-part of an EP2 dialog between the terminal and the acquirer.

```
ccspec GetInitialisationData =
  data D_ACL_GetInitialisation

  channels
    C_SI_Init: D_SI_Init

  process

  let
    Ter_Init =
       C_SI_Init !? sessionStart:D_SI_Init_SessionStart
    -> Ter_ConfigurationManagement

    Ter_ConfigurationManagement = C_SI_Init ? configMess:D_SI_Init ->
      if  (configMess:D_SI_SessionEnd) then
        Skip
      else if (configMess:D_SI_Init_ConfigDataRequest) then
        C_SI_Init !? response: D_SI_Init_ConfigDataResponse
        -> Ter_ConfigurationManagement
      else if (configMess : D_SI_Init_ConfigDataNotification) then
        C_SI_Init !? acknowledge: D_SI_Init_ConfigDataAcknowledge
        -> Ter_ConfigurationManagement
      else if (configMess : D_SI_Init_RemoveConfigDataNotification) then
        C_SI_Init !? acknowledge: D_SI_Init_RemoveConfigDataAcknowledge
        -> Ter_ConfigurationManagement
      else (configMess : D_SI_Init_ActivateConfigDataNotification) then
        C_SI_Init !? acknowledge:
        D_SI_Init_ActivateConfigDataAcknowledge
        -> Ter_ConfigurationManagement

    Acq_Init =
       C_SI_Init ? sessionStart: D_SI_Init_SessionStart
    -> Acq_ConfigurationManagement

    Acq_ConfigurationManagement =
         C_SI_Init ! seM -> Skip
    |~|  C_SI_Init ! cdrM
         -> C_SI_Init ? response: D_SI_Init_ConfigDataResponse
         -> Acq_ConfigurationManagement
    |~|  C_SI_Init ! cdnM
         -> C_SI_Init ? acknowledge: D_SI_Init_ConfigDataAcknowledge
         -> Acq_ConfigurationManagement
    |~|  C_SI_Init ! rcdnM
         -> C_SI_Init ? acknowledge:
         D_SI_Init_RemoveConfigDataAcknowledge
         -> Acq_ConfigurationManagement
    |~|  C_SI_Init ! acdnM
         -> C_SI_Init ? acknowledge:
         D_SI_Init_ActivateConfigDataAcknowledge
         -> Acq_ConfigurationManagement
  in
    Acq_Init |[ C_SI_Init ]| Ter_Init
end
```

Figure 5.8: A CSP-CASL specification of an EP2 dialog between the terminal and the acquirer.

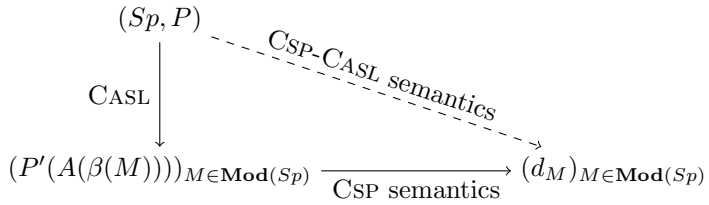

Figure 5.9: CSP-CASL semantics.

$P'(A(\beta(M)))$. To this end, we define for each model M, which might include partial functions, an equivalent model $\beta(M)$ in which partial functions are totalised. $\beta(M)$ gives rise to an alphabet of communications $A(\beta(M))$. In order to deal with CSP binding, we introduce variable evaluations $\nu : X \to \beta(M)$. With these notations we define the process $P'(A(\beta(M))) := [\![P]\!]_{\emptyset:\emptyset\to\beta(M)}$, where $\emptyset$ is the empty evaluation from the empty set to the model $\beta(M)$, i.e., P has no free variables. In the second step we point-wise apply a denotational CSP semantics. This translates a process $P'(A(\beta(M)))$ into its denotation d_M in the semantic domain of the chosen CSP model.

In the following we sketch the alphabet construction – see [Rog06] for the full details. The purpose of the alphabet construction is to transform a CASL model into a set for use as an alphabet of communications in the process algebra CSP. CASL models are defined in two steps: First, we define what a model over a many-sorted signature is. Using this concept we define what a model over a sub-sorted signature is. For the sake of readability we repeat certain notions from Section 3.3.1.

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

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

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

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

Together with the standard definition of first order logic formulae and their satisfaction, this definition yields the institution $PFOL^=$, see [Mos02] for the details.

A *sub-sorted signature* $\Sigma = (S, TF, PF, P, \leq)$ 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 point-wise to sequences of sorts. With each sub-sorted signature $\Sigma = (S, TF, PF, P, \leq)$ we associate a many-sorted signature $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$, which extends the underlying many-sorted signature (S, TF, PF, P) with

- a total *injection* function symbol $\mathtt{inj} : s \to s'$ for each pair of sorts $s \leq_S s'$,

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

Sub-sorted Σ*-models* are many-sorted $\hat{\Sigma}$-models satisfying in *PFOL*$^=$ the set of axioms $\hat{J}(\Sigma)$, which prescribe how the injection, projection, and membership behave[2]. A typical axiom in $\hat{J}(\Sigma)$ is $\mathtt{inj}_{s,s}(x) \stackrel{e}{=} x$ for $s \in S$. Together with the definition of sub-sorted first order logic formulae and their satisfaction, this definition yields the institution *SubPFOL*$^=$, see [Mos02] for the details.

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

A is a data-logic signature $\Sigma = (S, TF, PF, P, \leq)$ with local top elements, if for all $u, u', s \in S$ the following holds: if $u, u' \geq s$ then there exists $t \in S$ with $t \geq u, u'$. Relatively to a model M for a signature with top elements, we define an alphabet of communications

$$A(M) := (\biguplus_{s \in S} M_s)_{/\sim}$$

where $(s, x) \sim (s', x')$ iff either

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

or

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

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

for $s, s' \in S, x \in M_s, x' \in M_{s'}$. For signatures with local top elements the relation $\sim$ turns out to be an equivalence relation.

The following presents a CSP-CASL refinement [Rog06] decomposition theorem which allows one to split a CSP-CASL refinement into a first refinement of the data part followed by a refinement of the process part. These results are from Temesghen Kahsai [Kah07].

For a denotational CSP model with domain $\mathcal{D}$, the semantic domain of CSP-CASL consists of the class of I-indexed families of process denotations $d_M \in \mathcal{D}$, i.e.,

$$(d_M)_{M \in I}$$

where I is the class of *SubPFOL*$^=$ models.

[2] and also define how overloading works.

We define CSP-CASL refinement between two semantic objects, denoted as $\rightsquigarrow_{\mathcal{D}}$, by:

$$\begin{gathered}(d_M)_{M\in I} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M'\in I'} \\ \text{iff} \\ I' \subseteq I \wedge \forall M' \in I' : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'},\end{gathered}$$

Here $I' \subseteq I$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathcal{D}}$ is the refinement notion in the chosen CSP model $\mathcal{D}$. In the traces model $\mathcal{T}$ we have for instance $T \sqsubseteq_{\mathcal{T}} T' :\Leftrightarrow T' \subseteq T$, where T and T' are prefix closed sets of traces[3]. The definitions of CSP refinements for $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}, \mathcal{I}, \mathcal{U}\}$, c.f. [Ros98], which are all based on set inclusion, yield that CSP-CASL refinement is a preorder.

We can now characterise *data refinement* between two semantic objects denoted as $\overset{data}{\rightsquigarrow}$, by:

$$\begin{gathered}(d_M)_{M\in I} \overset{data}{\rightsquigarrow} (d'_{M'})_{M'\in I'} \\ \text{if} \\ I' \subseteq I \ \wedge \forall M' \in I' \ : d_{M'} = d'_{M'}\end{gathered}$$

Here $I' \subseteq I$ denotes inclusion of model classes over the same signature.

We can also characterise *process refinement* in a similar way at the semantic level, denoted as $\overset{process}{\rightsquigarrow_{\mathcal{D}}}$, by:

$$\begin{gathered}(d_{M'})_{M'\in I'} \overset{process}{\rightsquigarrow_{\mathcal{D}}} (d'_{M'})_{M'\in I'} \\ \text{if} \\ \forall M' \in I' : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'}\end{gathered}$$

We denote a CSP-CASL specification as a pair (Sp, P), where Sp and P represent the specification of the data part and the process part, respectively. We can now lift the notion of CSP-CASL refinement at the semantic level ($\rightsquigarrow_{\mathcal{D}}$) to CSP-CASL refinement on specifications, denoted by $\overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}}$. Note that in the data refinement is not necessary for the domain $_{\mathcal{D}}$ to be specified as the process part always remains constant. CSP-CASL refinement on two specifications is defines as:

$$\begin{gathered}(Sp, P) \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} (Sp', P') \\ \text{iff} \\ \mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp) \wedge \\ \forall M' \in \mathbf{Mod}(Sp') \ : [\![P]\!]_{\emptyset:\emptyset\to\beta(M')}]\!]_{\text{CSP}} \sqsubseteq_{\mathcal{D}} [\![[\![P']\!]_{\emptyset:\emptyset\to\beta(M')}]\!]_{\text{CSP}}\end{gathered}$$

where $mod(Sp)$ is the class of models for the specification Sp.

We can now use this to characterise *data refinement* and *process refinement* at the specification level. We obtain the following *data refinement* at the specification level for a *fixed* signature and a *fixed* process-part, which we denote as $\overset{\texttt{data}}{\rightsquigarrow}$:

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

[3] We follow here the CSP convention, where T' *refines* T is written as $T \sqsubseteq_{\mathcal{D}} T'$, i.e., the more specific process is on the right-hand side of the symbol.

We also obtain the following *process refinement* at the specification level for a *fixed* data-specification Sp, which we denote as $\overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}}$:

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

Here, $[\![\_]\!]_{\text{CSP}}$ is the evaluation of processes according to the CSP denotational semantics, and $\emptyset:\emptyset\to\beta(M)$ is the empty evaluation into the *CommSubPFOL*$^{=}$ model $\beta(M)$.

Theorem 5.1: CSP-CASL refinement between two specifications can be decomposed into first a *data refinement* followed by a *process refinement* i.e.,

$$\begin{array}{c}(Sp,P) \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P') \\ \Longleftrightarrow \\ (Sp,P) \overset{\texttt{data}}{\rightsquigarrow} (Sp',P) \;\; \textit{and} \;\; (Sp',P) \overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P')\end{array}$$

Proof. We prove this theorem by showing both directions of the equivalence individually:

"$\Longrightarrow$" Let $(Sp,P) \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P')$, by definition this holds iff :

1. $\mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp)$
2. $\forall M' \in \mathbf{Mod}(Sp') \;:\; [\![[\![P]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}} \sqsubseteq_{\mathcal{D}} [\![[\![P']\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}}$

Data refinement $(Sp,P) \overset{\texttt{data}}{\rightsquigarrow} (Sp',P)$ holds if $\mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp)$ and $\Sigma(Sp) = \Sigma(Sp')$, this holds thanks to (1). Process refinement $(Sp',P) \overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P')$ holds if $\forall M \in \mathbf{Mod}(Sp)$: $[\![[\![P]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}} \sqsubseteq_{\mathcal{D}} [\![[\![P']\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}}$. This is proven by (2) condition of CSP-CASL refinement.

"$\Longleftarrow$" Let $(Sp,P) \overset{\texttt{data}}{\rightsquigarrow} (Sp',P)$ if $\mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp) \;\wedge\; \Sigma(Sp) = \Sigma(Sp')$, and $(Sp',P) \overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P')$. This holds if $\forall M' \in \mathbf{Mod}(Sp') \;:\; [\![[\![P]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}} \sqsubseteq_{\mathcal{D}} [\![[\![P']\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}}$. By Taking both of the conditions we prove $(Sp,P) \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P')$.

The last direction can also be proven using the preorder property of CSP-CASL refinement. □

However is not possible to decompose a CSP-CASL refinement between specifications in to first a *process refinement* followed by a *data refinement*, i.e.,

$$\begin{array}{c}(Sp,P) \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P') \\ \not\Longrightarrow \\ (Sp,P) \overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp,P') \;\; \textit{and} \;\; (Sp,P') \overset{\texttt{data}}{\rightsquigarrow} (Sp',P')\end{array}$$

To prove this, we give a counter-example that shows that performing a process refinement followed by a data refinement leads to an inconsistent specification on the data-part.

Example 5.2: As a counter-example, we consider the following CSP-CASL specifications:

```
ccspec First =
data sorts  S, T
     ops a:S;
         b:T;
         f: S -> T
process
     a -> Stop
```

```
ccspec Second =
data sorts S, T
     ops a:S;
         b:T;
         f: S -> T
     axiom
         not(f(a) = b)
process
  if f(a)=b then b -> Stop
            else a -> Stop
```

The CSP-CASL specification `First` is a refinement of the CSP-CASL specification `Second`, i.e.,

$$\texttt{First} \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} \texttt{Second}$$

This holds if and only if $\mathbf{Mod}(Sp_{Second}) \subseteq \mathbf{Mod}(Sp_{First})$ and $\forall M' \in \mathbf{Mod}(Sp_{Second})$: $[\![[\![P_{First}]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}} \sqsubseteq_{\mathcal{D}} [\![[\![P_{Second}]\!]_{\emptyset:\emptyset\to\beta(M)}]\!]_{\text{CSP}}$. Let M be a model of Sp_{First} such that $M(f)(M(a)) = M(b)$. Let us consider now the process refinement and analyse the set of traces of $[\![P_{Second}]\!]$ and $[\![P_{First}]\!]$.

Evaluation	Traces
$[\![P_{Second}]\!]$	$\{\langle\rangle, M(b)\}$
$[\![P_{First}]\!]$	$\{\langle\rangle, M(a)\}$

$\{\langle\rangle, M(a)\} \not\subseteq \{\langle\rangle, M(b)\}$ as b and a are from different sorts. Therefore $[\![P_{First}]\!] \not\sqsubseteq_{\mathcal{T}} [\![P_{Second}]\!]$. This shows that we can't perform first a refinement on the process part and then on the data part.

The other direction of the implication

$$\begin{gathered}(Sp,P) \overset{\texttt{proc}}{\rightsquigarrow_{\mathcal{D}}} (Sp,P') \;\; \textit{and} \;\; (Sp,P') \overset{\texttt{data}}{\rightsquigarrow} (Sp',P') \\ \Longrightarrow \\ (Sp,P) \overset{\texttt{cc-ref}}{\rightsquigarrow_{\mathcal{D}}} (Sp',P')\end{gathered}$$

holds thanks to the preorder property of CSP-CASL refinement.

Chapter 6

Encoding the CSP-CASL semantics in Isabelle/HOL

Contents

In this chapter we first discuss the architecture of CSP-CASL-Prover and how it uses the existing tools HETS (see Section 4.2) and CSP-Prover (see Section 4.3). We then describe an algorithm which translates the data-part of a CSP-CASL specification into a semantically equivalent Isabelle/HOL theory, adds the semantic construction of CSP-CASL, and theorems with proofs that justify this construction. This theory file is ready for use within CSP-Prover.

6.1 Architecture of CSP-CASL-Prover

CSP-CASL-Prover uses the existing tools HETS and CSP-Prover discussed in Sections 4.2 and 4.3, respectively. Its proposed architecture is shown in Figure 6.1. The overall idea is that CSP-CASL-Prover takes a CSP-CASL process refinement statement as its input. The CSP-CASL specifications involved are parsed and transformed by CSP-CASL-Prover into a new file suitable for use in CSP-Prover. This file can then be directly used within CSP-Prover to interactively prove if the CSP-CASL process refinement holds. For example, deadlock freedom of a system of processes can be proven using such a refinement statement, see Section 7.2 for details.

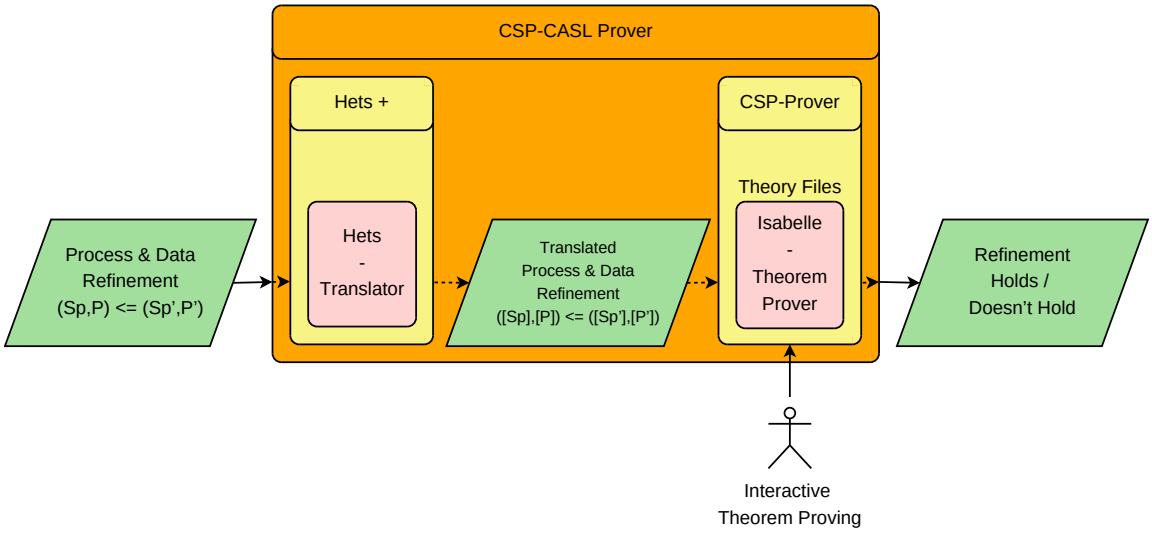

Figure 6.1: Diagram of the basic architecture of CSP-CASL-Prover.

CSP-CASL-Prover re-uses the existing functionality of HETS in order to produce part of the file that will be used as input to CSP-Prover. We take the data part of a CSP-CASL specification and translate this into Isabelle/HOL code via HETS. This generates (in general) several types in Isabelle/HOL, which need to be transformed into one alphabet to become the parameter of the CSP-Prover process type `'a proc`. This is expressed in Figure 6.1 by HETS being labelled as "Hets +", which represents the extra encoding that needs to be done. This is discussed in more detail in the following sections.

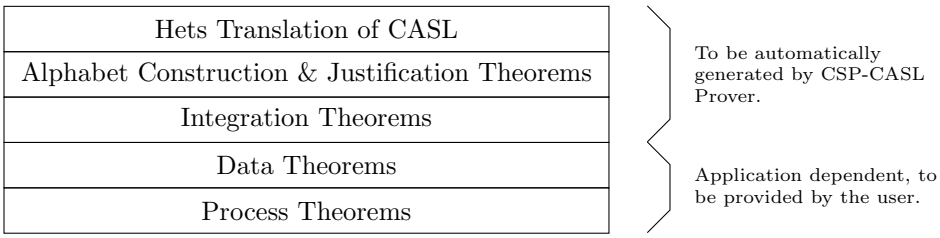

Figure 6.2: Structure of a translated CSP-CASL specification using CSP-CASL-Prover.

The final form of the file which is produced by CSP-CASL-Prover (i.e., HETS and the extra encoding) is labelled as "Translated Processes and Data Refinement" in Figure 6.1. Figure 6.2 shows how this file is split up into five distinct parts. The first three parts can all be automatically generated from the original CSP-CASL specification. The final two parts are dependent on the application. CSP-CASL-Prover provides place holder code that the user can fill in and expand for these two parts.

The first part of the file shown in Figure 6.2 "Hets Translation of CASL" is the direct encoding of the data part of the CSP-CASL specification which is produced by HETS– see Section 6.4. The second part "Alphabet Construction & Justification Theorems" provides the CSP-CASL semantics, namely the alphabet of communications, over which CSP processes can be constructed – see Section 6.5.

```
spec running_example =

sorts R, S, T, U
sorts S,U < T

op f: S -> S

end
```

Figure 6.3: CASL specification – Running example.

```
1 Main:
2   ProduceHeader
3   ProduceHetsEncoding
4   ProduceAlphabetAndJustificationTheorems
5   ProduceIntegrationTheorems
6   ProduceDataTheoremsPlaceHolder
7   ProduceProcessTranslations
```

Figure 6.4: CSP-CASL main algorithm.

The third part "Integration Theorems" provides the user with a mechanism to lift proof obligations on processes to proof obligations on data in the HETS encoding only – see Section 6.6. These integration theorems are crucial in keeping the final proof of the process refinement small, readable and manageable – see Section 7.2 for an example. The forth part is where the user shall write auxiliary theorems and proofs which are helpful for the specific process refinement to be proven – see Section 6.7. The final part is where the user shall provide the proof of the refinement between the processes – see Section 6.8. This is the users main goal.

6.2 The algorithm

The algorithm takes a CSP-CASL specification and produces a theory file which is suitable for use within Isabelle/HOL and CSP-Prover such that there is a single type `Alphabet` which can be used within CSP-Prover in order to create processes which use data specified by CASL as communications i.e., the type `Alphabet proc`. Such a theory file consists of five main sections (see Figure 6.2) and a small header.

Most of the steps in the algorithm are calls to sub-algorithms (we use sub-algorithms in order to split up the main algorithm into smaller more manageable parts), perform simple loops or output strings. Such strings are Isabelle/HOL commands which will be interpreted by Isabelle/HOL when the theory file is loaded.

We describe the algorithm using a procedural pseudocode style. We use the CASL specification in Figure 6.3 as a running example of how our algorithm works on the data-part of CSP-CASL

specifications. The specification in Figure 6.3 has four sorts R, S, T and U. Sorts S and U are sub-sorts of T, while sort R is completely unrelated. We also have a single total function f from S to S. The sub-sort structure will become important in the example when we construct the eq function (see Section 6.5.2).

The starting point of our algorithm can be seen in Figure 6.4, which simply calls many sub-algorithms. Each of the sub-algorithms corresponds to a section of the theory file that is produced (see Figure 6.2). First we produce a small header that makes the file a valid theory file in Isabelle/HOL with support for CSP-Prover. The HETS encoding of the data-part is then produced followed by the construction of the alphabet and justification theorems which rely on the constructions of the HETS encoding. Integration theorems are then produced which provide support for proving properties on the alphabet. Next a place holder for user specific theorems is produced. Finally, the translated processes are produced which use terms built from the alphabet as communications.

We will now describe each of the sub-algorithms within the remainder of this chapter.

6.3 Producing the header

The sub-algorithm ProduceHeader is very simple and is used to produce the header of the theory file. It outputs a series of strings which when interpreted perform actions in Isabelle/HOL. The following Isabelle/HOL code is produced by this sub-algorithm:

```
theory name
imports CSP_Model
begin
```

Line 1 sets up the name of the theory file, this name should be the name of the CSP-CASL specification (although the name can be anything without affecting the operation of the algorithm or resulting theory file). Line 2 outputs the statement that the theory file is based on the CSP model CSP_Model (a theory file provided by CSP-Prover). For this thesis we only use the stable failures model $\mathcal{F}$ and hence we use the text CSP_F for the parameter CSP_Model, an alternatively would be CSP_T if one wishes to use the traces model $\mathcal{T}$ instead. This allows us later to access the type proc provided by CSP-Prover. Line 3 simply begins the main contents of the theory file.

We have now produced the header for a valid Isabelle/HOL theory file ready for the rest of the algorithm to produce the contents of it.

6.4 Producing the HETS encoding

The purpose of the sub-algorithm ProduceHetsEncoding is to represent the data-part of the CSP-CASL specification in Isabelle/HOL so that we can build upon it later in the algorithm. The tool HETS (see Section 4.2) can already transform a CASL specification into Isabelle/HOL code. As our CSP-CASL specification consists of a data-part (CASL) and a process-part (CSP) we use the tool HETS to transform the CASL part into Isabelle/HOL code. This translation is done via the encoding

"CASL2PCFOL" → "PCFOL2CFOL" → "CFOL2IsabelleHOL". This particular transformation encodes the sub-sort relation as a pair of total injection and projection functions between each pair of sorts in the sub-sort relation of the specification. It encodes partial functions (including the projection functions) as total functions by providing a unique undefined element called the bottom element for each sort. See Section 4.2.1 for details of the encoding.

The direct output of HETS (version 0.60) cannot be used as Isabelle/HOL code. This is because there is no theory header, which has been taken care of by the sub-algorithm `ProduceHeader` (see Section 6.3) and also because the Isabelle/HOL command

```
axioms
```

which states that axioms are to follow, needs to be added between the function declarations and the axioms produced by HETS.

Figure 6.5 shows the HETS encoding of the specification of our running example (Figure 6.3). Line 30 has been added manually as described above. The dots "..." stand for omitted code, which does not have relevance for our further discussions.

At this point we now have a theory file which models the data-part of a CSP-CASL specification. However there is no single data-type that can be used which represents all sorts in the specification. This functionality is produced by the sub-algorithm ProduceAlphabetAndJustificationTheorems (see Section 6.5).

6.5 Producing the alphabet and justification theorems

The sub-algorithm `ProduceAlphabetAndJustificationTheorems` (see Figure 6.6) is the core of the CSP-CASL encoding and calls a number of other sub-algorithms (again we use sub-algorithms to maintain readability). First a new type `PreAlphabet` is produced (Line 2) followed by a function `eq` which tests whether two elements of the `PreAlphabet` are equal (Line 3). As we have sub-sorting available in the CASL specification (data-part of the CSP-CASL specification), many values of type `PreAlphabet` will represent the same value. Hence we need to work with equivalence classes of `PreAlphabet` where all the elements within a class represent the same value. We can build the quotient of `PreAlphabet` over the function `eq` if the function `eq` forms an equivalence relation. Line 4 produces the theorems and proofs that are needed in order to establish the function `eq` as an equivalence relation. Line 5 then uses these theorems to create the `Alphabet` of communications which is the quotient of the `PreAlphabet`. Finally lines 6 and 7 produce new types and functions over the type `Alphabet` which are used when defining CSP processes.

These sub-algorithms are described in the following sub-sections.

6.5.1 Producing the `PreAlphabet`

We need a way of collecting all the sorts of the CASL specification into a single type within Isabelle/HOL. This is what the sub-algorithm `ProducePreAlphabet` does, which we now de-

```
typedecl R
typedecl S
typedecl T
typedecl U

consts
f :: "S => S" ("f'(_')" [10] 999)
g__bottom_1 :: "R" ("g'_'_bottom")
g__bottom_2 :: "S" ("g'_'_bottom''")
g__bottom_3 :: "T" ("g'_'_bottom''''")
g__bottom_4 :: "U" ("g'_'_bottom'_3")
g__defined_1 :: "R => bool" ("g'_'_defined'(_')" [10] 999)
g__defined_2 :: "S => bool" ("g'_'_defined'''(_')" [10] 999)
g__defined_3 :: "T => bool" ("g'_'_defined'''''(_')" [10] 999)
g__defined_4 :: "U => bool" ("g'_'_defined'_3'(_')" [10] 999)
g__inj_1 :: "S => T" ("g'_'_inj'(_')" [10] 999)
g__inj_2 :: "U => T" ("g'_'_inj'''(_')" [10] 999)
g__proj_1 :: "T => S" ("g'_'_proj'(_')" [10] 999)
g__proj_2 :: "T => U" ("g'_'_proj'''(_')" [10] 999)

instance R:: type
by intro_classes
instance S:: type
by intro_classes
instance T:: type
by intro_classes
instance U:: type
by intro_classes

axioms

...
ga_notDefBottom: "ALL x. (~ g__defined(x)) = (x = g__bottom)"
...
ga_notDefBottom_1: "ALL x. (~ g__defined'(x)) = (x = g__bottom')"
...
ga_notDefBottom_2: "ALL x. (~ g__defined''(x)) = (x = g__bottom'')"
...
ga_notDefBottom_3: "ALL x. (~ g__defined_3(x)) = (x = g__bottom_3)"
...
ga_totality_1: "ALL x_1. g__defined''(g__inj(x_1)) = g__defined'(x_1)"
ga_totality_2: "ALL x_1. g__defined''(g__inj'(x_1)) = g__defined_3(x_1)"
...
ga_embedding_injectivity: "ALL x. ALL y. g__defined'(x) & g__defined'(y)
  --> g__inj(x) = g__inj(y) & g__defined''(g__inj(x))
  --> x = y & g__defined'(x)"
...

ga_embedding_injectivity_1: "ALL x. ALL y. g__defined_3(x) &
  g__defined_3(y)
  --> g__inj'(x) = g__inj'(y) & g__defined''(g__inj'(x))
  --> x = y & g__defined_3(x)"...
```

Figure 6.5: HETS encoding of Figure 6.3 in IsabelleHOL.

```
1 ProduceAlphabetAndJustificationTheorems:
2   ProducePreAlphabet
3   ProduceEqFunction
4   ProduceProofOfEquivellenceRelation
5   ProduceAlphabetAsQuotient
6   ProduceBarTypes
7   ProduceChooseFunctions
```

Figure 6.6: CSP-CASL sub-algorithm to produce the alphabet and justification theorems.

scribe.

HETS has already been used to translate the CASL specification into Isabelle/HOL code (see Section 6.4). As a result we have a type representing each sort in the specification. We use the `datatype` command to create the disjoint sum of the types representing the sorts. This is done by the following Isabelle/HOL code

```
datatype PreAlphabet = C_A Sort_A | ... | C_N Sort_N
```

where `Sort_A` ... `Sort_N` are variables to be replaced by the types which HETS has produced and `C_A` ... `C_N` are variables standing for unique constructors for each sort. In the theory files we have produced using this algorithm, we have used the unique constructors `C_A, C_B, C_C, ...,` etc. where `A`, `B` and `C` are the names of the sorts.

Performing the algorithm described above on our running example yields the following Isabelle-HOL code:

```
datatype PreAlphabet = C_R R | C_S S | C_T T | C_U U
```

Our running example has four sorts `R`, `S`, `T` and `U`. Hence, we need four constructors, one per sort. We have chosen to name the constructors `C_S` for each sort `S` in order to maintain readability.

6.5.2 Producing the `Eq` function

We now need a function which checks whether two elements of the `PreAlphabet` are equal with respect to the CSP-CASL semantics [Rog06]. This function is produced by the sub-algorithm `ProduceEqFunction`, which we now explore.

This CSP-CASL encoding only works correctly for specifications with *local top elements*. We assume that the specification has *local top elements*, which results in a simpler definition of equality which we now specify as the behaviour of the `eq` function. The static analysis which is performed by the CSP-CASL parser [Gim07] will reject any CSP-CASL specifications which do not have *local top elements*. If the input specification does not have *local top elements* then our algorithm will produce Isabelle/HOL code that has undefined behaviour with respect to the proof scripts when run with Isabelle/HOL. This will most likely manifest itself with the failure of the transitivity proof and the proofs for the integration theorems. However, it is impossible for this code to lead to

inconsistencies in Isabelle/HOL if it is used. If a proof fails, then the user will not be able to continue and will not be able to use the constructions that our algorithm has tried to produce.

The definition of the relation $\sim$ from the CSP-CASL semantics [Rog06] (see Section 5.2) is captured here by the function `eq` and some auxiliary functions. As we are working with a specific sub-sort graph for a given specification, the definition of our `eq` function can be slightly simpler than its semantic counterpart.

As bottom is treated as a normal element in our encoding and the up-casting of bottom (an injection function applied to bottom) always yields the bottom element of the super-sort, its behaviour in the `eq` function is the same as the behaviour for defined elements.

Two elements are equal when they are equal in all super sorts (and possibly the current sort if this is applicable i.e., when both elements are from the same sort). Two elements are never equal if they are from two disconnected components of the sub-sort graph. We use the functions which HETS provides to inject elements to the super sorts and check their equality at the appropriate sort levels.

This is realised via several functions which make use of currying. The functions are dependent on the CSP-CASL specification and hence the algorithm generates different Isabelle/HOL code (which describe the functions) for each CSP-CASL specification used with the algorithm.

Firstly a function called `compare_with_S` is produced for each sort `S` in the CSP-CASL specification. This function will take an element of type `S` and an element of type `PreAlphabet` and checks whether they are equal. These functions are then used in the function `eq`. We will explain these function in reverse order, starting with the `eq` function.

The `eq` function is defined using Isabelle/HOL's primitive recursion scheme (although we do not use it recursively) by the following code:

```
consts
  eq :: "PreAlphabet => PreAlphabet => bool"
primrec
  eq_A:  "eq(C_A ax) = compare_with_A ax"
  ...
  eq_N:  "eq(C_N nx) = compare_with_N nx"
```

where there is one line per sort `T` describing the behaviour of the function in the case that the first parameter is of sort `T`, which is recognised by the constructor `C_T`. Each definition then uses the already defined function `compare_with_T` by calling it with the first element stripped of the constructor (the variable `tx` in the line for sort `T`) and the second element of type `PreAlphabet` (via currying).

The above procedure yields the following code for our running example:

```
consts
  eq :: "PreAlphabet => PreAlphabet => bool"

primrec
  eq_R: "eq(C_R r) = compare_with_R r"
  eq_S: "eq(C_S s) = compare_with_S s"
```

```
  eq_T: "eq(C_T t) = compare_with_T t"
  eq_U:" eq(C_U u) = compare_with_U u"
```

Here we have defined the `eq` function using four auxiliary functions which have to be previously defined (these are explained next). The behaviour of this function is defined by case distinction on the form of the first argument. We have chosen here to use instead of the variable `x`, a lower case version of the sort names for the variables. This convention helps the code maintain its readability.

In order for the `eq` function to be defined, all the functions `compare_with_S` (for each sort `S`) must be previously defined. Such functions check whether the first element is equal to the second element. There is one function per sort `S` in order to deal with the case that the *second* element matches the sort. The first element has been already been stripped of the constructor (in the definition of `eq` and hence is of type `S` for the function `compare_with_S` for sort `S`. The second element is of type `PreAlphabet`. These functions are also defined using the primitive recursion scheme in a similar manner to the `eq` function by the following Isabelle/HOL code for each sort `S`:

```
consts
  compare_with_S :: "Sort => PreAlphabet => bool"
primrec
  compare_with_S_A:  "compare_with_S Sx (C_A Ay) = formula_A
  ...
  compare_with_S_N:  "compare_with_S Sx (C_N Ny) = formula_N
```

where there is again a line defining the behaviour for each sort `T`. The `formula` differs from line to line and is dependent on both the types of the elements we are comparing and the sub-sort relation of the specification. In the definition of the *formula* (`formula_A` ... `formula_N`) we make use of both parameters which were originally of type `PreAlphabet` but have now been stripped of the constructors. Thus we know their underlying types (as each line deals with a specific pair of types from the `PreAlphabet`).

Each formula must return the Boolean true if and only if both elements are equal at all super sort levels (and also the current level if both elements are of the same sort). If both elements are from disconnected components of the graph the formula is simply `False`. As we know the sub-sort relation and the types of both elements the formula can be written explicitly for each case.

We define the general form of the formula – for arbitrary sorts `S` and `T` which are in the same connected component of the sub-sort graph – as:

```
sx = ty & g__inj_s1(sx) = g__inj_t1(ty)
        & ...
        & g__inj_sn(sx) = g__inj_tm(ty)
```

Here `sx` is the first element and `tx` is the second element, we assume here that they are of types `S` and `T`, respectively. Both elements have been stripped of the constructors. If sort `S` is the same sort as `T` then we must perform the first test in the formula (`sx = ty`), otherwise this is omitted. There is a test for each common super-sort for `S` and `T` where `sx` and `ty` are up-cast to that sort and tested for equality at that sort level. To do this we use the injection function which the HETS

encoding provides (see Section 6.4), these are represented above by the functions `g__inj_s1` ... `g__inj_sn` and `g__inj_t1` ... `g__inj_tm`. Each of these tests is inserted into the formula by using the Boolean *and* operation (`&`) to join the tests. This definition causes each formula to be different depending on the sorts and sub-sort relation.

We repeat here the definition of the $\sim$ relation as defined in the CSP-CASL semantics (see Section 5.2). $(s,x) \sim (s',x')$ iff either

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

or

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

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

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

Our implementation of the relation $\sim$ is the `eq` function which uses the `compare_to` functions. Our functions are defined individually for each CSP-CASL specification. Our definition of $\sim$ coincides with the definition from the CSP-CASL semantics [Rog06].

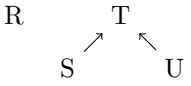

Figure 6.7: Sub-sort graph of the CSP-CASL specification shown in Figure 6.3.

We now present the code for the function `compare_with_S` which is produced for our running example. The code for the other three comparison functions is similar. Figure 6.7 shows the sub-sort graph of the specification of our running example.

```
consts
  compare_with_S :: "S => PreAlphabet => bool"
primrec
  "compare_with_S sx (C_R ry) = False"
  "compare_with_S sx (C_S sy) = ((sx = sy) &
                                 (g__inj(sx) = g__inj(sy)))"
  "compare_with_S sx (C_T ty) = (g__inj(sx) = ty)"
  "compare_with_S sx (C_U uy) = (g__inj(sx) = g__inj'(uy))"
```

We define the function by case distinction on form of the second element. We know the first element is of type `S` and hence we have four cases. In the first case of the first element being of sort `S` and the second of sort `R`, the formula is simply `False` as the sorts are from disconnected components of the graph and hence the elements `sx` and `ry` are never equal. In the second case of the second element being of the underlying sort `S` we have to compare the variables at both sort levels `S` and `T`. We do this by testing if `sx` and `ry` are equal and if their up-casting to sort `T` are equal. The third and forth cases are similar, as both elements are from differing sorts then they cannot be checked

```
1 ProduceProofOfEquivellenceRelation:
2   ProduceProofOfReflexivityOfEq
3   ProduceProofOfSymmetryOfEq
4   ProduceProofOfTransitivityOfEq
```

Figure 6.8: CSP-CASL sub-algorithm to produce header of theory file.

for equality at their local sort level. There is only one common super sort `T`. In the third case, `sx` is up-cast to sort `T` where it is compared for equality with `ty`. In the forth case both `sx` and `uy` are up-cast to sort `T` and compared for equality.

So far we have created a header, produced a data encoding via HETS and now produced a new type `PreAlphabet` with a function which tests whether two elements of the `PreAlphabet` are equal according to the CSP-CASL semantics and the CSP-CASL specification which the algorithm is performed against. Next we produce theorems and proofs which establish the function `eq` as an equivalence relation.

6.5.3 Producing the proof that the function `eq` is an equivalence relation

The algorithm always produces a function `eq` that defines a relation over the type `PreAlphabet`. In this section we prove that the function `eq` actually defines an equivalence relation. The sub-algorithm shown in Figure 6.8) (`ProduceProofOfEquivellenceRelation`) works by calling a number of other sub-algorithms (again we use sub-algorithms to maintain readability). Line 2 produces the theorem and proof that the `eq` function is reflexive. Line 3 produces the theorem and proof that the `eq` function is symmetric. Finally, Line 4 produces the theorem and proof that the function `eq` is transitive. These theorems are then used within Section 6.5.4 to construct the quotient.

6.5.3.1 Producing the proof of reflexivity of `Eq`

The goal of the sub-algorithm `ProduceProofOfReflexivityOfEq` is to produce the theorem and proof that the function `eq` is reflexive. This theorem and proof turns out to be very simple as both the theorem and proof are completely independent of the specification.

The following Isabelle/HOL code is produced by the sub-algorithm `ProduceProofOfReflexivityOfEq`:

```
1 theorem eq_refl: "eq x x"
2   apply(induct x)
3   apply(auto)
4 done
```

This particular piece of code does not alter depending upon which CSP-CASL specification the algorithm is performed upon.

```
1  ProduceProofOfSymmetryOfEq:
2    output: theorem eq_symm: "[| eq x y |] ==> eq y x"
3    output: apply(induct x)
4
5    for (i=0; i<n; i++)
6      output: prefer (i*n)+1
7      output: apply(induct y)
8
9    output: apply(auto)
10   output: done
```

Figure 6.9: CSP-CASL sub-algorithm to produce theorem and proof that `eq` is symmetric.

The main idea behind this code is to induct the variable `x` fully, which will result in finite case distinction as we are inducting over the type `PreAlphabet` which is defined as a non-recursive data type. Once we have fully inducted the variable `x` all the sub-goals (of which there will be one for each sort) will have no variables and Isabelle/HOL will be able to automatically solve them.

Line 1 sets up the theorem with the name `eq_refl` that an object of type `PreAlphabet` is equal with itself. As there is only a single variable the proof is very simple. Line 2 applies induction to the variable `x`, which generates a sub-goal for each sort in the specification because there is a constructor for each sort in the `PreAlphabet`. Line 3 then solves all the sub-goals automatically as all the sub-goals reduce to simple equations of the form `x=x`. Line 4 completes the proof with the Isabelle command `done`.

We have now established the function `eq` is reflexive and move on to establishing that it is also symmetric.

6.5.3.2 Producing the proof of symmetry of `Eq`

The goal of the sub-algorithm `ProduceProofOfSymmetryOfEq` (see Figure 6.9) is to produce the theorem and proof that the function `eq` is symmetric. This theorem and proof turns out to be a little more complex that that of reflexivity (see Section 6.5.3.1) because the proof structure uses induction based on the number of sorts in the specification.

The main idea behind this piece of code is similar to the idea behind the code for the proof of reflexivity. First, we induct all the variables fully, which will leave us with a finite case distinction with no variables. Then we allow Isabelle/HOL to automatically solve all the sub-goals.

Line 2 creates a theorem with the name `eq_symm` that states that the function `eq` is symmetric. Line 3 then applies induction on the variable `x` which results in one sub-goal for each sort because there is a constructor for each sort in the definition of the type `PreAlphabet`. The loop (Line 5) allows us to deal with each of these sub-goals in turn. We now wish to perform induction on the variable `y` in each sub-goal, however as we induct the variable `y` in the first sub-goal, another sub-goal is generated for each sort. We solve this by first pulling each sub-goal to the top of the list (Line 6) and only then applying induction to the top sub-goal (Line 7). At the end of the

induction there are n^2 sub-goals, where n is the number of sorts in the specification. Finally we allow Isabelle/HOL to automatically solve all the sub-goals in Line 9 using a similar technique to the reflexivity proof (see Section 6.5.3.1). Finally, Line 10 completes the proof.

By using this procedure on our running example, we derive the following code:

```
1 lemma eq_symm: "[| eq x y |] ==> eq y x"
2 apply(induct x)
3 prefer 1 apply(induct y)
4 prefer 5 apply(induct y)
5 prefer 9 apply(induct y)
6 prefer 13 apply(induct y)
7 apply(auto)
8 done
```

After the induction on `x` is performed in Line 2, we are left with 4 sub-goals. For each of these sub-goals we must induct the variable `y`, we do this in Lines 3 to 6. Finally, we solve all n^2 sub-goals by using the automatic proof method (Line 7) and complete the proof in Line 8 using the command `done`. We now show the proof state after Line 3 has been executed.

```
proof (prove): step 3

fixed variables: x, y

goal (lemma (eq_symm), 7 subgoals):
 1. !!R Ra. eq (C_R Ra) (C_R R) ==> eq (C_R R) (C_R Ra)
 2. !!S R. eq (C_R R) (C_S S) ==> eq (C_S S) (C_R R)
 3. !!T R. eq (C_R R) (C_T T) ==> eq (C_T T) (C_R R)
 4. !!U R. eq (C_R R) (C_U U) ==> eq (C_U U) (C_R R)
 5. !!S. eq (C_S S) y ==> eq y (C_S S)
 6. !!T. eq (C_T T) y ==> eq y (C_T T)
 7. !!U. eq (C_U U) y ==> eq y (C_U U)
```

This shows why the `prefer` commands are necessary. We have to first pull sub-goal 5 to the top of the sub-goal list before we can apply induction to the variable `y`. This is the purpose of Line 4 of the proof script.

We have now produced the theorem and proof that the function `eq` is symmetric.

6.5.3.3 Producing the proof of transitivity of `Eq`

We need one final proof to conclude that the function `eq` forms an equivalence relation. The theorem and proof of transitivity turns out to be more complicated due to the bottom elements that HETS has introduced. The sub-algorithm `ProduceProofOfTransitivityOfEq` first needs to generalise two sets of axioms that HETS has produced. Once this has been done the proof of transitivity of `eq` can be completed. The first set of axioms concerns the decomposition of the larger

injection functions into two smaller injection functions, while the second set of axioms concerns the injectivity of the injection functions.

We first deal with the decomposition lemmas. HETS produces axioms of the form (where `n` is a natural number and is omitted in the case of zero):

```
1 ga_transitivity_n : "ALL x. g__defined(x)
2   --> g__inj''(g__inj(x)) = g__inj'(x)
3     & g__defined''(g__inj''(g__inj(x)))"
```

The exact injection and definedness functions will change depending on the specification. This particular axiom states that the injection function `g__inj'` can be decomposed into the injection function `g__inj` followed by the the injection function `g__inj''` when the arguments are defined. As we have a unique bottom element for each sort which is preserved by the injection functions, these axioms generalise so that they also hold for the bottom elements, hence they hold for all elements. The generalisation of these axioms are the first set of lemmas that need to be produced in order to prove transitivity of the function `eq`.

For each of the transitivity axioms that HETS produced, we produce the following Isabelle/HOL code (where `n` is a natural number and is omitted in the case of zero):

```
1  lemma inj_decomposition_n: "g__inj''(g__inj(x)) = g__inj'(x)"
2      apply(case_tac "g__defined''(g__inj''(g__inj(x)))")
3
4    (* Case 1 *)
5      apply(subgoal_tac "g__defined(x)")
6      apply(insert ga_transitivity_n)
7      apply(simp)
8      apply(simp add: 'ga_totality_axioms')
9
10   (* Case 2*)
11     apply(subgoal_tac "~ g__defined''(g__inj'(x))")
12     apply(simp add: 'ga_notDefBottom_axioms')
13     apply(simp add: 'ga_totality_axioms')
14 done
```

Where the exact injection functions in Line 1 should match the HETS axiom that we are generalising (we work here with the HETS axiom stated above).

The lemma is called `inj_decomposition_n` where `n` should match the HETS axiom number (or omitted in the case of zero) and states that the injection function `g__inj'` can be decomposed into the injection function `g__inj` followed by the the injection function `g__inj''` for any element `x`.

We prove this by performing a case distinction upon the definedness of the left hand side. This is performed in Line 2 where the exact definedness function will be different for each lemma produced, but only one definedness function will be type correct in this context.

Case 1: In the first case we know that the left hand side is defined.

In order for the right hand side to be equal to the left hand side, the right hand side must also be defined. By the totality of the injection functions and the assumption that the right hand side is defined, we can conclude that `x` is also defined, thus we add this as a sub-goal in Line 5 (the exact definedness function will be different for each lemma produced, but only one definedness function will be type correct in this context). This adds an assumption to the first sub-goal. We can then use the HETS axiom which we are generalising and simplification to prove that the decomposition holds for defined elements, lines 6 and 7. Finally we have to prove that the sub-goal that we introduced on Line 5 holds. We do this by performing simplification with the added axioms `ga_totality_axioms` which state that the injections applied to defined elements yield defined elements (Line 8).

Case 2: In the second case we know that the left hand side is undefined. Hence, the right hand side must also be undefined if the equality holds. We assume the right hand side is undefined by adding this as a sub-goal in Line 11 (again the exact definedness function will change, but only one will be type correct in this context). As both sides are undefined, and there is a single undefined element (bottom) we know the equality holds. This is proven in Line 12 by adding the axioms that state the bottom element is unique (`ga_notDefBottom_axioms`). Finally, we prove the sub-goal which we added as an assumption in line 11 in the same way as we proved the sub-goal in the first case, by performing simplification with the totality axioms (Line 13).

Our running example does not have any transitivity axioms generated by HETS. The code and proof produced by the above procedure is very similar to the next procedure, for which our running example does produce code.

We now have all the HETS transitivity axioms generalised as lemmas called `inj_decomposition_n` (where `n` is a natural number and is omitted in the case of zero). These lemmas hold for all elements and not only defined elements.

The second set of lemmas that need to be produced are generalisations of the injectivity lemmas that HETS produced. The reasons for this are identical to the reasons for the transitivity axioms above and the production of the lemmas is almost identical to the production of the decomposition lemmas above.

HETS will have produced several axioms of the form:

```
1 ga_embedding_injectivity_n : "ALL x. ALL y.
2   g__defined(x) & g__defined(y)
3   --> g__inj(x) = g__inj(y) & g__defined'(g__inj(x))
4   --> x = y & g__defined(x)"
```

Where `n` is a natural number and is omitted in the case of zero. The exact injection and definedness functions will change depending on the specification. This particular axiom states that if `x` and `y` are defined, the injection of `x` equals the injection of `y` and the result of the injection on `x` is defined then `x` and `y` must be equal (i.e., the function `inj` is injective when both arguments are defined). As we have a unique bottom element for each sort which is preserved by the injection functions, these axioms generalise so that they also hold for the bottom elements, hence they hold

for all elements. The generalisation of these axioms are the second set of lemmas that need to be produced in order to prove transitivity of the function `eq`.

For each of the embedding injectivity axioms that HETS produced, we produce the following Isabelle/HOL code (where `n` is a natural number and is omitted in the case of zero):

```
1  lemma injectivity_n: "[| g__inj(x) = g__inj(y) |] ==> x = y"
2      apply(case_tac "g__defined'(g__inj(x))")
3
4    (* Case 1 *)
5      apply(subgoal_tac "g__defined(x)")
6      apply(subgoal_tac "g__defined(y)")
7      apply(insert ga_embedding_injectivity_n)
8      apply(simp)
9      apply(simp add: 'ga_totality_axioms')
10     apply(simp (no_asm_use) add: 'ga_totality_axioms')
11
12   (* Case 2*)
13     apply(subgoal_tac "~ g__defined(x)")
14     apply(subgoal_tac "~ g__defined(y)")
15     apply(simp add: 'ga_notDefBottom_axioms')
16     apply(simp add: 'ga_totality_axioms')
17     apply(simp (no_asm_use) add: 'ga_totality_axioms')
18 done
```

Where the exact injection function in Line 1 should match the HETS axiom that we are generalising (we work here with the HETS axiom stated above).

The lemma is called `injectivity_n` where `n` should match the HETS axiom number (or omitted in the case of zero) and states that the injection function `g__inj` is injective for any element `x`.

We prove this by performing a case distinction upon the definedness of the injection of `x` (i.e., `g__inj(x)`) within the assumption. This is performed in Line 2 where the exact definedness function will be different for each lemma produced, but only one definedness function will be type correct in this context.

Case 1: In the first case we know that the injection of `x` is defined and we assume that the injection of `x` equals the injection of `y`. By the totality of the injection functions and the assumption that the injection of `x` equals the injection of `y`, we can conclude that both `x` and `y` must also be defined. We add these as assumptions in Lines 5 and 6 (again the exact definedness function will be different for each lemma produced, but only one will be type correct in this context). We will have to prove both of these sub-goals at the end of this case. These sub-goals have added the assumptions that `x` and `y` are both defined to the first sub-goal. We can then use the HETS axiom which we are generalising and simplification to prove the injection function is injective for defined elements, lines 7 and 8. Finally we have to prove that the sub-goals that we introduced on Lines 5 and 6 hold. We do this by performing simplification with the added axioms (`ga_totality_axioms`) that state that the injection functions applied to defined elements yield defined elements (Lines 8 to 10).

However, in Line 10 we need to stop the simplifier from using simplified assumptions to simplify other assumptions and conclusions as this would result in non-terminating loops. We achieve this by invoking the simplifier with the option `no_asm_use`.

Case 2: In the second case we know that the injection of `x` is undefined and we assume that the injection of `x` equals the injection of `y`. By the totality of the injections and the assumption that the injection of `x` equals the injection of `y`, we can conclude that both `x` and `y` must be undefined as well. We add the facts `x` and `y` must be undefined as sub-goals in Lines 13 and 14 which become assumptions in the first sub-goal (again the exact definedness function will change, but only one will be type correct in this context). These will need to be proven at the end of this case. As `x` and `y` are undefined, and there is a single undefined element (bottom) we know the equality between `x` and `y` holds. This is proven in Line 15 by adding the axioms (`ga_notDefBottom_axioms`) that state the bottom element is the unique undefined element for each sort. We finally prove the sub-goals (lines 16 and 17) in the same way as we proved the sub-goals in case 1, by performing simplification with the totality axioms.

Our running example has exactly two embedding injectivity lemmas produced by HETS, one for each injection function. Hence we must produce two generalised injectivity lemmas. We present here only one of these functions as the other is very similar. The following code is produced for the corresponding HETS axiom `ga_embedding_injectivity` (Line 44 in Figure 6.5).

```
lemma injectivity: "[| g__inj(x) = g__inj(y) |] ==> x = y"
    apply(case_tac "g__defined''(g__inj(x))")

  (* Case 1 *)
    apply(subgoal_tac "g__defined'(x)")
    apply(subgoal_tac "g__defined'(y)")
    apply(insert ga_embedding_injectivity)
    apply(simp)
    apply(simp add: ga_totality_1 ga_totality_2)
    apply(simp (no_asm_use) add: ga_totality_1 ga_totality_2)

  (* Case 2*)
    apply(subgoal_tac "~ g__defined'(x)")
    apply(subgoal_tac "~ g__defined'(y)")
    apply(simp add: ga_notDefBottom ga_notDefBottom_1
                    ga_notDefBottom_2 ga_notDefBottom_3)
    apply(simp add: ga_totality_1 ga_totality_2)
    apply(simp (no_asm_use) add: ga_totality_1 ga_totality_2)
done
```

This code was produced by simply following the steps described above.

We now have all the HETS injectivity axioms generalised as lemmas called `injectivity_n` (where `n` is a natural number and is omitted in the case of zero). These lemmas hold for all elements and not only defined elements.

```
1  ProduceProofOfTransitivityOfEq:
2    ProduceDecompositionLemmas
3    ProduceInjectivityLemmas
4
5    output: theorem eq_trans: "[| eq x y ; eq y z |] ==> eq x z"
6    output: apply(induct x)
7
8    for (i=0; i<n; i++)
9      output: prefer (i*n)+1
10     output: apply(induct y)
11
12   for (i=0; i<n^2; i++)
13     output: prefer (i*n)+1
14     output: apply(induct z)
15
16   output: apply(auto simp add: 'Injectivity_lemmas
17                                 Decomposition_lemmas')
18   output: done
```

Figure 6.10: CSP-CASL sub-algorithm to produce theorem and proof that `eq` is transitive.

With both of these sets of lemmas available, proving transitivity of the function `eq` becomes relatively easy and follows the same proof approach as the proof of symmetry of the function `eq`, although its a little more complex. This is because there is an additional variable involved, hence more induction needs to take place.

Figure 6.10 shows the full sub-algorithm `ProduceProofOfTransitivityOfEq` that produces the transitivity proof. Lines 2 and 3 produce the lemmas that we have described above. Line 5 establishes the theorem that the function `eq` is transitive with the name `eq_trans`. In order to prove this we first induct all the variables fully, which will leave us with a finite case distinction with no variables. Then we allow Isabelle/HOL to automatically solve all the sub-goals.

Line 6 then applies induction on the variable `x` which results in one sub-goal for each sort as there is a constructor for each sort in the definition of the type `PreAlphabet`. The loop (Line 8) allows us to deal with each of these sub-goals in turn. We now wish to perform induction on the variable `y` in each sub-goal, however as we induct the variable `y` in the first sub-goal, another sub-goal is generated for each sort. We solve this by first pulling each sub-goal to the top of the list (Line 9) and only then applying induction to the top sub-goal (Line 10). At the end of the induction there is n^2 sub-goals, where n is the number of sorts in the specification.

However, each of these sub-goals contains the variable `z`. Thus we need more induction to create our complete finite case distinction. Another Loop in Line 12 allows us to apply induction to the variable `z` in the same way as we applied induction to the variable `y` in each sub-goal. After this loop we have n^3 sub-goals with no variables which forms our finite case distinction. All the sub-goals involve showing simple equalities between injection of `x` and `z`.

Finally we allow Isabelle/HOL to automatically solve all the sub-goals in Line 16 by adding the in-

jectivity and decomposition lemmas which we provided earlier as part of this sub-algorithm. These allow each of the larger injections to be substituted with two smaller injections. The smallest injections have been proven to be injective and hence Isabelle/HOL can show that the equalities within the sub-goals hold. Line 18 then completes the proof by outputting the Isabelle/HOL command `done`.

The following code is produced for the transitivity theorem for the specification of our running example.

```
1  lemma  eq_trans: "[| eq x y ; eq y z |] ==> eq x z"
2  apply(induct x)
3  prefer 1 apply(induct y)
4  prefer 5 apply(induct y)
5  prefer 9 apply(induct y)
6  prefer 13 apply(induct y)
7  prefer 1 apply(induct z)
8  prefer 5 apply(induct z)
9  prefer 9 apply(induct z)
10 prefer 13 apply(induct z)
11 prefer 17 apply(induct z)
12 prefer 21 apply(induct z)
13 prefer 25 apply(induct z)
14 prefer 29 apply(induct z)
15 prefer 33 apply(induct z)
16 prefer 37 apply(induct z)
17 prefer 41 apply(induct z)
18 prefer 45 apply(induct z)
19 prefer 49 apply(induct z)
20 prefer 53 apply(induct z)
21 prefer 57 apply(induct z)
22 prefer 61 apply(induct z)
23 apply(auto simp add: injectivity injectivity_1)
24 done
```

Here we have to induct according to the number of sorts. After the final induction (Line 22) there are 4^3 sub-goals. All these sub-goals are solved automatically by adding the generalised injectivity lemmas that were previously produced by this sub-algorithm.

We have now produced the theorem and proof that the function `eq` is transitive hence we have successfully produced all the proofs necessary to conclude that the function `eq` forms an equivalence relation.

We illustrate this proof idea by a concrete example. Consider the sub-sort structure shown in Figure 6.11 where the functions shown are the injections functions which HETS provides[1].

After applying the necessary induction to the proof goals, we obtain n^3 sub-goals, where n represent the number of sorts in the specification. As we have 4 sorts in our example, there will be 64 sub-

[1] We use the infix notation of $\sim$ in place of the Isabelle function `eq`.

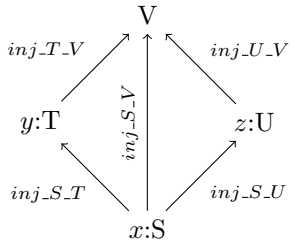

Figure 6.11: Example of a possible sub-sort structure with injection functions.

goals to discharge. One of the resulting sub-goals will be

$$x \sim y \wedge y \sim z \Rightarrow x \sim z$$

where x, y and z are variables of the types S, T and U, respectively. There will be one sub-goal for each permutation of x, y and z.

Expanding the definition of $x \sim z$ yields two new sub-goals: $inj\_S\_U(x) = z$ and $inj\_S\_V(x) = inj\_U\_V(z)$. We focus here on proving $inj\_S\_V(x) = inj\_U\_V(z)$. This equation means that x is equal to z in the sort V. Expanding the definition of $x \sim y$ we obtain the equation $inj\_S\_V(x) = inj\_T\_V(y)$. From $y \sim z$ we obtain the equation $inj\_T\_V(y) = inj\_U\_V(z)$. These two facts together yield the equation $inj\_S\_V(x) = inj\_U\_V(z)$. This proves one part of the goal, the other can be proven in a similar way using the fact that the functions we use are injections (these axioms are provided by HETS).

Isabelle/HOL can carry out all these proofs fully automatically, provided the simplifier is enriched with the injection and decomposition axioms that were previously produced, see lines 16 and 17 of Figure 6.10.

Interestingly this is a new method of proving transitivity of the `eq` function. The method from the proof in [Rog06], uses the facts that the specification has local top elements and that an injection followed by a projection yields the original value (i.e., $\texttt{pr}(\texttt{inj}(x)) = x$). The proof method we have does not use the projection functions at all, but instead uses the facts that the injection functions (functions for "casting" elements to different sorts) are indeed injective functions (HETS creates axioms to control this). It seems that it is easier to use this proof method within Isabelle/HOL than the proof method used in [Rog06].

6.5.4 Producing the alphabet as a quotient

At this stage the type `PreAlphabet` and the `eq` function are both declared. Theorems and proofs of reflexivity, symmetry and transitivity of the function `eq` have also been established. The sub-algorithm `ProduceAlphabetAsQuotient` can now use such established theorems to construct the alphabet of communications – the type `Alphabet`.

The sub-algorithm `ProduceAlphabetAsQuotient` produces the following Isabelle/HOL code[2]:

[2]Here the Isabelle/HOL command for the symbol $\sim$ is $\backslash <$ `sim` $>$, there is no equivalent ASCII code available.

```
1  instance PreAlphabet::eqv
2  by intro_classes
3
4  defs (overloaded)
5    preAlphabet_sim_def : "x \<sim> y == eq x y"
6
7  instance PreAlphabet::equiv
8  apply(intro_classes)
9  apply(unfold preAlphabet_sim_def)
10 apply(rule eq_refl)
11 apply(rule eq_trans)
12 apply(auto)
13 apply(rule eq_symm)
14 apply(simp)
15 done
```

This code is completely independent of the specification. Hence, it does not alter depending on what specification the algorithm is performed against.

Line 1 states that the type `PreAlphabet` is an instance of the axiomatic type class `eqv`. This is proven by Line 2. Now that the type `PreAlphabet` is an instance of the type class `eqv` we can provide a definition for the relation $\sim$. Lines 4 and 5 states that the relation $\sim$ is overloaded and `x` is in relation to `y` if the function `eq x y` evaluates to true. This is how we created the relation $\sim$ from the existing function `eq`.

Now that we have provided a definition for $\sim$ we can instantiate `PreAlphabet` as an instance of the axiomatic type class `equiv` – Line 7. This comes with a proof obligation that the $\sim$ relation is actually an equivalence relation. Line 8 begins the proof, which leaves us with three sub-goals where we must prove $\sim$ is reflexive, symmetric and transitive. Line 9 then unfolds the definition of $\sim$, which results in the sub-goals being transformed into proof obligations that the function `eq` is reflexive, symmetric and transitive. Lines 10 to 14 then apply the previously established theorems (see Section 6.5.3) in the correct order which successfully discharges all proof obligations. Finally Line 15 completes the proof with the Isabelle command `done`.

As the type `PreAlphabet` is a member of the axiomatic type class `equiv` we can use the built in type of `'a quot` to form the quotient of the `PreAlphabet`, i.e., the type `PreAlphabet quot`. The alphabet of communications can now be established using the following Isabelle/HOL code:

```
types Alphabet = "PreAlphabet quot"
```

This creates a new type `Alphabet` which is a type synonym of the type `PreAlphabet quot`. The type `Alphabet` will be automatically expanded in Isabelle/HOL into `PreAlphabet quot` every time it is used and unfortunately will not be converted back to `Alphabet` when shown to the user.

We have finally established a type for the alphabet of communications. This can now be used as a

parameter for the CSP-Prover type `'a proc`. The type `Alphabet proc` is the type of all CSP processes over the alphabet of communications. This alphabet is built from the CASL specification and captures all sorts and functions from the specification. The equivalence class of any term of the CASL specification can now be used as a communications event in a CSP process of type `Alphabet proc`.

6.5.5 Producing the bar types

Now that we have our alphabet of communications (the type `Alphabet`), we need a mechanism to allow us to address certain subsets of this alphabet. We need a subset for each original sort. These subsets are necessary for certain CSP-Prover operations. For instance CSP-Prover's *internal prefix choice* operator, which is written as:

```
! x : A -> P(x)
```

Here `x` is a variable, `A` can be any subset of the alphabet and `P(x)` is a Process. When writing CSP-CASL specifications only sort symbols can be used for the set `A`. Hence, we need a subset of the alphabet for each original sort in order to use this operation over the the alphabet.

The sub-algorithm `ProduceBarTypes` produces for each sort `S` a type `S_Bar`, which is a set of type `Alphabet` produced by:

```
1 typedef S_Bar = "{x::Alphabet. EX (y::S). x = class(C_S y)}"
2 by auto
```

Line 1 produces a new type `S_Bar` where the elements are exactly the equivalence class of each element of sort `S` wrapped in the appropriate constructor (i.e., `C_S` for each sort `S`). This comes with a proof obligations that the type is non-empty. This is discharged using the `auto` proof method (Line 2).

Isabelle/HOL has now created a set and a type called `S_Bar`, both of which coincide. Also two conversion functions have been defined, namely

```
Abs_S_Bar :: S_Bar => Alphabet
Rep_S_Bar :: Alphabet => S_Bar
```

for converting between the set and the type.

We can now write processes that use the new subsets of the alphabet of communications. For example, the process

```
? x:S_Bar -> x -> SKIP
```

can be written, where `S_Bar` is a set of `Alphabet`. Hence `x` is an element of type `Alphabet` and can be used as a valid communication event.

As we have four sorts in our running example (Figure 6.3). We must produce four new types mirroring the sorts. The following Isabelle/HOL code is produced by applying the above steps to the specification.

```
typedef R_Bar = "{x::Alphabet. EX (r::R). x = class(C_R r)}"
by auto

typedef S_Bar = "{x::Alphabet. EX (s::S). x = class(C_S s)}"
by auto

typedef T_Bar = "{x::Alphabet. EX (t::T). x = class(C_T t)}"
by auto

typedef U_Bar = "{x::Alphabet. EX (u::U). x = class(C_U u)}"
by auto
```

This construction allows us to write the following process

```
? x : R_Bar -> x -> SKIP
```

in CSP-Prover which will be of type `Alphabet proc`. This would correspond to the process

```
? x : R -> x -> SKIP
```

in the CSP-CASL specification.

6.5.6 Producing the choose functions

We now have new types `S_Bar` for each sort `S`. However, HETS has produced functions which operate on the underlying types (e.g., `S`) not the bar variants (e.g., `S_Bar`). There is a type problem when applying values to the bar types to the functions which HETS has produced. We solve this by creating for each sort `S` a function `choose_S`. This is the purpose of the sub-algorithm `ProduceChooseFunctions`. For each sort `S` we produce the following code:

```
consts
  choose_S :: "S_Bar => S"
axioms choose_S: "forall x y .
  (choose_S x = y) = ((class(C_S y)) = (Rep_S_Bar x))"
```

We have had to provide an axiomatic definition of this functions. The function `choose_S` allows us to take an element of `S_Bar` (i.e., an equivalence class of the term `C_S x` for some `x` of type `S`) and return the underlying element `x`. With these function available we can now use values of the bar types as parameters to the existing functions that HETS produced. Such results can then be packaged up to become elements of type `Alphabet` and hence valid communications events within processes.

6.6 Producing the integration theorems

In order to make working with process refinement proofs in our type system easy, Integration Theorems must be provided. The form of such theorems have been analysed in [Rog06]. There they discovered three types of integration theorems are necessary for semantical proofs. We produce here (using the sub-algorithm `ProduceIntegrationTheorems`) the first integration theorem. This allows a test of equality between two elements of type `Alphabet` to be reduced to a test on the underlying data specified in the HETS encoding which represents the data-part of the specification.

The following code is produced for each pair of sorts `S` and `T` (where `n` is a natural number and must be incremented for each lemma):

```
1 lemma integration_theorem_n: "(class(C_S x) = class(C_T t)) =
2                               (injection(x) = injection(y))"
3 apply(simp add: quot_equality)
4 apply(unfold preAlphabet_sim_def)
5 apply(auto simp add: 'Injectivity_lemmas
6                       Decomposition_lemmas')
7 done
```

Here the constructors `C_S` and `C_T` are the corresponding constructors for each sort `S` and `T`, respectively. The test on the R.H.S is a test of equality between the elements `x` and `y` at the level of their top-most sort, this is achieved by replacing the functions `injection_s` and `injection_t` with the corresponding injection functions produced by HETS which up-cast values to their top-most sort.

If both elements are from separate connected components of the sub-sort graph, this test is replaced by the Boolean value `False`. If there is more than one top-most sort available (i.e., in the case of isomorphic sorts) either sort level can be used in the test.

These theorems can be automatically proven by unfolding the definition of the equivalence classes and the $\sim$ relation – lines 3 and 4. Followed by invoking the automatic proof method (lines 5 and 6) with the added axioms that were already used as part of the transitivity proof (see Section 6.5.3.3).

In order to use the choose functions easily, we must provide some support theorems. For each sort `S` we produce the following Isabelle/HOL code:

```
lemma choose_S_lemma : "[| (C_S x) \<sim> (C_S y) |] ==> x = y"
apply(unfold preAlphabet_sim_def)
apply(auto)
done
```

Where the constructor `C_S` is the corresponding constructor for the sort `S`. This lemma states that if two elements of type `PreAlphabet` are from the same sort and are related by $\sim$ then the underlying elements are equal. This proof is necessary to check that the axiomatic definition of the choose function is legal and also to allow easier proofs when using the `choose` functions. This proof can be completed automatically by first unfolding the definition of $\sim$ (Line 2) and then unfolding and simplifying the definitions of the `eq` function (Line 3).

With these two types of integration theorems available, semantical proofs become more reasonable, as questions on the alphabet of communications can be reduced to questions on the underlying data specified by CASL. These are required to keep the process refinement proofs small, clear and manageable.

6.7 Producing the data theorems place holder

The algorithm `ProduceDataTheoremsPlaceHolder` simply produces some comments suggesting that the user should re-place these comments with *Data Theorems* that are specific to the problem area of the user. This is purely to guide the user and help maintain a logical structure to the resulting theory file.

6.8 Producing the process translations

We have produced a Haskell program that uses the CSP-CASL parser [Gim07] to produce process translations. Currently this translation process works only for a single sorted specification and processes with only constants as communications. It is straightforward future work to extend this program to full CSP-CASL. The point of this section is to show the feasibility of our approach.

```
data PROCESS
    = Skip Range
    | Stop Range
    | PrefixProcess EVENT PROCESS Range
    | ExternalChoice PROCESS PROCESS Range
    | InternalChoice PROCESS PROCESS Range
    | SynchronousParallel PROCESS PROCESS Range
    ...
```

Figure 6.12: Part of the abstract syntax for the processes of CSP-CASL parser [Gim07].

Figure 6.12 shows part of the Haskell abstract data type that is used to represent processes in the CSP-CASL parser. This data type is recursive and resembles what we expect of a type capturing the grammar of CSP. The CSP-CASL parser has pretty print functions available which transform values of such types into strings. Our translation program is a modification of the pretty print functions that accompany the CSP-CASL parser.

We will present our pretty print function for the following datatype used by CSP-CASL parser:

```
data PROC_ITEM = ProcDecl PROCESS_NAME PROC_ARGS PROC_ALPHABET
               | ProcEq PARM_PROCNAME PROCESS
                 deriving Show
```

Here the type `PARM_PROCNAME`, `PROC_ARGS`, and `PROC_ALPHABET` are all types defined by the CSP-CASL parser. The interesting case here is the `ProcEq` case. This is where our pretty print functions transform parse trees representing process equations into valid CSP-Prover code.

The following Haskell code is part of the original pretty print functions that accompany the CSP-CASL parser:

```
printProcItem :: PROC_ITEM -> Doc
printProcItem (ProcEq pn p) =
    (pretty pn) <+> equals <+> (pretty p)
```

We have modified this pretty print function to output valid CSP-Prover syntax. The modified version is:

```
printProcItem :: PROC_ITEM -> Doc
printProcItem (ProcEq pn p) =
    (text "consts") <+> (pretty pn) <+>
      (text ":: \"Alphabet proc\"") $+$
    (text "defs") <+> (pretty pn) <> (text "_def: \"") <+>
      (pretty pn) <+> (text "==") <+> (pretty p) <> (text "\"")
```

Here we only show the case of printing the values with the constructor `ProcEq`. The function `printProcItem` does not return strings, but a value of type (`Doc`) which represents a string. The function `<+>` appends two values of type `Doc` whilst adding a space between them, while the function `<>` appends two values of type Doc without adding a space between them. The function `$+$` appends two values of type `Doc` whilst adding a line break between them. Finally the function `text` converts a string to a value of type `Doc`. The functions handling the type `Doc` are defined within the HETS framework.

The idea behind the overall pretty print function is to transform process equations, e.g., `P = a -> b -> Q`, into processes that use our alphabet of communications whilst being written in CSP-Prover syntax. We do this by declaring the process name e.g., in our example `P`, as a new constant of type `Alphabet proc`. The R.H.S of the equation, in our example `a -> b -> Q`, is then translated separately and is bound by an Isabelle/HOL definition to the process name. The communications of the R.H.S are the equivalence classes of the original communications after they have been "wrapped" up in the appropriate constructor of the type `PreAlphabet`.

As an example we translate the following processes specified in CSP-CASL syntax:

```
P = a -> b -> SKIP
Q = P |~| c ->STOP
R = P [] (c -> SKIP)
```

We assume here that all constants are of sort `S`. We run our process translator on the processes, this yields the following CSP-Prover code:

```
consts P :: "Alphabet proc"
defs P_def: " P == class(C_S a) -> class(C_S b) -> SKIP"
consts Q :: "Alphabet proc"
```

```
defs Q_def: " Q == P ( ) |~| class(C_S c) -> STOP"
consts R :: "Alphabet proc"
defs R_def: " R == P ( ) [+] class(C_S c) -> SKIP"
```

Here the symbol `[+]` is CSP-Prover's syntax for the *external choice* operator. This code can now be directly inserted into the process part of the theory file produced by our algorithm.

6.9 Dependencies

The following table shows the dependencies of the PreAlphabet construction and the *Integration Theorems*. $T(D)$ denotes that the theorem is dependent on the parameter in the column heading, while $T(I)$ expresses that the theorem is independent of the parameter in the column heading, and similar $P(\_)$ expresses the dependencies of the proofs on the parameter in the column heading.

	Specification	# of Sorts	Sub-sort Structure
PreAlphabet Construction	$T(D)$ / $P(D)$	$T(I)$ / $P(D)$	$T(D)$ / $P(D)$
eq_Reflexivity	$T(I)$ / $P(I)$	$T(I)$ / $P(I)$	$T(I)$ / $P(I)$
eq_Symmetry	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$	$T(I)$ / $P(I)$
eq_Transitivity	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$
Integration Theorems	$T(D)$ / $P(D)$	$T(I)$ / $P(D)$	$T(D)$ / $P(D)$

The reflexivity property of the `eq` relation is completely independent of the specification whereas the proof of symmetry relies only on the number of sorts in the specification and the proof of transitivity relies on the number of sorts and the sub-sort structure. The integration theorems are the most dependent on the specification. All these proofs can be automatically generated our algorithm.

Chapter 7

Proofs in CSP-CASL-Prover

Contents

7.1 The four core examples

Four core examples have been discussed in Section 5.1.1. We present the proofs of the equality between the relevant processes according to the CSP-CASL semantics using CSP-CASL-Prover.

7.1.1 Core example 1

Figure 5.1 shows the first core example which a CSP-CASL prover should be able to prove. We must be able to show that the process `tcs1 = c -> SKIP || d -> SKIP` specified in the CSP-CASL specification is equivalent to the process `STOP`. We have performed our algorithm on the CSP-CASL specification in Figure 5.1. This lead to a theory file where we proved that this equality holds.

Figure 7.1 shows the final proof script of the process equality. Line 1 establishes the theorem. Line 3 applies the CSP-Prover tactic `cspF_hsf_tac`. Finally we have to apply a theorem which we

```
1 theorem "c -> SKIP || d -> SKIP =F STOP"
2 apply(tactic {*cspF_hsf_tac 1 *})
3 apply(auto simp add: Syntactic_Integration_Theorem_A)
4 done
```

Figure 7.1: CSP-CASL proof for the core CSP-CASL example 1 (Figure 5.1).

```
1 theorem "(class (C_S c)) -> SKIP || (class(C_T d)) -> SKIP =F(
2          class(C_S c)) -> SKIP"
3 apply(tactic {*cspF_hsf_tac 1 *} |
4       auto simp add: Syntactic_Integration_Theorem_B)+
5 done
```

Figure 7.2: CSP-CASL proof for the core CSP-CASL example 2 (Figure 5.2).

believe is a form of a *Syntactical Integration Theorem* – which we had to prove manually for now. Simplification with this theorem completes the proof.

7.1.2 Core example 2

Figure 5.2 shows the second core example which a CSP-CASL prover should be able to prove. The challenge here is to show that the same process as in the first core example (`tcs1 = c -> SKIP || d -> SKIP`) is equivalent to the process `c -> SKIP` when we have the specification shown in Figure 5.2.

Figure 7.2 shows the proof of this challenge using CSP-CASL prover. It is very similar to the proof for core example 1. We apply CSP-Prover's tactics whilst applying a similar theorem as we proved for the first example.

7.1.3 Core example 3

Figure 5.3 shows the third core example for a CSP-CASL prover. This is the most complicated example of the four. We must prove that the process `? x:S -> f(x) -> SKIP [| T |] ? y:T -> (if def y then P else Q)` is equivalent to the process `? x:S -> f(x) -> (SKIP [| T |] Q)`. This involves us having to use our first *Data Theorem*. It relies on the fact that the result of the function `f` is always undefined.

Figure 7.3 shows the proof script for this example. We have deviated here from the algorithm documented throughout this thesis by not using the choose functions, but instead using a projection function in combination with the built in `pick` function on equivalence classes. We have chosen to do this in order to explore other techniques for a CSP-CASL-Prover. Both this method and our documented algorithm produce working theory files that can be used to prove this process equivalence easily. The proof here is quite simple, it applies CSP-Prover's tactics along with automatic theorem proving using some *Syntactical Integration Theorems* and the *Data Theorem* that we had to provide.

7.1.4 Core example 4

We conclude this section with the forth core example shown in Figure 5.4. Here the challenge is to prove that the process `f(a) -> SKIP || g(c) -> SKIP` is equivalent to the process `g(c) -> SKIP` according to the CSP-CASL semantics.

```
theorem "? x : S_Bar ->  class(C_T (f (proj_S (pick x)))) -> SKIP
         |[T_Bar]|
         ? y : T_Bar -> IF g__defined (proj_T (pick y)) THEN
           P ELSE Q
         =F
         ? x : S_Bottom -> class(C_T (f (proj_S (pick x)))) ->
         (SKIP |[T_Bottom]| Q)"

apply(tactic {*cspF_hsf_tac 1 *} |
  auto simp add: Syntactic_Integration_Theorem_C
                 Syntactic_Integration_Theorem_D
                 Data_Theorem_A)+
done
```

Figure 7.3: CSP-CASL proof for the core CSP-CASL example 3 (Figure 5.3).

```
theorem "f(a) -> SKIP || g(c) -> SKIP =F class(C_C(g c)) -> SKIP"
  apply(tactic {*cspF_hsf_tac 1 *} |
        auto simp add: Syntactic_Integration_Theorem_E)+
done
```

Figure 7.4: CSP-CASL proof for the core CSP-CASL example 4 (Figure 5.4).

The proof script shown in Figure 7.4 completes our proof on the four core example. This proof is essentially the same as the other examples and uses CSP-Prover's powerful tactics along with a *Syntactic Integration Theorem*.

It is future work to identify the forms and automatic proofs of the *Syntactical Integration Theorems*. We have shown that our approach can deal easily with all four core examples using essentially the same proof method.

7.2 EP2 dialog is deadlock free

As an application of CSP-CASL-Prover we prove deadlock freedom in an industrial setting. Here we prove deadlock freedom of a dialog in the EP2 system (see Section 5.1.3) as shown in Figures 5.7 and 5.8.

Our approach is to prove that, in the stable failures model $\mathcal{F}$, the EP2 system is a refinement of the sequential system shown in Figure 7.5. Here, we have an `Abstract` process that sends a `SessionStart` value and then enters a loop. The `Loop` process either sends a `SessionEnd` message and terminates, or it sends a certain type of *request* message followed by a *response* message (of the type corresponding to the to the type of the *request* message) and then repeats the loop. The process `Loop` chooses internally, which of these five branches is taken. As this system has no parallelism it is impossible for it to deadlock. Process refinement within stable

```
ccspec GetInitialisationData =

  data D_ACL_GetInitialisation

  channels
    C_SI_Init: D_SI_Init

  process

 let
    Abstract =
       C_SI_Init !? sessionStart: D_SI_Init_SessionStart -> Loop

    Loop =   C_SI_Init ! seM
             -> SKIP
          |~| C_SI_Init ! cdrM
             -> C_SI_Init !? response: D_SI_Init_ConfigDataResponse
             -> Loop
          |~| C_SI_Init ! cdnM
             -> C_SI_Init !? acknowledge:
                D_SI_Init_ConfigDataAcknowledge
             -> Loop
          |~| C_SI_Init ! rcdnM
             -> C_SI_Init !? acknowledge:
                D_SI_Init_RemoveConfigDataAcknowledge
             -> Loop
          |~| C_SI_Init ! acdnM
             -> C_SI_Init !? acknowledge:
                D_SI_Init_ActivateConfigDataAcknowledge
             -> Loop
  in
    Abstract
end
```

Figure 7.5: A CSP-CASL specification of the EP2 dialog between the terminal and the acquirer as a sequential system.

```
1  theorem ep2: "Abs_System =F System"
2  apply (unfold System_def Abs_System_def)
3  apply (rule cspF_fp_induct_left[of _ "Abs_System_to_System"])
4  apply (simp_all)
5  apply (induct_tac p)
6
7  (* main part *)
8
9  apply (tactic {* cspF_hsf_tac 1 *} | rule cspF_decompo |
10        auto simp add: csp_prefix_ss_def image_iff inj_on_def)+
11 done
```

Figure 7.6: Proof of deadlock freedom of the EP2 system (see Figures 5.7 and 5.8).

failures model preserves deadlock freedom. Hence if we can show that the EP2 system is indeed a refinement of the sequential system, then the EP2 system is guaranteed to be deadlock free.

For our refinement proof we apply the algorithms discussed in this paper on both the EP2 system as well as on the sequential system specification (they both have the same data-part).

In the modelling of the EP2 dialog we ensured that certain message groups do not overlap, i.e., messages of type `SessionEnd`, `ConfigDataRequest`, `D_SI_Init_ConfigDataNotification`, `D_SI_Init_RemoveConfigDataNotification`, and `D_SI_Init_ActivateConfigDataAcknowledge` are never equal to each other. This was done using axioms in lines 15 to 37 of Figure 5.7).

We have to prove theorems that lift the axioms to undefined elements – this is an indication of the type of *Integration Theorems* that are required for syntactical proofs. Once we have proven these theorems and added them to Isabelle/HOL's simplifier set, we can then prove deadlock freedom as shown in Figure 7.6 (we actually show more, namely that both systems are equivalent). This refinement proof involves recursive process definitions. These are first unfolded, then (metric) fixed point induction is applied. A powerful tactic from CSP-Prover finally discharges the proof obligation. The whole proof script involves syntactic proof techniques only.

7.3 Statistics

	Example 1	Example 2	Example 3	Example 4	EP2
CSP-CASL Construction	1	1	1	2	95
Data Theorems & Process Refinements	1	1	1	1	20
Sum	2	2	2	3	115

Figure 7.7: Table showing the running time (in seconds) of Isabelle/HOL code.

The algorithm has been carried out manually on the all four core examples and the EP2 specification. Figure 7.7 shows the running time of the theory files that have been produced by CSP-CASL prover for each of the proof discussed in this section[1]. All running times are reasonably short. The EP2 example involves an unusually large number of sorts and sub-sort relations. This explains the relatively long execution time for the proofs in the alphabet construction. This is still a good runtime in relation to the proof time of the process part.

[1]The tests were carried out on a computer with a 1.5GHz computer with 512Mb.

Chapter 8

Conclusions and future work

Contents

8.1 Summary

CSP-CASL Semantics	Our Algorithm
sub-sorted signature to many-sorted signature using `inj`, `pr` and $\in^S_{S'}$	HETS translation *SubPCFOL*$^=$ to *PCFOL*$^=$
alphabet construction: $A(M) := (\biguplus_{s \in S} M_s)_{/\sim}$	types Alphabet = "PreAlphabet quot"
definition of $\sim$	definition of `eq` and `compare_to` functions.
$\sim$ is an equivalence relation	proof obligations from the instantiation `instance PreAlphabet::equiv`

Figure 8.1: Table showing the matching of our algorithm to the CSP-CASL semantics.

In this thesis we have explored essential theorem proving methods and techniques for CSP-CASL. This has lead to the development of an architecture for CSP-CASL-Prover which re-uses the tools HETS and CSP-Prover. We designed up to the algorithmic level procedures for theorem proving support on CSP-CASL specifications. This construction faithfully respects the semantics of CSP-CASL as shown in Figure 8.1.

We have applied our approach to a case study of industrial strength by carrying out our algorithm manually. This has shown several key facts. Firstly, that our concept works out. Secondly, that it is

possible to perform theorem proving on CSP-CASL specifications with current technology. Thirdly, that our approach scales up to large real-world distributed systems.

We especially achieved:

- The derived specifications are readable and manageable for human beings. They are of a reasonable size in terms of the number of lines.
- When using CSP-CASL-Prover, reasoning about CSP-CASL specifications becomes as easy as reasoning about data and processes separately.
- The derived theory files are practical and the runtime of them in Isabelle/HOL is reasonable.

The results presented within this thesis have already been published in [OIR07, OIR08].

8.2 Future work

There are several aspects of work that should be undertaken to follow on from this project. Firstly, the algorithms described in Chapter 6 should be implemented in Haskell. This will also lead to further development of the proof infrastructure offered by CSP-CASL-Prover. Secondly, a semi-deep encoding should be investigated. Thirdly, the potential of CSP-CASL-Prover should be demonstrated, e.g., by undertaking a larger case study in the contex of the *Grand Challenge 6: Dependable Systems Evolution* initiative.

8.2.1 Implementation and further development of CSP-CASL-Prover

The algorithms described in Chapter 6 should be implemented within the HETS framework. As HETS is implemented in the functional programming language Haskell, the algorithms will also need to be implemented in Haskell. HETS already contains a parser and a static analyser for CSP-CASL [Gim07]. Built in to the static analysis for CSP-CASL is a check that rejects all CSP-CASL specifications which do not have *local top elements*. The algorithms for CSP-CASL-Prover require specification to have *local top elements*.

HETS is currently in a state where it is appropriate for the algorithms to be implemented. The implementation will need to call the HETS machinery to parse and check CSP-CASL specifications, once accepted the implementation will need to access certain information in order to provide the algorithms with their required input. The data that needs to be accessed form a parse tree for a given CSP-CASL specification is:

- What sorts symbols are there?
- What does the sub-sort graph look like?
- What are all the constants, functions and predicate symbols and their profiles?

With these questions answered, the implementation has all the information required to execute the algorithms for CSP-CASL-Prover.

8.2.2 Semi-deep encoding

As seen in the thesis the shallow encoding works out. However, it comes with the "downside" that the following items of the encoding are heavily dependent on the specification:

- The construction of the `eq` function.
- The proof that the function `eq` is symmetric.
- The proof that the `eq` function is transitive.

The theorem and proof that the function `eq` is reflexive is simple enough to be completely independent of the specification. Hence the semantical construction is individual for each specification as the proofs of symmetry and transitivity of the function `eq` are dependent on the specification.

We suggest a semi-deep encoding in contrast to the well studied techniques of shallow and deep encoding [NvOP00]. Such an encoding would consist of:

- The shallow encoding enriched by,
- A sub-sort graph where it is assumed that the graph and the encoding fit together.

The semantical construction can then be built using theorems and proofs which only depend on properties of this graph. Figure 4.3 shows the CASL specification which we have used as a running example in Section 4.2.1 in order to explain the Isabelle encoding of CASL. This specification has two sorts `S` and `T`. Sort `S` is a sub-sort of `T`. There is also a partial function from sort `S` to sort `T`. We will now use this same example in order to illustrate what a semi-deep encoding might look like.

A semi-deep encoding has the same starting point as the shallow encoding, that is we translate the specification into Isabelle as outline in Section 4.2.1. HETS must add to the translated code (Figure 4.6) some addition Isabelle/HOL commands. The following code might be added for the semi-deep encoding:

```
datatype the_sorts = S | T
consts
  subsorts :: "(the_sorts * the_sorts) set"
defs
  subsorts_def : "subsorts == {(S,S),(S,T),(T,T)}"

consts
  injection : "'a => the_sorts => 'a"
defs
  injection_def : "..."

consts
  sort_of :: "'a => the_sorts"
  is_bottom :: "'a => bool"
```

There is a `datatype` command which creates a new type `the_sorts` with two constructors `S` and `T` which represent the names of the sorts in the original specification. We then have a binary relation `subsorts` which in Isabelle/HOL is defined as a set of pairs `(the_sorts *`

the_sorts). The definition of this relation is the set of all pairs of sorts that are in the original sub-sort relation (we may need to take the transitive closure – we assume here that this is necessary). We also need a function that takes an element of an arbitrary type, tells us what sort it is from (this is the function sort_of). Finally we need a function that tests whether an element is a bottom element or not, this is the function is_bottom. We have omitted the definition of the functions sort_of and is_bottom as these cause no trouble and can be easily defined using the primitive recursion scheme.

We now need a function that can inject an arbitrary element of any sort to a super sort. For instance if we have x of type S then we would like the function application

```
injection x T
```

to inject x from type S to type T, the result would be an element of type T. The target type is specified by the second argument which is of type the_sorts. It is future work to develop an appropriate type signature and definition for the injection function in Isabelle/HOL.

Now we can define what it means for a specification to have *local top elements*. This can be done with the following Isabelle/HOL code:

```
consts
  LocalTopElements :: bool
defs
  LocalTopElements_def : "LocalTopElements ==
  forall s u v . ((s,u) : subsorts & (s,v) : subsorts)
  --> (exists t . ((u,t) : subsorts & (v,t) : subsorts))"
```

We have specified a constant LocalTopElements of type bool which is true when the specification has local top elements.

```
consts
  eq :: "'a => 'b => bool"
defs
  eq_def : "eq (x::'a) (y::'b) ==
  (  is_bottom x
   & is_bottom y
   & exists u . ((sort_of x,u) : subsorts &
                 (sort_of y,u) : subsorts)
  ) |
  (  ~ is_bottom x
   & ~ is_bottom y
   & exists u . ((sort_of x,u) : subsorts &
                 (sort_of y,u) : subsorts)
   & forall u . (((sort_of x,u) : subsorts &
                  (sort_of y,u) : subsorts)
                 -->
                 (forall z . (((sort_of x,z) : subsorts &
                              (sort_of y, z) : subsorts)
```

```
                                --> (injection x z = injection y z)
                              )
                  )
                ) "
```

Here we have specified the function `eq` exactly as outlined in Section 5.2. The above sketch illustrates the rough idea of this approach. It is future work to find a type correct and valid encoding.

We can now prove that the function `eq` is reflexive, symmetric and transitive using the assumption that `LocalTopElements=True`. These proofs will now be independent of the specification and only depend on a proof that `LocalTopElements=True`. This is now the only proof which is dependent on the specification. The construction is also independent of the specification except for the definitions of the functions `subsorts`, `injection`, `sort_of`, and `is_bottom`.

As a result of a semi-deep encoding, the proofs that the function `eq` is symmetric and transitive become independent of the specification. CSP-CASL-Prover is now required to produce static proofs of the symmetry and transitivity of the function `eq`, which are always the same for each specification. The old proof of transitivity proved n^3 sub-goals where n is the number of sorts in the specification. The new proof will not split the goal into as many sub-goals and will be static, hence it may be built in to the heap system of Isabelle and be proven once and for all.

There is a trade-off with the new proofs. That is the new proofs rely on the assumption that the sub-sort graph has local top elements. This new extra theorem and proof must be provided by CSP-CASL-Prover. This proof will be easier and shorter than the previous proofs of symmetry and transitivity of the function `eq`. Hence this is a good trade-off and an improvement over the shallow encoding.

This approach will require modification of the HETS tool and hence will require the co-operation of the HETS development team at Bremen University.

8.2.3 Grand Challenge 6: Dependable Systems Evolution Initiative

Following an initiative by T. Hoare, in a combined effort, Computer Scientists in the U.K. have identified several Grand Challenges [Man, Hoa, GCH, HM05] for the field, and have joined their forces to advance the state of Software Engineering and Computer Science. These Grand Challenges have international recognition with special sessions at conferences such as *ETAPS'05* (The European Joint Conferences on Theory and Practice of Software)[1]. Within the Grand Challenge initiative are several Grand Challenges with the goal of advancing different areas of research. Due to the style of each challenge, a solution would indicate a major breakthrough in the area of the challenge.

The *Grand Challenge 6 – Dependable Systems Evolution* (Committee members: Juan Bicarregui, Jonathan Bowen, Tony Hoare, Cliff Jones, John McDermid, Colin O'Halloran, Peter O'Hearn, Brian Randell, Martyn Thomas and Jim Woodcock (chair)) [GC6] intends to give evidence that fully verified software is possible and to advance verification technology. In order to accomplish this they wish to:

[1] http://www.etaps05.inf.ed.ac.uk/

- Create a repository of verified software.
- Identify "large projects" which will advance the field.
- Create tools that help in the development of completely reliable and dependable systems.

The Grand Challenge 6 initiative has been recognised internationally. The following conferences in 2008 had special dedicated sessions for the initiative:

- *ICECCS'08* (International Conference on Engineering of Complex Computer Systems)[2],
- *FM'08* (International Symposium on Formal Methods)[3],
- *SBMF'08* (Brazilian Symposium on Formal Methods)[4],
- *ABZ'08* (Abstract State Machines, B and Z)[5], and
- *VSTTE'08* (Verified Software: Theories, Tools, Experiments)[6].

The Mondex pilot project [Mon] was the first large project launched in January 2006 and was set to have a duration of one year. Mondex is an electronic purse that stores money on smart cards. Mondex enables people to carry, store and spend cash values using a payment card. The money on these smart cards behave exactly like normal cash and can be transferred to other purses without requiring signatures, PIN numbers or authorisation [Mon]. The Mondex pilot project has now come to an end. During the project many interesting properties of the Mondex system were verified using various formal methods and specification languages such as *Z*, *B*, *Event-B* and *RAISE* [JWCW08].

There are new large projects being identified within the Grand Challenge 6 initiative [GC608]. Among these "large projects" we have identified the following two potential applications for analysis and verification of CSP-CASL-Prover:

- The Pacemaker formal methods challenge.
- The FreeRTOS challenge.

8.2.3.1 Pacemaker formal methods challenge

The pacemaker formal methods challenge is an international challenge which is recognised within the Grand Challenge 6 initiative. It is managed by the *Software Quality Research Laboratory* [Pac] which is associated with the Department of Computing and Software at McMaster University, Canada [cas].

A pacemaker is a small medical device that is implanted into patients, an example can be seen in Figure 8.2. It controls abnormal heart rhythms using electrical impulses delivered by electrodes contacting the heart muscles. These electrical impulses regulate the beating of the heart.

[2] http://www.iceccs.org/
[3] http://www.fm2008.abo.fi/
[4] http://www.lasid.ufba.br/sbmf2008/
[5] http://www.cs.york.ac.uk/circus/mc/abz/
[6] http://qpq.csl.sri.com/vsr/vstte-08/

Figure 8.2: Image of a pacemaker [Pac].

The company *Boston Scientific* [Bos] has released to the public domain a system specification of a previous generation pacemaker. The aims of this project are fairly wide. Participants can contribute anything from a complete version of the pacemaker software to just a formal requirements document [Pac].

The heart pacemaker is a state based system. It is a strength of CSP-CASL to describe such state based systems. States correspond to processes, transitions correspond to the action prefix operator, and branching can be captured by choice operators. Parallel composition can also be used for neat and elegant descriptions of such state based systems. Once the pacemaker system has been modelled, one is able to prove interesting properties of the system, such as liveness using the CSP-CASL notions of refinement.

8.2.3.2 FreeRTOS challenge

In co-operation with the company *Wittenstein High Integrity Systems* [Wit] the Grand Challenge 6 group has identified the verification of the operating system *FreeRTOS* as a Grand Challenge.

FreeRTOS is a portable, open source, mini real time kernel designed specifically for small embedded systems which can be used in commercial applications [Fre]. A *real time kernel* (sometimes called a *real time scheduler*) is the part of the kernel that is responsible for deciding which task should be currently executing and switching the execution between such tasks [Fre].

Clearly such a scheduler has concurrent aspects. There are multiple processes (or tasks) which would like to be executing on the processor. Only one task can be executing at any one time and it is the responsibility of the scheduler as to which one has this privilege. Tasks can be in one of the following states:

- Running – When the task is actually executing and utilising the processor.
- Ready – When the task is able to execute but a different task is currently in the *Running* state.
- Blocked – When the task is waiting for some temporal or external event.
- Suspended – When the task has been explicitly suspended.

The transitions from one state to another can be seen in Figure 8.3. Only tasks in the *Ready* state can be moved into the *Running* state and only a single task can be in the *Running* state at any one time. A task may move to the *Suspended* state from any other state by the `vTaskSuspend()` API call being made. A suspended task can only move to a *Ready* state by the `vTaskResume()` API call being made. A running task may move to the *Blocked* state by calling any blocking API function. For example, reading from an external device would cause the task to block. A blocked

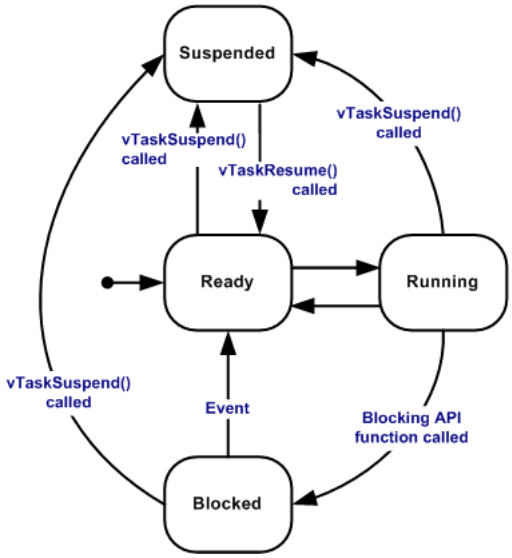

Figure 8.3: Diagram showing the valid task state transitions [Fre].

task may move to the *Ready* state if an event occurs. For example a task that is blocked because it is waiting on an external device to read data would become ready when the device has finished reading the data and has signalled so with an event.

CSP-CASL is well suited to model such Real Time Schedulers because such systems have large amounts of concurrency and are state based. The states correspond to processes and the transitions correspond to action prefix events. There are a number of standard functions that are used in *Real Time Operating Systems* and also studied in the classical *Process Algebra* literature such as:

- Queues,
- Binary semaphores,
- Counting semaphores,
- Mutexes, and
- Recursive mutexes.

All of these are used for inter-task communications, for instance when processes synchronise in order to prevent race conditions from occurring. Once modelled we can use CSP-CASL-Prover to verify interesting properties of such a system. For instance some properties that we may wish to prove are:

- That the scheduler meets its specifications.
- That the scheduler does not deadlock or livelock.
- That binary semaphores do prevent race conditions.

However, in order to properly model such a system, we must abstract from the real time aspect of *FreeRTOS* as CSP-CASL has no real time mechanisms. Creating a work around would also be

possible if we do not wish to abstract away from the real time aspects of such a system. We feel that this aspect does not detract from the neat and elegant modelling that CSP-CASL can provide for such a system.

Bibliography

[Asp00] D. Aspinall. Proof general: A generic tool for proof development. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 38–42, London, UK, 2000. Springer-Verlag.

[BFG+05] B. Badban, W. Fokkink, J. F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[Bir98] R. Bird. *Introduction to Functional Programming*. Pearson Education, 1998.

[BM04] M. Bidoit and P. D. Mosses. CASL *User Manual*. LNCS 2900. Springer, 2004.

[Bos] Homepage of the compnay Boston Scientific. `http://www.bostonscientific.com/`.

[cas] Homepage of the Department of Computing and Software at McMaster University, Canada. `http://www.cas.mcmaster.ca/`.

[ep202] *eft/pos 2000 Specification, version 1.0.1*. EP2 Consortium, 2002.

[Fia04] J. L. Fiadeiro. *Categories for Software Engineering*. Springer, 2004.

[Fre] Webpage on FreeRTOS. `http://www.freertos.org/`.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[GC6] Homepage of the Grand Challenge 6: Dependable systems evolution initiative. `http://www.fmnet.info/gc6/`.

[GC608] Webpage on the grand challenge 6 workshop, March 2008. `http://www.smithinst.ac.uk/Events/Verification08/`.

[GCH] Homepage of the Grand Challenges. `http://www.ukcrc.org.uk/grand_challenges/`.

[Gim07] A. Gimblett. Tool support for CSP-CASL, 2007. MPhil thesis. Submitted.

[GP95] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[GRS05] A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment system in CSP-CASL. In *WADT 2004*, LNCS 3423, pages 61–78. Springer, 2005.

[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.

[HM05] C. A. R. Hoare and R. Milner. Grand challenges for computing research. *Comput. J.*, 48(1):49–52, 2005.

[Hoa] C. A. R. Hoare. Criteria for a Grand Challenge. `http://www.cra.org/Activities/grand.challenges/hoare.pdf` consulted February 2003.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[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] Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In *CONCUR'06*, LNCS 4137, pages 158–172. Springer, 2006.

[IR07] Y. Isobe and M. Roggenbach. User guide CSP-Prover Ver 3.0, 2007.

[IRar] Y. Isobe and M. Roggenbach. Proof Principles of CSP – CSP-Prover in Practice. In H. Haasis, H.-J. Kreowski, and B. Scholz-Reiter, editors, *LDIC 2007*. Springer, to appear.

[JWCW08] C. Jones, J. Woodcock, J. Cooke, and J. Wing. Applicable formal methods. *Formal Aspects of Computing*, 20(1), January 2008.

[Kah07] T. Kahsai. Personal communication, 2007.

[LEW97] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of abstract data types*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[LM01] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report Technical Report UU-CS-2001-35, Universiteit Utrecht, 2001.

[Man] K. Mander. Introduction to the Grand Challenges. `http://www.bcs.org/server.php?show=ConWebDoc.4688`.

[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.

[Mon] Webpage on the Mondex pilot project. `http://www.gc6.clrc.ac.uk/VSR/VSR-Mondex.aspx`.

[Mos02] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[Mos04] P. D. Mosses, editor. CASL *Reference Manual*. LNCS 2960. Springer, 2004.

[Mos06] T. Mossakowski. Hets user guide. Technical report, Department of Computer Science; Universität Bremen; Bibliothekstr. 1, 28359 Bremen; http://www.informatik.uni-bremen.de/, 2006.

[MR07] T. Mossakowski and M. Roggenbach. Structured CSP – A process algebra as an institution. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *WADT 2006*, LNCS 4409, pages 92–110. Springer, 2007.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[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.

[NvOP00] T. Nipkow, D. von Oheimb, and C. Pusch. *micro*Java: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*. IOS Press, 2000.

[OIR07] L. O'Reilly, Y. Isobe, and M. Roggenbach. Integrating Theorem Proving for Processes and Data. In *9th JSSST Workshop on Programming and Programming Languages (PPL2007)*. Japan Society for Software Science and Technology, 2007.

[OIR08] L. O'Reilly, Y. Isobe, and M. Roggenbach. CSP-CASL-Prover – Tool integration and algorithms for automated proof generation. In M. Haveraaen, J. Power, and M. Seisenberger, editors, *CALCO Young Researchers Workshop, CALCO-jnr 2007, selected papers*, UIB Report no 2008-367, pages 17–34. Department of Informatics University of Bergen, Febuary 2008.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE-11*, LNAI 607, pages 748–752. Springer, 1992.

[Pac] Webpage on Pacemaker Formal Methods Challenge. `http://sqrl.mcmaster.ca/pacemaker.htm`.

[Pau94] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

[Pro] Webpage on Proof General. `http://proofgeneral.inf.ed.ac.uk/`.

[Rog06] M. 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] B. Roscoe. Revivals, stuckness and responsiveness, 2005. unpublished draft.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, March 1999.

[Wad93] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

[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.

[Wit] Webpage of Wittenstein High Integrity Systems. `http://openrtos.highintegritysystems.com/`.