A Well-formedness Checker for μCRL

Joris Hillebrand[1] Henri Korver[2*]

[1] *University of Amsterdam, Programming Research Group*
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

[2] *Department of Philosophy, Utrecht University*
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

E-mail: joris@fwi.uva.nl korver@phil.ruu.nl

Abstract

We present an algebraic specification for the static semantics of the process language μCRL. This specification can be executed in the ASF+SDF meta-environment which automatically derives a rewriting system for it. In this way, we immediately obtain a prototype of an enriched type checker (well-formedness checker) for μCRL. Our prototype is rather efficient and shows that automatic prototyping of formal specifications is feasible.

Key Words & Phrases: Formal specification, Algebraic specification, Specification language, μCRL, Static semantics, Type checking, ASF+SDF

*Partly supported by the European Communities under RACE project no. 2076, Broadband Object Oriented Service Technology (BOOST), and the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO).

Contents

1 Introduction

This paper presents the algebraic specification of a well-formedness checker for the language μCRL. We call it a well-formedness checker as it can be considered as an enriched type checker, i.e., it checks the internal consistency of a μCRL specification and some extra properties that are desirable. The definition of μCRL can be found in [GP95].

The language μCRL can be compared with CRL [RBE+93], LOTOS [BB87, Bri87], PSF [MV90, Mau91], which are used for the specification of distributed systems. However, there are two main differences. First of all, μCRL has a considerably smaller syntax (no modularisation facilities etc.); in exchange, it has a sharply defined and well-understood semantic basis. Second, it features a proof theory for proving equality between data and processes.

The specification has been written in the ASF+SDF formalism (see [Kli93]). This approach has three main advantages:

- Our specification is highly readable as ASF+SDF has an algebraic structure and supports mix-fix notation. The constraints as have been posed in [GP95] correspond directly with the ASF+SDF code. In particular, each constraint can be related to an equation in our specification.
- We benefit from the incremental nature of ASF+SDF as μCRL is still under development.
- The specification can be executed such that we automatically obtain a well-formedness checker. In this way, a working tool is generated from the formal description of the static semantics of μCRL. An example can be found in Figure 1, where we defined a constant function `foo` of sort `Bar` without defining the sort `Bar`.

As a final remark, we note that we follow the static semantics described in [GP95] exactly, so we do not comment on design choices regarding syntax or semantics.

The paper is organized as follows. In Section 2, ASF+SDF and its meta-environment are described. In Section 3, the language μCRL is introduced, and we list the properties for which we will check μCRL specifications. Section 4 describes the specification of the well-formedness checker. Finally, related and future work are discussed in Section 5. The entire specification is listed in Appendix A.

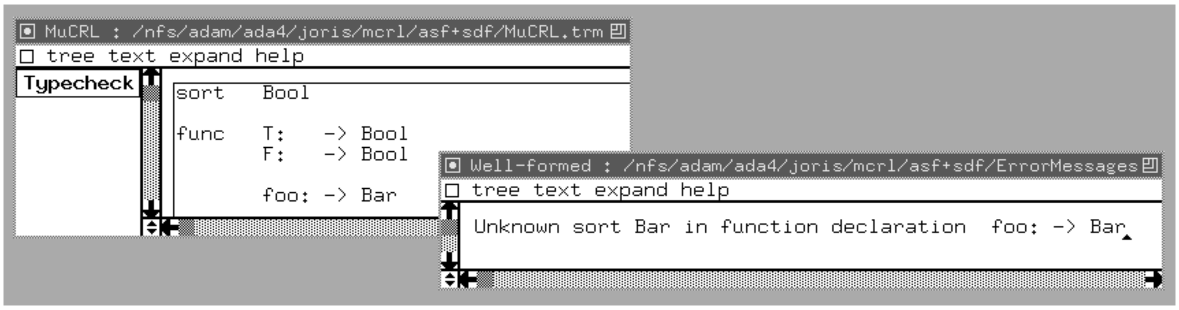

Figure 1: Example of an erroneous specification.

Acknowledgements. The authors would like to thank the following persons for their help on many levels of the realization of this paper. Alban Ponse explained many details of μCRL and commented on earlier versions of this paper. Wilco Koorn suggested to map a μCRL specification onto a list of errors. He also assisted in the use of the ASF+SDF system. Paul Klint and Mark van den Brand made suggestions to keep this paper readable. Mark van den Brand also commented on earlier versions of the paper and the specification. Jan Rekers and Emma van der Meulen helped in the initial start-up of the ASF+SDF system. Eelco Visser has been of great help in the use of the ToLaTeX tool. Without his help, the ASF+SDF code in this paper would have been far less readable.

2 The ASF+SDF Meta-environment

The well-formedness checker described in this paper has been written in the ASF+SDF specification formalism, a combination of ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism).

In ASF [BHK89b], we have modules describing data by means of functions and equations over these functions. Each module has a signature part that defines the abstract syntax of the functions. With these functions, data elements of a certain sort can be constructed. A set of conditional equations provides the functions with semantics. Data elements with the same meaning can be identified using the equations.

In SDF [HHKR89, HK89b], lexical, context-free, as well as abstract syntax can be defined simultaneously. From these definitions, parsers can be generated incrementally. This makes an implicit translation from text strings to abstract syntax trees possible.

The combination of ASF and SDF has resulted in a specification formalism, in which semantics are associated with the tree representation of text strings. This means that the equations are associated with the abstract syntax trees generated from text strings. It allows complete syntactic freedom. An ASF+SDF specification consists of modules having the following sections: an imports section, where other modules can be imported, a section in which the sorts of a signature can be defined, a section for the definition of lexical syntax, one for the definition of context-free syntax, a section for the definition of variables to be used in the equations, and a section for conditional equations.

The ASF+SDF *meta-environment* [Kli93] supports writing specifications in the ASF+SDF formalism. The meta-environment makes it possible to execute ASF+SDF specifications, by generating parsers from these specifications and deriving term rewriting systems for them. Also, prettyprinting and syntax-directed editing of modules and terms are supported. This makes the ASF+SDF meta-environment very suitable for prototyping and testing of specifications.

3 μCRL

In this section, the specification language μCRL will be discussed shortly. μCRL stands for *micro* Common Representation Language. CRL [RBE$^+$93] was designed in the RACE (Research and technology development in Advanced Communications in Europe) project SPECS (Specification and Programming Environment for Communication Software). The purpose was to design a language containing several features of current specification languages. Then each of these languages could be translated into CRL. Tools for these languages would then only have to be written for CRL. See [RBE$^+$93] for more information on SPECS and RACE.

μCRL contains some essential elements of CRL, but has far less language features. It focuses mainly on the specification of data and processes, without modularisation, to make a thorough analysis of processes with data possible. This has resulted in a proof theory, based on natural deduction, by means of which processes and data terms in a μCRL specification can be proven equal. μCRL has been defined in [GP95]. A description of the proof theory can be found in [GP94].

3.1 Syntax

As just stated, μCRL is a specification language for data and processes. The specification of data is done by the declaration of sorts and functions over these sorts. Next, the functions can be given a meaning by specifying a many-sorted term rewriting system. See the following example, where we specify the natural numbers with addition.

sort	Natural			
func	0:		$\longrightarrow$	Natural
	succ:	Natural	$\longrightarrow$	Natural
	add:	Natural # Natural	$\longrightarrow$	Natural

var m,n: Natural
rew add(0,n) = n
add(succ(m),n) = succ(add(m,n))

Processes are described in the style of CCS ([Mil80]), CSP ([Hoa85]), or ACP ([BK84a], [BK84b], for an overview see [BW90]). The syntax of the process expressions has been taken from ACP. Processes are built from atomic actions, using particular operators, e.g., for alternative or sequential composition. As in ACP, it is possible to define communications between atomic actions. In the next example, we specify a two-bit buffer over the naturals.

act read1, read2, send2, comm2, send3: Natural

comm read2 | send2 = comm2

proc OneBitBuff1 = sum(n:Natural, read1(n) . send2(n)) . OneBitBuff1
OneBitBuff2 = sum(n:Natural, read2(n) . send3(n)) . OneBitBuff2

TwoBitBuff = hide({comm2}, encap({read2, send2}, OneBitBuff1 || OneBitBuff2))

For the formal syntax of μCRL we refer to [GP95]. There the syntax is given in BNF. An SDF definition of the syntax can be found in Appendix A.4. Also [GP95] contains an SDF definition, but it differs from the BNF definition when it comes to process expressions. Our SDF definition matches the original BNF definition.

Our SDF definition does not have sort names for list constructs, since lists have been built into the ASF+SDF formalism. For example, a sequence of one or more *Names* is not called a *Space-name-list*, but is written *Name+*. We also use ASF+SDF lists to denote that a μCRL specification consists of one or more smaller specifications. Therefore, we define sort, function, rewrite, action, communication, and process specifications to be *Single-specifications*. Next, we define a *Specification* as a sequence of one or more Single-specifications:

context-free syntax
Sort-specification $\rightarrow$ Single-specification
Function-specification $\rightarrow$ Single-specification
Rewrite-specification $\rightarrow$ Single-specification
Action-specification $\rightarrow$ Single-specification
Communication-specification $\rightarrow$ Single-specification
Process-specification $\rightarrow$ Single-specification

Single-specification+ $\rightarrow$ Specification

3.2 Static semantics

In this section, we give an informal formulation of the constraints we pose on a μCRL specification. In section 4, we will show how we can formalize these constraints.

Sorts

1. All sort names must be different.

Functions

1. No two functions can have the same name and the same input pattern.
2. The input sorts and the output sort of every function must be declared.

Variables

1. All variable names must be different.
2. A variable can not have the same name as a constant function, or as an action or process without arguments.
3. The sort of each variable must be declared.

Rewrite rules

1. The data term at the left-hand side of the equation must be an application of a declared function.
2. The two sides of a rewrite rule should be correctly-typed data terms of the same sort.
3. The variables that appear on the right-hand side of the equation must also appear on the left-hand side of the equation.

Data terms

1. A data term must either be a declared variable, or an application of a declared function.
2. If a data term is a function application, its arguments, if any, must be correctly-typed data terms.

Actions

1. No two actions can have the same name and the same input pattern.
2. An action can not have the same name and the same input pattern as a function.
3. All sorts in an input pattern must be declared.

Communications

1. At most one communication can be defined on each pair of action names.
2. For each action name in a communication declaration an action must be declared.
3. The actions declared for one of the action names on which a communication is defined must have the same input patterns as the actions declared for the other action name on which that communication is defined.
4. The actions declared for the action name that is the result of a communication must have all the input patterns of the actions declared for the action names on which that communication is defined.

Processes
In this paragraph, we say that process declarations have the form *Process header* = *Process body*.

1. No two process declarations can have the same header, i.e., the same name and the same input pattern.

2. A process header can not have the same name and the same input pattern as a function or an action.

3. The constraints on variables as mentioned above apply on the sequence of variable declarations (which can be empty) in a process header. These variables are local to the process declaration.

4. A process body must be a process expression of one of the following forms. Correctness of a process expression is always defined with respect to its local variables. The following priority exists between the process operators: "+" < {"$\|$", "$|$", "$\lfloor\!\lfloor$"} < "$\triangleleft\ \triangleright$" < "$\cdot$".

   (a) $p_1 + p_2$, or
   (b) $p_1 \parallel p_2$, or
   (c) $p_1 \lfloor\!\lfloor p_2$, or
   (d) $p_1 \mid p_2$, or
   (e) $p_1 \cdot p_2$, in which case p_1 and p_2 must be correctly-typed process expressions.
   (f) $p_1 \triangleleft t \triangleright p_2$, in which case t must be a correctly-typed data term of sort "Bool", and p_1 and p_2 must be correctly-typed process expressions.
   (g) δ
   (h) τ
   (i) $\partial(\{n_1, \ldots, n_m\}, p)$, or
   (j) $\tau(\{n_1, \ldots, n_m\}, p)$, in which case all names $n_1, \ldots, n_m$ must be different, for each of these names an action with that name must be declared, and p must be a correctly-typed process expression.
   (k) $\rho(\{n_1 \rightarrow n'_1, \ldots, n_m \rightarrow n'_m\}, p)$, in which case all names $n_1, \ldots, n_m$ must be different, for each name $n_1, \ldots, n_m, n'_1, \ldots, n'_m$ an action must be declared, the input patterns of the actions declared for a name n_k must also be input patterns of the actions declared for the name n'_k, and p must be a correctly-typed process expression.
   (l) $\sum(x : S, p)$, in which case the constraints on variables as mentioned above apply on the variable declaration $x : S$, and p must be a correctly-typed process expression.
   (m) n, in which case an action or process n without input must be declared.
   (n) $n(t_1, \ldots, t_m)$, in which case an action or process n with an input pattern that matches the sorts of $t_1, \ldots, t_m$ must be declared. Moreover, $t_1, \ldots, t_m$ must be correctly-typed data terms.
   (o) (p), in which case p must be a correctly-typed process expression.

3.3 Well-formedness

In the previous section, we defined under which conditions a μCRL specification is correct, i.e., when it is internally consistent.

There are, however, some extra properties one would like to impose on a specification. For instance, to give the if-then-else process construction a meaning, one would like to have exactly two boolean constants: true ("T") and false ("F"). Unfortunately, it is undecidable whether or not true can be rewritten into false (or vice versa) in a specification, or whether a specification has any other boolean normal forms. Therefore, the best we can do is to make sure that the sort "Bool" with constants "T" and "F" has been declared. Furthermore, we want the communication function not only to be commutative, but also associative, since this is the case in ACP. Finally, we do not want empty sorts, because this would make it very hard to provide a specification with semantics.

A μCRL specification fulfilling the following requirements is called *well-formed*:

1. The specification is statically semantically correct as described in the previous section.
2. The communication function is associative.
3. The specification has no empty sorts. This means that for each sort there exists a closed data term.
4. The sort "Bool" is declared with constant functions "T" and "F".

4 A well-formedness checker

In this section, we specify a function that checks whether or not a μCRL specification is well-formed. The function returns a list of errors when the specification is not well-formed. Since the first demand for well-formedness is that a specification is statically semantically correct, we start to specify this property.

The following variables are used throughout the paper:

variables

E [*0-9'*]∗	→ Specification
SE [*0-9'*]∗	→ Single-specification
SEs [*0-9'*]∗	→ Single-specification+
Sig [*0-9'*]∗	→ Signature
Names [*0-9'*]∗	→ Names
Fun [*0-9'*]∗	→ Fun
Act [*0-9'*]∗	→ Act
Comm [*0-9'*]∗	→ Comm
Proc [*0-9'*]∗	→ Proc
Var [*0-9'*]∗	→ Var
Err [*0-9'*]∗	→ Error
Errs [*0-9'*]∗	→ {Error ";"}+
OErrs [*0-9'*]∗	→ {Error ";"}∗
N [*0-9'*]∗	→ Name
Ns [*0-9'*]∗	→ Name+
ONs [*0-9'*]∗	→ Name∗
Xns [*0-9'*]∗	→ {Name "#"}+
OXns [*0-9'*]∗	→ {Name "#"}∗
Var-sec [*0-9'*]∗	→ Variable-declaration-section
Types [*0-9'*]∗	→ Types

variables

Fd [*0-9'*]∗	→ Function-declaration
Fds [*0-9'*]∗	→ Function-declaration+
OFds [*0-9'*]∗	→ Function-declaration∗
Ads [*0-9'*]∗	→ Action-declaration+
OAds [*0-9'*]∗	→ Action-declaration∗
Cds [*0-9'*]∗	→ Communication-declaration+
OCds [*0-9'*]∗	→ Communication-declaration∗
Pds [*0-9'*]∗	→ Process-declaration+
OPds [*0-9'*]∗	→ Process-declaration∗
Pre [*0-9'*]∗	→ Process-expression
De [*0-9'*]∗	→ Dot-expression
Rw [*0-9'*]∗	→ Rewrite-rule
Rws [*0-9'*]∗	→ Rewrite-rule+
Trm [*0-9'*]∗	→ Data-term
Trms [*0-9'*]∗	→ {Data-term ","}+
OVds [*0-9'*]∗	→ Variable-declaration∗
Svds [*0-9'*]∗	→ {Single-variable-declaration ","}+

We already mentioned that a μCRL specification consists of the declaration of sorts, functions over these sorts, variables, rewrite rules, actions, and communications and processes over these actions.

The language μCRL has a few properties worth mentioning. First of all, it is not necessary that, for instance, all actions are declared in one action specification. A μCRL specification can contain several sort, function, variable, rewrite, action, communication, and process specifications. Also, μCRL has "application-before-definition", which means that, for instance, one can declare a function of which the output sort is declared after the function definition. Therefore, we first collect all sort, function, action, communication, and process declarations from a given specification. The five-tuple thus obtained is called the *signature* of the specification. Next, we examine whether the specification is statically semantically correct with respect to its signature.

The signature of the entire specification will be an argument of every higher-level check function, what makes it a kind of global variable or type environment. This approach allows us to check small pieces of the specification, while having access to relevant information about other parts of the specification.

4.1 Signature

As stated above, the signature of a μCRL specification is a five-tuple, of which each component is a sequence of respectively sorts, functions, actions, communications, and processes declared.

In order to be able to define a signature, we first define these sequences.

context-free syntax
"[" Name∗ "]" → Names
"[" Function-declaration∗ "]" → Fun
"[" Action-declaration∗ "]" → Act
"[" Communication-declaration∗ "]" → Comm
"[" Process-declaration∗ "]" → Proc

As one can see, a sequence is just zero or more elements between square brackets. Since a sort is just a name, the sequence of sorts is a sequence of names, called Names. Furthermore, we define the concatenation of two sequences, and a function that checks whether a specific element is present in a sequence:

context-free syntax
Act "+" Act → Act {**right**}
Process-declaration in Proc → BOOL

Naturally, the same functions have been defined for the other sequences as well.

Now we can define a signature as follows:

context-free syntax
"(" Names "," Fun "," Act "," Comm "," Proc ")" → Signature

We add a function sig that gives the signature of a specification, and functions to extract the different components from a signature.

context-free syntax
sig(Specification) → Signature

sigsort(Signature) → Names
sigfun(Signature) → Fun
sigact(Signature) → Act
sigcomm(Signature) → Comm

sigproc(Signature) $\rightarrow$ Proc

These functions are defined by conditional equations. Below, we give a few examples. The complete specification can be found in appendix A.

$$\mathsf{sig}(\mathsf{func}\ \mathit{Fds}) = ([], [\mathit{Fds}], [], [], []) \qquad \text{[sig-2]}$$

$$\frac{\mathsf{sig}(\mathit{SE}) = (\mathit{Names}_1, \mathit{Fun}_1, \mathit{Act}_1, \mathit{Comm}_1, \mathit{Proc}_1),\quad \mathsf{sig}(\mathit{SEs}) = (\mathit{Names}_2, \mathit{Fun}_2, \mathit{Act}_2, \mathit{Comm}_2, \mathit{Proc}_2)}{\begin{aligned}\mathsf{sig}(\mathit{SE}\ \mathit{SEs}) = (&\mathit{Names}_1 + \mathit{Names}_2,\\ &\mathit{Fun}_1 + \mathit{Fun}_2,\\ &\mathit{Act}_1 + \mathit{Act}_2,\\ &\mathit{Comm}_1 + \mathit{Comm}_2,\\ &\mathit{Proc}_1 + \mathit{Proc}_2)\end{aligned}} \qquad \text{[sig-7]}$$

$$\mathsf{sigsort}((\mathit{Names}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})) = \mathit{Names} \qquad \text{[sigsort-1]}$$

4.2 Errors

Our top-level type-check function is defined as follows:

context-free syntax
ssc(Specification) $\rightarrow$ Errors

It maps a given specification onto a list of errors. Therefore, we have defined an error message for every constraint in Sections 3.2 and 3.3. Such an error message is generated if the constraint behind it is not fulfilled. The error messages are mostly mix-fix functions that allow pieces of the specification to be given as arguments. In this way, the context of an error can be given together with the error message. For example, the following error message is generated when some sort has been multiply declared:

context-free syntax
"Sort" Name "has" "been" "defined" "more" "than" "once" $\rightarrow$ Error

Here, the Name argument is used for the sort name concerned. The other words in the error message have been quoted separately for ease of use. For instance, it is now allowed to split the message in two lines.

The function join concatenates two lists of errors. The message No errors were found is a special constant. It is returned by the type-check function if a specification (or a piece of it) is error-free. Since the entire specification is divided into smaller pieces for type checking, this message is generated for every piece that is error-free. Therefore, multiple appearances of the same error message have to be removed. Also, the special constant No errors were found has to be removed as soon as any other (real) errors are found. This is expressed by the following equations:

$$\mathsf{join}(\mathit{Errs}_1, \mathit{Errs}_2) = \mathit{Errs}_1; \mathit{Errs}_2 \qquad \text{[join-1]}$$

$$\mathit{OErrs}_1; \mathit{Err}; \mathit{OErrs}_2; \mathit{Err}; \mathit{OErrs}_3 = \mathit{OErrs}_1; \mathit{Err}; \mathit{OErrs}_2; \mathit{OErrs}_3 \qquad \text{[errs-1]}$$

$$\mathit{OErrs}; \text{No errors were found}; \mathit{Errs} = \mathit{OErrs}; \mathit{Errs} \qquad \text{[errs-2]}$$

$$\mathit{Errs}; \text{No errors were found}; \mathit{OErrs} = \mathit{Errs}; \mathit{OErrs} \qquad \text{[errs-3]}$$

4.3 Static semantics

As stated earlier, we will check a specification piece by piece, while we use the signature of the entire specification as a sort of global type environment. This is expressed by the top equation for the type-check function ssc:

$$\mathsf{ssc}(E) \;=\; \mathsf{ssc\text{-}spec}(\mathsf{sig}(E),\, E) \qquad \text{[ssc-1]}$$

Next, we split the specification into single specifications:

$$\begin{array}{l}
\mathsf{ssc\text{-}spec}(Sig,\, SE\ SEs) = \qquad\qquad \text{[ssc-spec-7]}\\
\ \mathsf{join}(\mathsf{no\text{-}sort\text{-}overlap}(\mathsf{sigsort}(\mathsf{sig}(SE)),\, \mathsf{sigsort}(\mathsf{sig}(SEs))),\\
\quad\ \mathsf{join}(\mathsf{no\text{-}fun\text{-}overlap}(\mathsf{sigfun}(\mathsf{sig}(SE)),\, \mathsf{sigfun}(\mathsf{sig}(SEs))),\\
\qquad\ \mathsf{join}(\mathsf{no\text{-}act\text{-}overlap}(\mathsf{sigact}(\mathsf{sig}(SE)),\, \mathsf{sigact}(\mathsf{sig}(SEs))),\\
\qquad\quad\ \mathsf{join}(\mathsf{no\text{-}comm\text{-}overlap}(\mathsf{sigcomm}(\mathsf{sig}(SE)),\, \mathsf{sigcomm}(\mathsf{sig}(SEs))),\\
\qquad\qquad\ \mathsf{join}(\mathsf{no\text{-}proc\text{-}overlap}(\mathsf{sigproc}(\mathsf{sig}(SE)),\, \mathsf{sigproc}(\mathsf{sig}(SEs))),\\
\qquad\qquad\quad\ \mathsf{join}(\mathsf{ssc\text{-}spec}(Sig,\, SE),\\
\qquad\qquad\qquad\ \mathsf{ssc\text{-}spec}(Sig,\, SEs)))))))
\end{array}$$

This equation shows that a sequence of two or more single specifications is correct if each of the single specifications is correct. Moreover, if a specification consists of two or more single specifications, we must make sure that each sort, function, action, communication, or process declaration appears in only one single specification. Although each single specification will later be checked for double declarations separately, it is still possible that the same declaration appears in two single specifications.

Now, each single specification can be checked separately, with respect to the signature of the entire specification:

$$\begin{array}{ll}
\mathsf{ssc\text{-}spec}(Sig,\, \mathsf{sort}\ Ns) \;=\; \ldots & \text{[ssc-spec-1]}\\
\mathsf{ssc\text{-}spec}(Sig,\, \mathsf{func}\ Fds) \;=\; \ldots & \text{[ssc-spec-2]}\\
\mathsf{ssc\text{-}spec}(Sig,\, Var\text{-}sec\ \mathsf{rew}\ Rws) \;=\; \ldots & \text{[ssc-spec-3]}\\
\mathsf{ssc\text{-}spec}(Sig,\, \mathsf{act}\ Ads) \;=\; \ldots & \text{[ssc-spec-4]}\\
\mathsf{ssc\text{-}spec}(Sig,\, \mathsf{comm}\ Cds) \;=\; \ldots & \text{[ssc-spec-5]}\\
\mathsf{ssc\text{-}spec}(Sig,\, \mathsf{proc}\ Pds) \;=\; \ldots & \text{[ssc-spec-6]}
\end{array}$$

We have divided the constraints formulated in Sections 3.2 and 3.3 in a number of categories. We shall now show how we have specified the necessary check functions for these categories.

4.3.1 Identification

The first property to check a specification for is whether it is possible to uniquely identify functions, actions, processes, and variables. For instance, an application of a function with a certain name and a certain input pattern can not be identified if two or more functions with that name and that input pattern have been declared.

First of all, each sort can only be declared once:

$$\mathsf{ssc\text{-}spec}(Sig,\, \mathsf{sort}\ Ns) \;=\; \mathsf{uniq\text{-}sorts}([Ns]) \qquad \text{[ssc-spec-1]}$$

$$\mathsf{uniq\text{-}sorts}([\,]) \;=\; \mathsf{No\ errors\ were\ found} \qquad \text{[uniq-sorts-1]}$$

$$\frac{N\ \mathsf{in}\ [ONs] \neq \mathsf{true}}{\mathsf{uniq\text{-}sorts}([N\ ONs]) = \mathsf{uniq\text{-}sorts}([ONs])} \qquad \text{[uniq-sorts-2]}$$

$$\frac{N \text{ in } [ONs] = \text{true}}{\text{uniq-sorts}([N\ ONs]) = \text{join}(\text{Sort } N \text{ has been defined more than once}, \text{uniq-sorts}([ONs]))} \qquad \text{[uniq-sorts-3]}$$

For each sort in the sequence of declared sorts we look whether it is also present in the rest of the sequence. If this is the case, that sort was declared more than once.

We can not handle the function declarations in exact the same way, because it is not enough to check that all function declarations are pairwise different. Two functions having the same name and the same input pattern, but different output sorts, are also not allowed. Therefore, we will use the auxiliary function is-function. This function takes the name and arguments, if any, of a function declaration, and checks whether or not another function with that name and those arguments has been declared.

$$\frac{\text{is-function}([OFds], N) \neq \text{true}}{\text{uniq-fun}([N : \ \rightarrow N'\ OFds]) = \text{uniq-fun}([OFds])} \qquad \text{[uniq-fun-2]}$$

or, in case of a function with arguments:

$$\frac{\text{is-function}([OFds], N : Xns) \neq \text{true}}{\text{uniq-fun}([N : Xns \rightarrow N'\ OFds]) = \text{uniq-fun}([OFds])} \qquad \text{[uniq-fun-4]}$$

with is-function defined as follows:

$$\text{is-function}([OFds_1\ N : \ \rightarrow N'\ OFds_2], N) = \text{true} \qquad \text{[is-function-noargs-1]}$$
$$\text{is-function}([OFds_1\ N : Xns \rightarrow N'\ OFds_2], N : Xns) = \text{true} \qquad \text{[is-function-args-1]}$$

We explicitly keep different equations for functions with and without arguments, because is-function will also be called with action declarations later.

Processes are dealt with in the same way:

$$\frac{\text{var-sorts}([Svds]) = Xns}{\text{is-process}([OPds_1\ N(Svds) = Pre\ OPds_2], N(Xns)) = \text{true}} \qquad \text{[is-process-args-1]}$$

$$\frac{\text{is-process}([OPds], N(\text{var-sorts}([Svds]))) \neq \text{true}}{\text{uniq-proc}([N(Svds) = Pre\ OPds]) = \text{uniq-proc}([OPds])} \qquad \text{[uniq-proc-4]}$$

Because we do not care about the *names* of the variables in the process header, we use the auxiliary function var-sorts to remove these names. The result is returned as an *X-name-list*: a list of names, separated by "#". We will use this construction frequently throughout this paper. It is a very usable format, since it is the format for the input patterns in function and action declarations. Whenever in the sequel we want to have information about the sorts of a list of data terms or a list of variable declarations, this information is returned as an X-name-list. Now matching function, action, or process declarations can be found quickly.

$$\text{var-sorts}([N : N']) = N' \qquad \text{[var-sorts-1]}$$

$$\frac{Xns = \text{var-sorts}([Svds])}{\text{var-sorts}([N : N', Svds]) = N' \#\ Xns} \qquad \text{[var-sorts-2]}$$

With communications, we have to take care of the fact that the communication function is commutative:

$$\text{is-comm}([\mathit{OCds}_1\ N_1 \mid N_2 = N_3\ \mathit{OCds}_2],\ N_1 \mid N_2) \ = \ \text{true} \qquad \text{[is-comm-1]}$$
$$\text{is-comm}([\mathit{OCds}_1\ N_2 \mid N_1 = N_3\ \mathit{OCds}_2],\ N_1 \mid N_2) \ = \ \text{true} \qquad \text{[is-comm-2]}$$

Just as we do not want two functions with the same name and the same input pattern, we do not want an action to have the same name and the same input pattern as a function. In that case we would not be able to distinct that function from that action.

Therefore, one of the checks for actions is done by calling is-function:

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N) \neq \text{true}}{\text{check-ad}(\mathit{Sig},\ N) = \text{No errors were found}} \qquad \text{[check-ad-1]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N : \mathit{Xns}) = \text{true}}{\text{check-ad}(\mathit{Sig},\ N : \mathit{Xns}) = N(\mathit{Xns}) \text{ has been defined as both a function and an action}} \qquad \text{[check-ad-4]}$$

Here, N is the action declaration concerned, with or without input (Xns). Likewise, we do not want a process to have the same name and input pattern as a function or an action:

$$\frac{\begin{array}{r}\mathit{Xns} = \text{var-sorts}([\mathit{Svds}]),\\ \text{is-function}(\text{sigfun}(\mathit{Sig}),\ N : \mathit{Xns}) = \text{true}\end{array}}{\begin{array}{l}\text{check-pd}(\mathit{Sig},\ N(\mathit{Svds}) = \mathit{Pre}) =\\ N(\mathit{Xns}) \text{ has been defined as both a function and a process}\end{array}} \qquad \text{[check-pd-5]}$$

$$\frac{\text{is-action}(\text{sigact}(\mathit{Sig}),\ N) = \text{true}}{\text{check-pd}(\mathit{Sig},\ N = \mathit{Pre}) = N \text{ has been defined as both an action and a process}} \qquad \text{[check-pd-3]}$$

Here, $N = \mathit{Pre}$ (or $N(\mathit{Svds}) = \mathit{Pre}$, in the case of a process with input) is the process declaration concerned. Finally, we do not want a variable to have the same name as a function, action, or process without input:

$$\frac{\begin{array}{r}N' \text{ in sigsort}(\mathit{Sig}) = \text{true},\\ \text{is-function}(\text{sigfun}(\mathit{Sig}),\ N) \neq \text{true},\\ \text{is-action}(\text{sigact}(\mathit{Sig}),\ N) \neq \text{true},\\ \text{is-process}(\text{sigproc}(\mathit{Sig}),\ N) \neq \text{true}\end{array}}{\text{check-vd}(\mathit{Sig},\ N : N') = \text{No errors were found}} \qquad \text{[check-vd-1]}$$

The first condition of this equation checks whether the output sort of the variable has been declared, which brings us to the next paragraph.

4.3.2 Declaration of input and output sorts

Functions have an output sort and can have input sorts. Actions and processes can have data arguments. In all these cases the arguments are data terms of a certain sort. These sorts are specified in the declaration of functions, actions, and processes. We have to check whether these sorts are declared sorts, which should be the case. For processes, this is done by checking the variable declarations in the header. One of the constraints for variables is that their sort has been declared (see also equation [check-vd-1] above):

$$\frac{N' \text{ in sigsort}(\mathit{Sig}) \neq \text{true}}{\text{check-vd}(\mathit{Sig},\ N : N') = \text{Unknown sort } N' \text{ of variable } N} \qquad \text{[check-vd-2]}$$

For function and action declarations, we have functions that check whether the associated sorts have been declared. For functions, this is the function sorts-defined-in-fun:

$$\text{sorts-defined-in-fun}(\mathit{Names},\ [\mathit{OXns}\ \#\ N'],\ N : \mathit{OXns} \rightarrow N')$$

Here, *Names* is a sequence containing all the sorts declared in the specification (the first element of the signature-tuple). The last argument is the function declaration, of which we want to check whether its input and output sorts have been declared. Its input sorts (*OXns*) and its output sort (N') have been collected in the second argument. Note that the third argument is only used in the error message.

Now sorts-defined-in-fun checks for each sort name in the second argument whether or not it is contained in the sequence of all declared sorts in the first argument:

$$\text{sorts-defined-in-fun}(\mathit{Names},\ [],\ \mathit{Fd}) = \text{No errors were found} \qquad \text{[sorts-defined-in-fun-1]}$$

$$\frac{N \text{ in } \mathit{Names} = \text{true}}{\begin{array}{l}\text{sorts-defined-in-fun}(\mathit{Names},\ [N\#\ \mathit{OXns}],\ \mathit{Fd}) = \\ \text{sorts-defined-in-fun}(\mathit{Names},\ [\mathit{OXns}],\ \mathit{Fd})\end{array}} \qquad \text{[sorts-defined-in-fun-2]}$$

$$\frac{N \text{ in } \mathit{Names} \neq \text{true}}{\begin{array}{l}\text{sorts-defined-in-fun}(\mathit{Names},\ [N\#\ \mathit{OXns}],\ \mathit{Fd}) = \\ \text{join(Unknown sort } N \text{ in function declaration } \mathit{Fd}, \\ \quad \text{sorts-defined-in-fun}(\mathit{Names},\ [\mathit{OXns}],\ \mathit{Fd}))\end{array}} \qquad \text{[sorts-defined-in-fun-3]}$$

For action declarations we have the function sorts-defined-in-act:

$$\text{sorts-defined-in-act}(\text{sigsort}(\mathit{Sig}),\ [\mathit{Xns}],\ N : \mathit{Xns})$$

with the action declaration to be investigated in the third argument, and its input sorts in the second argument. The function sorts-defined-in-act has exactly the same behaviour as sorts-defined-in-fun. In fact, if we had left out the third argument, we would not have needed two functions. The reason to use two separate functions (with three arguments) all the same is to be able to generate more specific error messages.

4.3.3 Declaration of applied items

Functions and variables that are used (called) in rewrite rules or process bodies must have been declared. The same holds for actions and processes called in a process body. Since variables can be used in rewrite rules as well as in process bodies, we define correctness of rewrite rules and process bodies not only with respect to the signature, but with respect to the signature and a set of variables. For rewrite rules, this set of variables (*Var*) is formed by the variables declared in a rewrite specification. In the case of a process body, *Var* is formed by the variables declared in the process header.

To start with, the left-hand side of a rewrite rule must be an application of a declared function:

$$\frac{\begin{array}{r}\mathit{Xns} = \text{sort-list}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trms}), \\ \text{is-function}(\text{sigfun}(\mathit{Sig}),\ N : \mathit{Xns}) \neq \text{true}\end{array}}{\begin{array}{l}\text{check-rw}(\mathit{Sig},\ \mathit{Var},\ N(\mathit{Trms}) = \mathit{Trm}) = \\ \text{Unknown function } N(\mathit{Xns}) \text{ on left-hand-side of rewrite-rule}\end{array}} \qquad \text{[check-rw-5]}$$

Here, sort-list calculates the sort of each data term in a list of data terms and returns these sorts as an X-name-list.

Likewise, a data term must either be a declared variable, or an application of a declared function:

$$\frac{\textsf{is-variable}(\textit{Var}, N) = \textsf{true}}{\textsf{ssc-t}(\textit{Sig}, \textit{Var}, N) = \textsf{No errors were found}} \qquad \text{[ssc-t-1]}$$

$$\frac{\begin{array}{r}\textit{Xns} = \textsf{sort-list}(\textit{Sig}, \textit{Var}, \textit{Trms}),\\ \textsf{is-function}(\textsf{sigfun}(\textit{Sig}), N : \textit{Xns}) \neq \textsf{true}\end{array}}{\textsf{ssc-t}(\textit{Sig}, \textit{Var}, N(\textit{Trms})) = \textsf{Unknown function } N(\textit{Xns}) \textsf{ in data-term}} \qquad \text{[ssc-t-5]}$$

Actions and processes called in process bodies must have been declared as well:

$$\frac{\textsf{is-process}(\textsf{sigproc}(\textit{Sig}), N) = \textsf{true}}{\textsf{ssc-be}(\textit{Sig}, \textit{Var}, N) = \textsf{No errors were found}} \qquad \text{[ssc-be-7]}$$

$$\frac{\textsf{is-action}(\textsf{sigact}(\textit{Sig}), N) = \textsf{true}}{\textsf{ssc-be}(\textit{Sig}, \textit{Var}, N) = \textsf{No errors were found}} \qquad \text{[ssc-be-8]}$$

$$\frac{\begin{array}{r}\textit{Xns} = \textsf{sort-list}(\textit{Sig}, \textit{Var}, \textit{Trms}),\\ \textsf{is-process}(\textsf{sigproc}(\textit{Sig}), N(\textit{Xns})) \neq \textsf{true},\\ \textsf{is-action}(\textsf{sigact}(\textit{Sig}), N : \textit{Xns}) \neq \textsf{true}\end{array}}{\textsf{ssc-be}(\textit{Sig}, \textit{Var}, N(\textit{Trms})) = \textsf{Unknown process or action } N(\textit{Xns}) \textsf{ in process expression}} \qquad \text{[ssc-be-12]}$$

Here, N is the function, variable, action, or process called, with or without arguments ($\textit{Trms}$).

4.3.4 Communications

An interesting aspect of μCRL is the way communication works. Communication takes place between actions. However, communications are defined on action *names*. If one defines communication between two action names, all actions defined with these names can communicate. An example:

act	a, b, c	
	a, b, c:	Natural
	c:	Natural # Natural
comm	a | b = c	

In this example, two pairs of actions can communicate: "a" can communicate with "b" into "c" *and* "a: Natural" can communicate with "b: Natural" into "c: Natural".

Now one of the demands that will be checked for is that actions with the same input patterns have been defined for the action names in the communication declaration. In the above example these input patterns are the empty pattern and "Natural". The actions resulting from the communication should also have been defined with these input patterns, although these actions can be defined with more input patterns. In the above example, "c" has also been defined with the input pattern "Natural # Natural".

Because almost all the constraints on a communication declaration are defined in terms of the input patterns of action names, we will need the auxiliary function types. It returns a list of all the input patterns with which actions with a certain name have been declared. In the following equations, the input patterns of actions with name N are collected:

$$\textsf{types}([], N) = [] \qquad \text{[types-1]}$$
$$\textsf{types}([N\ \textit{OAds}], N) = [\textsf{empty-pattern}] + \textsf{types}([\textit{OAds}], N) \qquad \text{[types-2]}$$
$$\textsf{types}([N : \textit{Xns}\ \textit{OAds}], N) = [\textit{Xns}] + \textsf{types}([\textit{OAds}], N) \qquad \text{[types-3]}$$

$$\frac{N' \neq N}{\mathsf{types}([N'\ OAds],\ N) = \mathsf{types}([OAds],\ N)} \qquad \text{[types-4]}$$

$$\frac{N' \neq N}{\mathsf{types}([N' : Xns\ OAds],\ N) = \mathsf{types}([OAds],\ N)} \qquad \text{[types-5]}$$

As one can see, types returns the special constant empty-pattern when an action has been declared without input. Therefore, types only returns an empty list if no actions at all have been declared for a certain name.

Now we can perform the necessary checks on a communication declaration:

$$\frac{\begin{array}{r} Types_1 = \mathsf{types}(Act,\ N_1), \\ Types_1 \neq [], \\ \mathsf{equal}(Types_1,\ \mathsf{types}(Act,\ N_2)) = \mathsf{true}, \\ Types_1\ \mathsf{sub\text{-}of}\ \mathsf{types}(Act,\ N_3) = \mathsf{true} \end{array}}{\mathsf{check\text{-}cd}(Act,\ N_1 \mid N_2 = N_3) = \text{No errors were found}} \qquad \text{[check-cd-1]}$$

In this equation, we check the communication declaration $N_1 \mid N_2 = N_3$. In the third condition of this equation we check whether the actions declared with name N_1 have the same input patterns as the actions declared with name N_2 (the third constraint for communications of Section 3.2). In the fourth condition, we check the fourth constraint for communications of Section 3.2. Since we demand that $Types_1$ is not empty, at least one action must have been declared for each action name (the second constraint for communications of Section 3.2).

We now show the equations expressing the cases in which a communication declaration is incorrect:

$$\frac{\mathsf{types}(Act,\ N_1) = []}{\begin{array}{l} \mathsf{check\text{-}cd}(Act,\ N_1 \mid N_2 = N_3) = \\ \text{Unknown action-name } N_1 \text{ in communication declaration } N_1 \mid N_2 = N_3 \end{array}} \qquad \text{[check-cd-2]}$$

$$\frac{\mathsf{equal}(\mathsf{types}(Act,\ N_1),\ \mathsf{types}(Act,\ N_2)) \neq \mathsf{true}}{\begin{array}{l} \mathsf{check\text{-}cd}(Act,\ N_1 \mid N_2 = N_3) = \\ \text{Input-patterns of action-names in communication declaration } N_1 \mid N_2 = N_3 \\ \text{do not match} \end{array}} \qquad \text{[check-cd-3]}$$

$$\frac{\mathsf{types}(Act,\ N_1)\ \mathsf{sub\text{-}of}\ \mathsf{types}(Act,\ N_3) \neq \mathsf{true}}{\begin{array}{l} \mathsf{check\text{-}cd}(Act,\ N_1 \mid N_2 = N_3) = \\ \text{Input-patterns of action-names in communication declaration } N_1 \mid N_2 = N_3 \\ \text{do not match} \end{array}} \qquad \text{[check-cd-4]}$$

Some of the constraints on communication declarations apply in a similar way to other language constructs in μCRL. For instance, in a process with an encapsulation or a hiding construct, the set also contains only action *names*. So here too we have to make sure that actions with these names have been declared:

$$\frac{\mathsf{types}(Act,\ N) \neq []}{\mathsf{check\text{-}encap}(Act,\ N) = \text{No errors were found}} \qquad \text{[check-encap-1]}$$

$$\frac{\mathsf{types}(Act,\ N) = []}{\mathsf{check\text{-}hide}(Act,\ N) = \text{Unknown action-name } N \text{ in hiding-set}} \qquad \text{[check-hide-2]}$$

Here, N is a name from an encapsulation or hiding set that is checked.

Also renamings are only defined on action *names*, so the same check is performed on the names in a renaming declaration. Moreover, in a renaming declaration $N \rightarrow N'$, the actions with name N' should be declared with at least all the input patterns as the actions declared with name N (case **k** of the fourth constraint for processes of Section 3.2):

$$\frac{\text{types}(Act,\ N)\ \text{sub-of types}(Act,\ N') \neq \text{true}}{\begin{array}{l}\text{check-rd}(Act,\ N \rightarrow N') = \\ \text{Input-patterns of action-names in renaming declaration } N \rightarrow N' \text{ do not match}\end{array}} \quad \text{[check-rd-3]}$$

4.3.5 Miscellanea

Finally, we will show a few examples of possible errors in rewrite rules, and a possible error in a conditional process expression:

- The two sides of a rewrite rule should have the same type:

$$\frac{\text{sort}(Sig,\ Var,\ N(Trms)) \neq \text{sort}(Sig,\ Var,\ Trm)}{\begin{array}{l}\text{check-rw}(Sig,\ Var,\ N(Trms) = Trm) = \\ \text{Left-hand-side and right-hand-side of rewrite-rule } N(Trms) = Trm \\ \text{have different types}\end{array}} \quad \text{[check-rw-6]}$$

  Here, sort determines of which sort a given data term is.

- Variables in the right-hand side of a rewrite rule should also occur in the left-hand side:

$$\frac{N : N' \text{ in } Var \neq \text{true}}{\begin{array}{l}\text{vars-right-also-left}(Var,\ [N : N'\ OVds],\ Rw) = \\ \text{join(Variable } N : N' \text{ in rewrite-rule } Rw \text{ appears in right-hand-side only,} \\ \qquad \text{vars-right-also-left}(Var,\ [OVds],\ Rw))\end{array}} \quad \text{[vars-right-also-left-3]}$$

  Here, the first argument is a sequence containing all the variables that occur in the left-hand side of the rewrite rule concerned (Rw); the second argument is a sequence containing all the variables that occur in the right-hand side of Rw.

- The condition in a conditional expression should be of sort "Bool":

$$\frac{\text{sort}(Sig,\ Var,\ Trm) \neq \text{name}(\texttt{"B" "o" "o" "l"})}{\text{ssc-ce}(Sig,\ Var,\ De_1 \lhd Trm \rhd De_2) = \text{Condition } Trm \text{ is not of sort `Bool'}} \quad \text{[ssc-ce-3]}$$

  Here, $De_1 \lhd Trm \rhd De_2$ is a conditional expression with condition Trm.
  (Technical note: we use lexical syntax for the name "Bool", because there also exists a variable of the ASF+SDF sort BOOL with name "Bool" in the library.)

4.4 Well-formedness

In this section, we will give functions that check for the conditions, as formulated in Section 3.3, under which a specification is well-formed.

4.4.1 Declaration of "Bool", "T", and "F"

Checking whether a sort or a function has been declared is easy. We did that all the time when checking for static-semantical correctness. We take the sort or function element of the signature, and check whether the desired sort or function is present:

$$\frac{\mathsf{name}(\texttt{"B" "o" "o" "l"})\ \mathsf{in}\ \mathsf{sigsort}(\mathsf{sig}(E)) = \mathsf{true}}{\mathsf{bool\text{-}declared}(E) = \mathsf{No\ errors\ were\ found}} \quad \text{[bool-declared-1]}$$

$$\frac{\texttt{F :}\ \rightarrow \mathsf{name}(\texttt{"B" "o" "o" "l"})\ \mathsf{in}\ \mathsf{sigfun}(\mathsf{sig}(E)) \neq \mathsf{true}}{\mathsf{false\text{-}declared}(E) = \mathsf{Function\ `F{:}{\rightarrow}Bool'\ should\ be\ defined}} \quad \text{[false-declared-2]}$$

4.4.2 No empty sorts

To check for empty sorts, we have to make sure that there exists a data term without variables for each sort declared. This is done for a given sort name by check-sort:

$$\mathsf{check\text{-}sort}(\mathit{Fun},\ N,\ \mathit{Fun})$$

Here, N is the sort name investigated, and Fun the function element of the signature. Now check-sort will scan Fun for a function with output sort N. If such a function does not exist, sort N is empty. If such function does exist, we have to make sure that none of its input sorts are empty. From this follows that if there is a constant function with output sort N, sort N is not empty.

$$\mathsf{check\text{-}sort}([],\ N,\ \mathit{Fun}) \ = \ \mathsf{Sort}\ N\ \mathsf{is\ empty} \quad \text{[check-sort-1]}$$

$$\frac{N'' \neq N}{\mathsf{check\text{-}sort}([N' : \mathit{OXns} \rightarrow N''\ \mathit{OFds}],\ N,\ \mathit{Fun}) = \mathsf{check\text{-}sort}([\mathit{OFds}],\ N,\ \mathit{Fun})} \quad \text{[check-sort-2]}$$

$$\frac{\begin{array}{l}\mathsf{scan\text{-}sorts}(\mathsf{make\text{-}sort\text{-}sequence}([\mathit{OXns}]),\\ \qquad \mathsf{remove\text{-}fd}(\mathit{Fun},\ N' : \mathit{OXns} \rightarrow N)) = \mathsf{No\ errors\ were\ found}\end{array}}{\mathsf{check\text{-}sort}([N' : \mathit{OXns} \rightarrow N\ \mathit{OFds}],\ N,\ \mathit{Fun}) = \mathsf{No\ errors\ were\ found}} \quad \text{[check-sort-3]}$$

$$\frac{\begin{array}{l}\mathsf{scan\text{-}sorts}(\mathsf{make\text{-}sort\text{-}sequence}([\mathit{OXns}]),\\ \qquad \mathsf{remove\text{-}fd}(\mathit{Fun},\ N' : \mathit{OXns} \rightarrow N)) \neq \mathsf{No\ errors\ were\ found}\end{array}}{\mathsf{check\text{-}sort}([N' : \mathit{OXns} \rightarrow N\ \mathit{OFds}],\ N,\ \mathit{Fun}) = \mathsf{check\text{-}sort}([\mathit{OFds}],\ N,\ \mathit{Fun})} \quad \text{[check-sort-4]}$$

To avoid loops in the last two equations, we remove the function of which we check the input sorts. The auxiliary function make-sort-sequence turns an X-name-list into a sequence of sort names, by removing the #s between the names in the X-name-list.

4.4.3 Associativity of communication

Associativity of the communication function means that $(N_1 \mid N_2) \mid N_3$ equals $N_1 \mid (N_2 \mid N_3)$. Now we check the communication function for associativity in a couple of steps. First we make the commutativity of the communication function explicit, so that we do not have to work two-way all the time. This is done by the function comm-star:

$$\begin{aligned}\mathsf{comm\text{-}star}([N_1 \mid N_2 = N_3\ \mathit{OCds}]) = {} & [N_1 \mid N_2 = N_3\ N_2 \mid N_1 = N_3] \\ & + \mathsf{comm\text{-}star}([\mathit{OCds}])\end{aligned} \quad \text{[comm-star-2]}$$

Next, thirds looks for third parties for each communication declaration, that is, for each communication $N_1 \mid N_2 = N$ we look for communications $N \mid N_3 = N'$:

$$\mathsf{thirds}(N_1 \mid N_2 = N, [N \mid N_3 = N'\ OCds]) = [N \mid N_3 = N'] + \mathsf{thirds}(N_1 \mid N_2 = N, [OCds]) \quad \text{[thirds-2]}$$

$$\frac{N'' \neq N}{\mathsf{thirds}(N_1 \mid N_2 = N, [N'' \mid N_3 = N'\ OCds]) = \mathsf{thirds}(N_1 \mid N_2 = N, [OCds])} \quad \text{[thirds-3]}$$

Finally, check-third checks for each communication $N_1 \mid N_2 = N$ and a third party $N \mid N_3 = N'$ (so that $(N_1 \mid N_2) \mid N_3 = N'$) whether communications $N_2 \mid N_3 = N''$ and $N_1 \mid N'' = N'$ are also present (so that also $N_1 \mid (N_2 \mid N_3) = N'$). The actual check is performed in assoc:

$$\frac{\begin{array}{l} Comm = [OCds_1\ N_2 \mid N_3 = N''\ OCds_2], \\ Comm = [OCds_1{}'\ N_1 \mid N'' = N'\ OCds_2{}'] \end{array}}{\mathsf{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', Comm) = \mathsf{true}} \quad \text{[assoc-1]}$$

$$\frac{\mathsf{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', Comm) = \mathsf{true}}{\mathsf{check\text{-}third}(N_1 \mid N_2 = N, N \mid N_3 = N', Comm) = \mathsf{No\ errors\ were\ found}} \quad \text{[check-third-1]}$$

$$\frac{\mathsf{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', Comm) \neq \mathsf{true}}{\begin{array}{l}\mathsf{check\text{-}third}(N_1 \mid N_2 = N, N \mid N_3 = N', Comm) = \\ \mathsf{Communication}\ N_1 \mid N_2 \mid N_3\ \mathsf{is\ not\ associative}\end{array}} \quad \text{[check-third-2]}$$

We finish this section with another example of an erroneous μCRL specification. In Figure 2, we specified the alternating bit protocol. However, in the process `S`, the undefined process `X` is called. Of course this should be `S` again.

5 Related and future work

5.1 Related work

Algebraic specifications of type checkers exist for various languages. The correspondence between these specifications is some method to keep track of information concerning names and types of various elements of the language. Sometimes a special construct (environment) is defined for this purpose. In other specifications, the necessary information is extracted from the program itself.

In [Deu94], a specification for the static semantics of Pascal has been given. A *context* is defined, in which the names and types of variables, constants, functions and procedures with the types of their parameters are kept. The context may be different at different points in the program (scope rules).

A similar approach can be found in the specifications of type checkers for the toy languages PICO [BHK89c, HK89a] and ASPLE [vdM88]. Since these languages are very small, all that is being stored in the *type environment* (PICO) or *mode environment* (ASPLE) are declarations of variables.

In the specifications of type checkers for the object-oriented languages POOL [Wal89] and TKOL [Din92] the entire program is passed as an argument to the type-check function, as is the case in our specification. In the case of POOL, functions have been written to extract information about names and types from the program. In the case of TKOL, a local type environment for a specific class is generated when needed. This local type environment contains the names and types of variables and the signature of methods local to the class concerned.

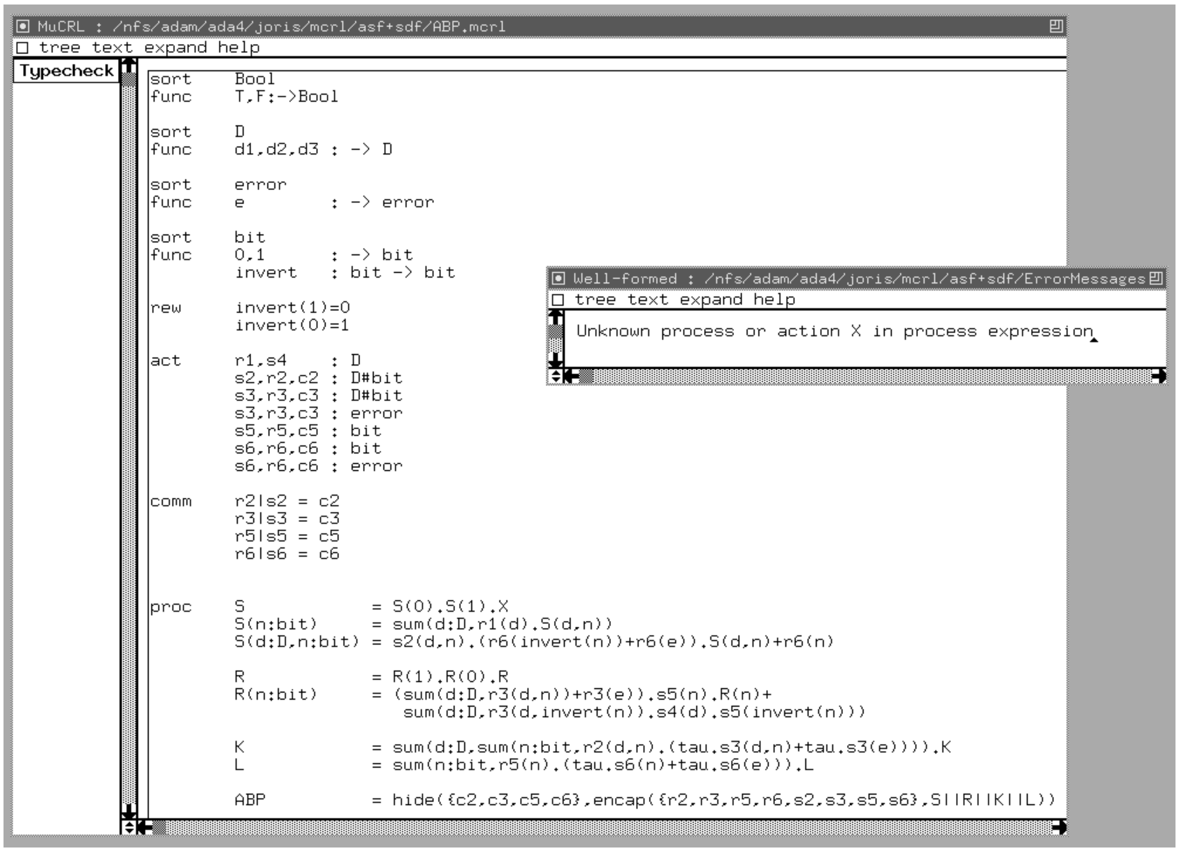

Figure 2: Specification of the alternating bit protocol with an error.

All these type checkers have in common that some type-check function is applied to the abstract-syntax tree of the program. The program is therefore checked part by part, according to the syntactical structure of the language. As a reference, either some context or environment, or the entire program is passed as an argument, containing all information necessary for type checking. This is also the approach chosen in the current paper. A μCRL program (specification) is checked part by part; the signature of the specification is passed for reference. The signature is nothing more than the original program, where all declarations of the same kind have been put together. Functions have been written to extract specific information from the context. Because μCRL has application-before-definition, a specification can not be checked from top to bottom in one pass, as would, for instance, be possible in Pascal, where forward references must be explicit.

In [Vis92], a different approach is chosen for the specification for the static semantics of the object-oriented language Eiffel. An advanced, reusable, context is defined. Now an Eiffel program, which is a number of classes, is split-up in separate classes that are put in the context. Next, the program itself is not used anymore, because all further operations are performed on the context. These further operations are a number of passes through the context. In each pass, present information is checked on internal consistency. Also, finer information is distilled from the present information. This finer information is again stored in the context and is checked in a next pass. This approach follows from the circularity (mutual dependency) that is allowed at some levels in Eiffel programs.

Origin tracking. Our well-formedness checker takes a μCRL specification and returns a list of possible shortcomings. An attractive feature would be the possibility to relate each error to its source in the μCRL specification. This principle is called *origin tracking* and is described in [DKT93].

In [Din94], a method is offered for origin tracking, departing from non-reduced type-check terms. The idea is that the type-check function returns a boolean value that is either true, or a non-standard value (a non-reduced term). This can be obtained by not specifying the "false" cases. In case of an error, some function will not reduce to true, leaving a non-reduced term. The information in this term can then be used for origin tracking. It is shown in [Din94] that for this method to work properly, one should avoid the use of conditions in the specification of the type checker. A way to remove conditions from equations in a specification is offered in [Din94].

Finally, [Deu94] shows that high-quality origins can be established for specifications from the class of *primitive recursive schemes*. These are specifications with a syntax-directed nature. This is typically the case in our ASF+SDF code where the type-check function is distributed syntax-directed over a μCRL specification.

5.2 Future work

In μCRL, process algebra is used for the specification of processes. Current research topics, such as the ToolBus [BK94], depend heavily on process algebra. Application of the verification techniques of μCRL to these systems could therefore be valuable. One can imagine detailed implementations to be proven equal to some abstract process specification. The ToolBus, however, contains some high-level process algebra features that are currently not present in μCRL. These features include iteration, priorities, state operator, and process creation. Adding these features to μCRL, and updating the well-formedness checker accordingly, can therefore be considered to be useful future work.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BHK89a] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In *Algebraic specification* [BHK89a], pages 1–66.

[BHK89c] J.A. Bergstra, J. Heering, and P. Klint. A simple programming language and its implementation. In *Algebraic specification* [BHK89a], pages 67–132.

[BK84a] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings 11th ICALP,* Antwerpen, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984. Full version appeared in [PVvV94].

[BK84b] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BK94] J.A. Bergstra and P. Klint. The TOOLBUS – a component interconnection architecture. Report P9408, Programming Research Group, University of Amsterdam, 1994.

[Bri87] E. Brinksma, editor. *Information Processing Systems – Open Systems Interconnection – LOTOS – a formal description Technique based on the temporal ordering of observational behaviour.* ISO/TC97/SC21. 1987.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[Deu94] A. van Deursen. *Executable laguage definitions – case studies and origin tracking techniques.* PhD thesis, University of Amsterdam, 1994.

[Din92] T.B. Dinesh. Object oriented programming — inheritance to adoption. Technical Report 92-02, Department of Computer Science, University of Iowa, May 1992. Ph.D. Thesis.

[Din94] T.B. Dinesh. Type checking revisited: Modular error handling. In D.J. Andrews, J.F. Groote, and C.A. Middleburg, editors, *Semantics of Specification Languages*, Workshops in Computing, pages 216–231. Springer-Verlag, 1994. Utrecht 1993.

[DKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symbolic Computation*, 15:523–545, 1993.

[GP91] J.F. Groote and A. Ponse. Proof theory for μCRL. Report CS-R9138, CWI, Amsterdam, 1991.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, pages 232–251. Workshops in Computing, Springer-Verlag, 1994. Full version available as [GP91].

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer-Verlag, 1995.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual –. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[HK89a] J. Heering and P. Klint. PICO revisited. In Bergstra et al. [BHK89a], pages 359–379.

[HK89b] J. Heering and P. Klint. The syntax definition formalism SDF. In Bergstra et al. [BHK89a], pages 283–297.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Mau91] S. Mauw. *PSF – A Process Specification Formalism.* PhD thesis, University of Amsterdam, December 1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[RBE+93] R. Reed, W. Bouma, J.D. Evans, M. Dauphin, and M. Michel, editors. *Specification and Programming Environment for Communication Software.* North-Holland, 1993.

[vdM88] E.A. van der Meulen. Algebraic specification of a compiler for a language with pointers. Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), dec 1988.

[Vis92] Eelco Visser. Syntax and static semantics of Eiffel. A case study in algebraic specification techniques. Unpublished technical report, available as ftp://ftp.cwi.nl/pub/gipe/reports/Vis92.ps.Z, December 1992.

[Wal89] H.R. Walters. The static semantics of POOL. In Bergstra et al. [BHK89a], pages 163–198.

A The specification

This appendix contains the complete specification of the well-formedness checker for μCRL. The first three modules are used to define layout and the booleans. These modules have been taken from a library. The organization of the remaining modules follows immediately from [GP95]. First sequences are defined in order to define a signature. This is followed by a module containing all error messages. Next, functions related to variables have been specified. The main function in this module checks as to whether variables in a sequence are consistent with respect to a signature. If this is the case, the sequence is called a *set of variables over* the signature. A separate module has been used, as a separate section is used in [GP95]. The reason for this is that the predicate "set of variables over" is also used outside the scope of type checking.

Module SSC contains the functions that check for static-semantic correctness. In the last module, one can find the functions that check for the remaining demands for well-formedness.

Auxiliary functions have been specified in separate modules that are imported in the modules in which the auxiliary functions are needed.

A.1 Layout

exports
  lexical syntax
    "%%"~[\n]*[\n] → LAYOUT
    [␣\t\n] → LAYOUT

A.2 Bool-con

imports Layout$^{A.1}$
exports
  sorts BOOL-CON
  context-free syntax
    true → BOOL-CON
    false → BOOL-CON
  variables
    Bool-con [*1-9*]* → BOOL-CON

A.3 Booleans

imports Bool-con$^{A.2}$
exports
  sorts BOOL
  context-free syntax
    BOOL-CON → BOOL
    BOOL "|" BOOL → BOOL {**assoc**}
    BOOL "&" BOOL → BOOL {**assoc**}
    "not" BOOL → BOOL
    "(" BOOL ")" → BOOL {**bracket**}
  variables
    Bool [*1-9'*]* → BOOL
  priorities
    "|" < "&" < "not"

equations

true | *Bool* = true [1]

false | *Bool* = *Bool* [2]

true & *Bool* = *Bool* [3]

false & *Bool* = false [4]

not true = false [5]

not false = true [6]

A.4 MuCRL

imports Layout$^{A.1}$
exports
sorts Name Sort-specification Function-specification Function-declaration Rewrite-specification Variable-declaration-section Variable-declaration Data-term Rewrite-rules-section Rewrite-rule Process-expression Parallel-expression Merge-parallel-expression Comm-parallel-expression Cond-expression Dot-expression Basic-expression Renaming-declaration Single-variable-declaration Process-specification Process-declaration Action-specification Action-declaration Communication-specification Communication-declaration Single-specification Specification
lexical syntax
[a-zA-Z0-9]+ → Name
context-free syntax

```
sort Name+                                         → Sort-specification

func Function-declaration+                         → Function-specification
{Name ","}+ ":" {Name "#"}* "→" Name               → Function-declaration

Variable-declaration-section
Rewrite-rules-section                              → Rewrite-specification
var Variable-declaration+                          → Variable-declaration-section
                                                   → Variable-declaration-section
{Name ","}+ ":" Name                               → Variable-declaration
Name                                               → Data-term
Name "(" {Data-term ","}+ ")"                      → Data-term
rew Rewrite-rule+                                  → Rewrite-rules-section
Name "=" Data-term                                 → Rewrite-rule
Name "(" {Data-term ","}+ ")" "=" Data-term        → Rewrite-rule

Parallel-expression                                → Process-expression
Parallel-expression "+" Process-expression         → Process-expression

Merge-parallel-expression                          → Parallel-expression
Comm-parallel-expression                           → Parallel-expression
Cond-expression                                    → Parallel-expression
Cond-expression "||_" Cond-expression              → Parallel-expression

Cond-expression "||"
Merge-parallel-expression                          → Merge-parallel-expression
Cond-expression "||" Cond-expression               → Merge-parallel-expression

Cond-expression "|" Comm-parallel-expression       → Comm-parallel-expression
Cond-expression "|" Cond-expression                → Comm-parallel-expression

Dot-expression                                     → Cond-expression
Dot-expression "◁" Data-term "▷"
Dot-expression                                     → Cond-expression

Basic-expression                                   → Dot-expression
Basic-expression "." Dot-expression                → Dot-expression
```

delta	→ Basic-expression
tau	→ Basic-expression
encap “(” “{” {Name “,”}+ “}” “,” Process-expression “)”	→ Basic-expression
hide “(” “{” {Name “,”}+ “}” “,” Process-expression “)”	→ Basic-expression
rename “(” “{” {Renaming-declaration “,”}+ “}” “,” Process-expression “)”	→ Basic-expression
sum “(” Single-variable-declaration “,” Process-expression “)”	→ Basic-expression
Name	→ Basic-expression
Name “(” {Data-term “,”}+ “)”	→ Basic-expression
“(” Process-expression “)”	→ Basic-expression
Name “→” Name	→ Renaming-declaration
Name “:” Name	→ Single-variable-declaration
proc Process-declaration+	→ Process-specification
Name “=” Process-expression	→ Process-declaration
Name “(” {Single-variable-declaration “,”}+ “)” “=” Process-expression	→ Process-declaration
act Action-declaration+	→ Action-specification
Name	→ Action-declaration
{Name “,”}+ “:” {Name “#”}+	→ Action-declaration
comm Communication-declaration+	→ Communication-specification
Name “|” Name “=” Name	→ Communication-declaration
Sort-specification	→ Single-specification
Function-specification	→ Single-specification
Rewrite-specification	→ Single-specification
Action-specification	→ Single-specification
Communication-specification	→ Single-specification
Process-specification	→ Single-specification
Single-specification+	→ Specification

variables

Cs [*0-9′*]+	→ CHAR+
N [*0-9′*]*	→ Name
Ns [*0-9′*]*	→ Name+
ONs [*0-9′*]*	→ Name*
Cns [*0-9′*]*	→ {Name “,”}+
OCns [*0-9′*]*	→ {Name “,”}*
Xns [*0-9′*]*	→ {Name “#”}+
OXns [*0-9′*]*	→ {Name “#”}*
Fd [*0-9′*]*	→ Function-declaration
Fds [*0-9′*]*	→ Function-declaration+
OFds [*0-9′*]*	→ Function-declaration*

Rew-spec [*0-9*′]∗	→ Rewrite-specification
Var-sec [*0-9*′]∗	→ Variable-declaration-section
Vd [*0-9*′]∗	→ Variable-declaration
Vds [*0-9*′]∗	→ Variable-declaration+
OVds [*0-9*′]∗	→ Variable-declaration∗
Trm [*0-9*′]∗	→ Data-term
Trms [*0-9*′]∗	→ {Data-term ","}+
OTrms [*0-9*′]∗	→ {Data-term ","}∗
Rw [*0-9*′]∗	→ Rewrite-rule
Rws [*0-9*′]∗	→ Rewrite-rule+
ORws [*0-9*′]∗	→ Rewrite-rule∗
Pre [*0-9*′]∗	→ Process-expression
Pae [*0-9*′]∗	→ Parallel-expression
Mpe [*0-9*′]∗	→ Merge-parallel-expression
Cpe [*0-9*′]∗	→ Comm-parallel-expression
Ce [*0-9*′]∗	→ Cond-expression
De [*0-9*′]∗	→ Dot-expression
Be [*0-9*′]∗	→ Basic-expression
Rd [*0-9*′]∗	→ Renaming-declaration
Rds [*0-9*′]∗	→ {Renaming-declaration ","}+
ORds [*0-9*′]∗	→ {Renaming-declaration ","}∗
Svd [*0-9*′]∗	→ Single-variable-declaration
Svds [*0-9*′]∗	→ {Single-variable-declaration ","}+
OSvds [*0-9*′]∗	→ {Single-variable-declaration ","}∗
Pd [*0-9*′]∗	→ Process-declaration
Pds [*0-9*′]∗	→ Process-declaration+
OPds [*0-9*′]∗	→ Process-declaration∗
Ad [*0-9*′]∗	→ Action-declaration
Ads [*0-9*′]∗	→ Action-declaration+
OAds [*0-9*′]∗	→ Action-declaration∗
Cd [*0-9*′]∗	→ Communication-declaration
Cds [*0-9*′]∗	→ Communication-declaration+
OCds [*0-9*′]∗	→ Communication-declaration∗
SE [*0-9*′]∗	→ Single-specification
SEs [*0-9*′]∗	→ Single-specification+
OSEs [*0-9*′]∗	→ Single-specification∗
E [*0-9*′]∗	→ Specification

A.5 Sequences

imports Booleans$^{A.3}$ MuCRL$^{A.4}$
exports
sorts Names Fun Act Comm Proc Var
context-free syntax

"[" Name∗ "]"	→ Names	
"[" Function-declaration∗ "]"	→ Fun	
"[" Action-declaration∗ "]"	→ Act	
"[" Communication-declaration∗ "]"	→ Comm	
"[" Process-declaration∗ "]"	→ Proc	
"[" Variable-declaration∗ "]"	→ Var	
Names "+" Names	→ Names	{**right**}
Fun "+" Fun	→ Fun	{**right**}
Act "+" Act	→ Act	{**right**}
Comm "+" Comm	→ Comm	{**right**}
Proc "+" Proc	→ Proc	{**right**}
Var "+" Var	→ Var	{**right**}
Name in Names	→ BOOL	
Function-declaration in Fun	→ BOOL	
Action-declaration in Act	→ BOOL	
Communication-declaration in Comm	→ BOOL	
Process-declaration in Proc	→ BOOL	
Variable-declaration in Var	→ BOOL	

variables

Names [0-9′]∗	→ Names
Fun [0-9′]∗	→ Fun
Act [0-9′]∗	→ Act
Comm [0-9′]∗	→ Comm
Proc [0-9′]∗	→ Proc
Var [0-9′]∗	→ Var

equations

$$[ONs_1] + [ONs_2] = [ONs_1\ ONs_2] \quad \text{[union-1]}$$
$$[OFds_1] + [OFds_2] = [OFds_1\ OFds_2] \quad \text{[union-2]}$$
$$[OAds_1] + [OAds_2] = [OAds_1\ OAds_2] \quad \text{[union-3]}$$
$$[OCds_1] + [OCds_2] = [OCds_1\ OCds_2] \quad \text{[union-4]}$$
$$[OPds_1] + [OPds_2] = [OPds_1\ OPds_2] \quad \text{[union-5]}$$
$$[OVds_1] + [OVds_2] = [OVds_1\ OVds_2] \quad \text{[union-6]}$$

$$N \text{ in } [ONs_1\ N\ ONs_2] = \text{true} \quad \text{[in-1]}$$
$$Fd \text{ in } [OFds_1\ Fd\ OFds_2] = \text{true} \quad \text{[in-2]}$$
$$Ad \text{ in } [OAds_1\ Ad\ OAds_2] = \text{true} \quad \text{[in-3]}$$
$$Cd \text{ in } [OCds_1\ Cd\ OCds_2] = \text{true} \quad \text{[in-4]}$$
$$Pd \text{ in } [OPds_1\ Pd\ OPds_2] = \text{true} \quad \text{[in-5]}$$
$$Vd \text{ in } [OVds_1\ Vd\ OVds_2] = \text{true} \quad \text{[in-6]}$$

A.6 Signature

imports Sequences$^{A.5}$
exports
sorts Signature
context-free syntax

"(" Names "," Fun "," Act "," Comm "," Proc ")" $\rightarrow$ Signature

sig(Specification) $\rightarrow$ Signature

sigsort(Signature) $\rightarrow$ Names
sigfun(Signature) $\rightarrow$ Fun
sigact(Signature) $\rightarrow$ Act
sigcomm(Signature) $\rightarrow$ Comm
sigproc(Signature) $\rightarrow$ Proc

variables

Sig [*0-9'*]$*$ $\rightarrow$ Signature

equations

split into 'single' declarations:

$$[\mathit{OFds}_1\ N, \mathit{Cns} :\ \rightarrow N'\ \mathit{OFds}_2] = [\mathit{OFds}_1\ N :\ \rightarrow N'\ \mathit{Cns} :\ \rightarrow N'\ \mathit{OFds}_2] \qquad \text{[fds-1]}$$
$$[\mathit{OFds}_1\ N, \mathit{Cns} : \mathit{Xns} \rightarrow N'\ \mathit{OFds}_2] = [\mathit{OFds}_1\ N : \mathit{Xns} \rightarrow N'\ \mathit{Cns} : \mathit{Xns} \rightarrow N'\ \mathit{OFds}_2] \qquad \text{[fds-2]}$$
$$[\mathit{OAds}_1\ N, \mathit{Cns} : \mathit{Xns}\ \mathit{OAds}_2] = [\mathit{OAds}_1\ N : \mathit{Xns}\ \mathit{Cns} : \mathit{Xns}\ \mathit{OAds}_2] \qquad \text{[ads-1]}$$

$$\mathrm{sig}(\mathrm{sort}\ \mathit{Ns}) = ([\mathit{Ns}], [], [], [], []) \qquad \text{[sig-1]}$$
$$\mathrm{sig}(\mathrm{func}\ \mathit{Fds}) = ([], [\mathit{Fds}], [], [], []) \qquad \text{[sig-2]}$$
$$\mathrm{sig}(\mathit{Rew\text{-}spec}) = ([], [], [], [], []) \qquad \text{[sig-3]}$$
$$\mathrm{sig}(\mathrm{act}\ \mathit{Ads}) = ([], [], [\mathit{Ads}], [], []) \qquad \text{[sig-4]}$$
$$\mathrm{sig}(\mathrm{comm}\ \mathit{Cds}) = ([], [], [], [\mathit{Cds}], []) \qquad \text{[sig-5]}$$
$$\mathrm{sig}(\mathrm{proc}\ \mathit{Pds}) = ([], [], [], [], [\mathit{Pds}]) \qquad \text{[sig-6]}$$

$$\frac{\begin{array}{c}\mathrm{sig}(\mathit{SE}) = (\mathit{Names}_1, \mathit{Fun}_1, \mathit{Act}_1, \mathit{Comm}_1, \mathit{Proc}_1),\\ \mathrm{sig}(\mathit{SEs}) = (\mathit{Names}_2, \mathit{Fun}_2, \mathit{Act}_2, \mathit{Comm}_2, \mathit{Proc}_2)\end{array}}{\begin{array}{l}\mathrm{sig}(\mathit{SE}\ \mathit{SEs}) = (\mathit{Names}_1 + \mathit{Names}_2,\\ \qquad \mathit{Fun}_1 + \mathit{Fun}_2,\\ \qquad \mathit{Act}_1 + \mathit{Act}_2,\\ \qquad \mathit{Comm}_1 + \mathit{Comm}_2,\\ \qquad \mathit{Proc}_1 + \mathit{Proc}_2)\end{array}} \qquad \text{[sig-7]}$$

$$\mathrm{sigsort}((\mathit{Names}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})) = \mathit{Names} \qquad \text{[sigsort-1]}$$
$$\mathrm{sigfun}((\mathit{Names}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})) = \mathit{Fun} \qquad \text{[sigfun-1]}$$
$$\mathrm{sigact}((\mathit{Names}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})) = \mathit{Act} \qquad \text{[sigact-1]}$$
$$\mathrm{sigcomm}((\mathit{Names}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})) = \mathit{Comm} \qquad \text{[sigcomm-1]}$$
$$\mathrm{sigproc}((\mathit{Names}, \mathit{Fun}, \mathit{Act}, \mathit{Comm}, \mathit{Proc})) = \mathit{Proc} \qquad \text{[sigproc-1]}$$

A.7 Errors

imports MuCRL$^{A.4}$
exports
sorts Error Errors
context-free syntax

{Error ";"}+	→ Errors
join(Errors, Errors)	→ Errors
"No" "errors" "were" "found"	→ Error
"Variable" Name "has" "been" "defined" "more" "than" "once"	→ Error
"Unknown" "sort" Name "of" "variable" Name	→ Error
Name "has" "been" "defined" "as" "both" "a" "function" "and" "a" "variable"	→ Error
Name "has" "been" "defined" "as" "both" "an" "action" "and" "a" "variable"	→ Error
Name "has" "been" "defined" "as" "both" "a" "process" "and" "a" "variable"	→ Error
"Sort" Name "has" "been" "defined" "more" "than" "once"	→ Error
"Function" Name "has" "been" "defined" "more" "than" "once"	→ Error
"Function" Name ":" {Name "#"}+ "has" "been" "defined" "more" "than" "once"	→ Error
"Unknown" "sort" Name "in" "function" "declaration" Function-declaration	→ Error
"Unknown" "function" Name "on" "left-hand-side" "of" "rewrite-rule"	→ Error
"Left-hand-side" "and" "right-hand-side" "of" "rewrite-rule" Rewrite-rule "have" "different" "types"	→ Error
"Unknown" "function" Name "(" {Name "#"}+ ")" "on" "left-hand-side" "of" "rewrite-rule"	→ Error
"Variable" Variable-declaration "in" "rewrite-rule" Rewrite-rule "appears" "in" "right-hand-side" "only"	→ Error
"Unknown" "function" "or" "variable" Name "in" "data-term"	→ Error
"Unknown" "function" Name "(" {Name "#"}+ ")" "in" "data-term"	→ Error
"Action" Action-declaration "has" "been" "defined" "more" "than" "once"	→ Error
Name "has" "been" "defined" "as" "both" "a" "function" "and" "an" "action"	→ Error
Name "(" {Name "#"}+ ")" "has"	

“been” “defined” “as” “both” “a” “function” “and” “an” “action” $\rightarrow$ Error

“Unknown”
“sort” Name “in” “action” “declaration” Action-declaration $\rightarrow$ Error

“Communication”
Name “|” Name “has” “been” “defined” “more” “than” “once” $\rightarrow$ Error

“Unknown” “action-name”
Name “in” “communication” “declaration” Communication-declaration $\rightarrow$ Error

“Input-patterns” “of” “action-names” “in” “communication”
“declaration” Communication-declaration “do” “not” “match” $\rightarrow$ Error

“Process” Name “has” “been” “defined” “more” “than” “once” $\rightarrow$ Error

“Process” Name
“(” {Name “#”}+ “)” “has” “been” “defined” “more” “than” “once” $\rightarrow$ Error

Name “has”
“been” “defined” “as” “both” “a” “function” “and” “a” “process” $\rightarrow$ Error

Name “has”
“been” “defined” “as” “both” “an” “action” “and” “a” “process” $\rightarrow$ Error

Name “(” {Name “#”}+ “)” “has”
“been” “defined” “as” “both” “a” “function” “and” “a” “process” $\rightarrow$ Error

Name “(” {Name “#”}+ “)” “has”
“been” “defined” “as” “both” “an” “action” “and” “a” “process” $\rightarrow$ Error

“Condition” Data-term “is” “not” “of” “sort” “‘Bool’” $\rightarrow$ Error

“Action-name”
Name “appears” “more” “than” “once” “in” “one” “encapsulation-set” $\rightarrow$ Error

“Unknown” “action-name” Name “in” “encapsulation-set” $\rightarrow$ Error

“Action-name”
Name “appears” “more” “than” “once” “in” “one” “hiding-set” $\rightarrow$ Error

“Unknown” “action-name” Name “in” “hiding-set” $\rightarrow$ Error

“Action-name” Name “is”
“being” “renamed” “more” “than” “once” “in” “one” “renaming-set” $\rightarrow$ Error

“Unknown” “action-name”
Name “in” “renaming” “declaration” Renaming-declaration $\rightarrow$ Error

“Input-patterns” “of” “action-names” “in”
“renaming” “declaration” Renaming-declaration “do” “not” “match” $\rightarrow$ Error

“Unknown” “process” “or” “action” Name “in” “process” “expression” $\rightarrow$ Error

“Unknown” “process”
“or” “action” Name “(” {Name “#”}+ “)” “in” “process” “expression” $\rightarrow$ Error

"Communication" Name "|" Name "|" Name "is" "not" "associative"	$\rightarrow$ Error
"Sort" Name "is" "empty"	$\rightarrow$ Error
"Sort" "'Bool'" "should" "be" "defined"	$\rightarrow$ Error
"Function" "'T:$\rightarrow$Bool'" "should" "be" "defined"	$\rightarrow$ Error
"Function" "'F:$\rightarrow$Bool'" "should" "be" "defined"	$\rightarrow$ Error

variables

Err [*0-9'*]$*$ $\rightarrow$ Error
Errs [*0-9'*]$*$ $\rightarrow$ {Error ";"}+
OErrs [*0-9'*]$*$ $\rightarrow$ {Error ";"}$*$

equations

$\text{join}(\mathit{Errs}_1, \mathit{Errs}_2) = \mathit{Errs}_1; \mathit{Errs}_2$ [join-1]

$\mathit{OErrs}_1; \mathit{Err}; \mathit{OErrs}_2; \mathit{Err}; \mathit{OErrs}_3 = \mathit{OErrs}_1; \mathit{Err}; \mathit{OErrs}_2; \mathit{OErrs}_3$ [errs-1]

OErrs; No errors were found; $\mathit{Errs} = \mathit{OErrs}; \mathit{Errs}$ [errs-2]

Errs; No errors were found; $\mathit{OErrs} = \mathit{Errs}; \mathit{OErrs}$ [errs-3]

A.8 Variables-aux

imports Signature$^{A.6}$ Errors$^{A.7}$
exports
  sorts X-name-list
  context-free syntax

{Name "#"}+	→ X-name-list
is-function(Fun, Name)	→ BOOL
is-function "(" Fun "," Name ":" X-name-list ")"	→ BOOL
is-action(Act, Name)	→ BOOL
is-process(Proc, Name)	→ BOOL
is-variable(Var, Name)	→ BOOL
Var "|" Name	→ Var

  variables
    Xlist [0-9']∗ → X-name-list
equations

$$\text{is-function}([\mathit{OFds}_1\ N : \ \to N'\ \mathit{OFds}_2], N) = \text{true} \qquad \text{[is-function-noargs-1]}$$

$$\text{is-function}([\mathit{OFds}_1\ N : \mathit{Xns} \to N'\ \mathit{OFds}_2], N : \mathit{Xns}) = \text{true} \qquad \text{[is-function-args-1]}$$

$$\text{is-action}([\mathit{OAds}_1\ N\ \mathit{OAds}_2], N) = \text{true} \qquad \text{[is-action-noargs-1]}$$

$$\text{is-process}([\mathit{OPds}_1\ N = \mathit{Pre}\ \mathit{OPds}_2], N) = \text{true} \qquad \text{[is-process-noargs-1]}$$

$$\text{is-variable}([\mathit{OVds}_1\ N : N'\ \mathit{OVds}_2], N) = \text{true} \qquad \text{[is-variable-1]}$$

$$[\mathit{OVds}_1\ N : N'\ \mathit{OVds}_2] \mid N = [\mathit{OVds}_1\ \mathit{OVds}_2] \mid N \qquad \text{[var-remove-1]}$$

$$\frac{\text{is-variable}(\mathit{Var}, N) \neq \text{true}}{\mathit{Var} \mid N = \mathit{Var}} \qquad \text{[var-remove-2]}$$

A.9 Variables

imports Variables-aux$^{A.8}$
exports
  context-free syntax
    var-over-sig(Signature, Var) → Errors

    vars(Variable-declaration-section) → Var

    sort(Signature, Var, Data-term) → Name
    sort-list(Signature, Var, {Data-term ","}+) → X-name-list

    var(Signature, Var, Data-term) → Var
    var-list(Signature, Var, {Data-term ","}+) → Var
hiddens
  context-free syntax
    uniq-vars(Var) → Errors

    scan-vars(Signature, Var) → Errors
    check-vd(Signature, Variable-declaration) → Error
equations
  split into 'single' declarations:

$$[\mathit{OVds}_1\ N, \mathit{Cns} : N'\ \mathit{OVds}_2] = [\mathit{OVds}_1\ N : N'\ \mathit{Cns} : N'\ \mathit{OVds}_2] \qquad \text{[vds-1]}$$

$$\text{vars}() = [] \qquad \text{[vars-1]}$$
$$\text{vars}(\text{var}\ \mathit{Vds}) = [\mathit{Vds}] \qquad \text{[vars-2]}$$

$$\text{var-over-sig}(\mathit{Sig}, \mathit{Var}) = \text{join}(\text{uniq-vars}(\mathit{Var}), \text{scan-vars}(\mathit{Sig}, \mathit{Var})) \qquad \text{[var-over-sig-1]}$$

$$\text{uniq-vars}([]) = \text{No errors were found} \qquad \text{[uniq-vars-1]}$$

$$\frac{\text{is-variable}([\mathit{OVds}], N) \neq \text{true}}{\text{uniq-vars}([N : N'\ \mathit{OVds}]) = \text{uniq-vars}([\mathit{OVds}])} \qquad \text{[uniq-vars-2]}$$

$$\frac{\text{is-variable}([\mathit{OVds}], N) = \text{true}}{\text{uniq-vars}([N : N'\ \mathit{OVds}]) = \text{join}(\text{Variable } N \text{ has been defined more than once}, \text{uniq-vars}([\mathit{OVds}]))} \qquad \text{[uniq-vars-3]}$$

$$\text{scan-vars}(\mathit{Sig}, []) = \text{No errors were found} \qquad \text{[scan-vars-1]}$$

$$\text{scan-vars}(\mathit{Sig}, [\mathit{Vd}\ \mathit{OVds}]) = \text{join}(\text{check-vd}(\mathit{Sig}, \mathit{Vd}), \text{scan-vars}(\mathit{Sig}, [\mathit{OVds}])) \qquad \text{[scan-vars-2]}$$

$$\frac{\begin{array}{c} N' \text{ in sigsort}(\mathit{Sig}) = \text{true}, \\ \text{is-function}(\text{sigfun}(\mathit{Sig}), N) \neq \text{true}, \\ \text{is-action}(\text{sigact}(\mathit{Sig}), N) \neq \text{true}, \\ \text{is-process}(\text{sigproc}(\mathit{Sig}), N) \neq \text{true} \end{array}}{\text{check-vd}(\mathit{Sig}, N : N') = \text{No errors were found}} \qquad \text{[check-vd-1]}$$

$$\frac{N' \text{ in sigsort}(Sig) \neq \text{true}}{\text{check-vd}(Sig,\ N : N') = \text{Unknown sort } N' \text{ of variable } N} \qquad \text{[check-vd-2]}$$

$$\frac{\text{is-function}(\text{sigfun}(Sig),\ N) = \text{true}}{\text{check-vd}(Sig,\ N : N') = N \text{ has been defined as both a function and a variable}} \qquad \text{[check-vd-3]}$$

$$\frac{\text{is-action}(\text{sigact}(Sig),\ N) = \text{true}}{\text{check-vd}(Sig,\ N : N') = N \text{ has been defined as both an action and a variable}} \qquad \text{[check-vd-4]}$$

$$\frac{\text{is-process}(\text{sigproc}(Sig),\ N) = \text{true}}{\text{check-vd}(Sig,\ N : N') = N \text{ has been defined as both a process and a variable}} \qquad \text{[check-vd-5]}$$

$$\text{sort}(Sig,\ [OVds_1\ \ N : N'\ \ OVds_2],\ N) \ = \ N' \qquad \text{[sort-1]}$$

$$\frac{\text{sigfun}(Sig) = [OFds_1\ \ N : \ \to N'\ \ OFds_2]}{\text{sort}(Sig,\ Var,\ N) = N'} \qquad \text{[sort-2]}$$

$$\frac{\begin{array}{c} Xns = \text{sort-list}(Sig,\ Var,\ Trms), \\ \text{sigfun}(Sig) = [OFds_1\ \ N : Xns \to N'\ \ OFds_2] \end{array}}{\text{sort}(Sig,\ Var,\ N(Trms)) = N'} \qquad \text{[sort-3]}$$

The next two equations describe hypothetical cases.
They have been added to make the function 'sort' complete.

$$\frac{\begin{array}{c} \text{is-variable}(Var,\ N) \neq \text{true}, \\ \text{is-function}(\text{sigfun}(Sig),\ N) \neq \text{true} \end{array}}{\text{sort}(Sig,\ Var,\ N) = \texttt{unknown}} \qquad \text{[sort-4]}$$

$$\frac{\begin{array}{c} Xns = \text{sort-list}(Sig,\ Var,\ Trms), \\ \text{is-function}(\text{sigfun}(Sig),\ N : Xns) \neq \text{true} \end{array}}{\text{sort}(Sig,\ Var,\ N(Trms)) = \texttt{unknown}} \qquad \text{[sort-5]}$$

$$\text{sort-list}(Sig,\ Var,\ Trm) \ = \ \text{sort}(Sig,\ Var,\ Trm) \qquad \text{[sort-list-1]}$$

$$\frac{Xns = \text{sort-list}(Sig,\ Var,\ Trms)}{\text{sort-list}(Sig,\ Var,\ Trm,\ Trms) = \text{sort}(Sig,\ Var,\ Trm) \#\ Xns} \qquad \text{[sort-list-2]}$$

$$\text{var}(Sig,\ [OVds_1\ \ N : N'\ \ OVds_2],\ N) \ = \ [N : N'] \qquad \text{[var-1]}$$

$$\frac{\text{is-variable}(Var,\ N) \neq \text{true}}{\text{var}(Sig,\ Var,\ N) = []} \qquad \text{[var-2]}$$

$$\text{var}(Sig,\ Var,\ N(Trms)) \ = \ \text{var-list}(Sig,\ Var,\ Trms) \qquad \text{[var-3]}$$

$$\text{var-list}(Sig,\ Var,\ Trm) \ = \ \text{var}(Sig,\ Var,\ Trm) \qquad \text{[var-list-1]}$$

$$\begin{aligned} \text{var-list}(Sig,\ Var,\ Trm,\ Trms) = {} & \text{var}(Sig,\ Var,\ Trm) \\ & + \text{var-list}(Sig,\ Var,\ Trms) \end{aligned} \qquad \text{[var-list-2]}$$

A.10 SSC-aux

imports Variables$^{A.9}$
exports
sorts Input-pattern Types
context-free syntax

empty-pattern	→ Input-pattern
{Name "#"}+	→ Input-pattern
"[" Input-pattern∗ "]"	→ Types
is-action "(" Act "," Name ":" X-name-list ")"	→ BOOL
is-comm "(" Comm "," Name "|" Name ")"	→ BOOL
is-process "(" Proc "," Name "(" X-name-list ")" ")"	→ BOOL
types(Act, Name)	→ Types
Types "+" Types	→ Types
Input-pattern in Types	→ BOOL
Types sub-of Types	→ BOOL
equal(Types, Types)	→ BOOL
make-var-sequence "(" "[" {Single-variable-declaration ","}∗ "]" ")"	→ Var
var-sorts "(" "[" {Single-variable-declaration ","}+ "]" ")"	→ X-name-list
Name in "[" {Name ","}∗ "]"	→ BOOL
is-renamed "(" "[" {Renaming-declaration ","}∗ "]" "," Name ")"	→ BOOL

variables

Ip [0-9′]∗ → Input-pattern
Ips [0-9′]∗ → Input-pattern+
OIps [0-9′]∗ → Input-pattern∗

Types [0-9′]∗ → Types

equations

$$\text{is-action}([\mathit{OAds}_1\ N : \mathit{Xns}\ \mathit{OAds}_2], N : \mathit{Xns}) \ = \ \text{true} \qquad \text{[is-action-args-1]}$$

$$\text{is-comm}([\mathit{OCds}_1\ N_1 \mid N_2 = N_3\ \mathit{OCds}_2], N_1 \mid N_2) \ = \ \text{true} \qquad \text{[is-comm-1]}$$
$$\text{is-comm}([\mathit{OCds}_1\ N_2 \mid N_1 = N_3\ \mathit{OCds}_2], N_1 \mid N_2) \ = \ \text{true} \qquad \text{[is-comm-2]}$$

$$\frac{\text{var-sorts}([\mathit{Svds}]) = \mathit{Xns}}{\text{is-process}([\mathit{OPds}_1\ N(\mathit{Svds}) = \mathit{Pre}\ \mathit{OPds}_2], N(\mathit{Xns})) = \text{true}} \qquad \text{[is-process-args-1]}$$

$$\text{types}([], N) \ = \ [] \qquad \text{[types-1]}$$
$$\text{types}([N\ \mathit{OAds}], N) \ = \ [\text{empty-pattern}] + \text{types}([\mathit{OAds}], N) \qquad \text{[types-2]}$$
$$\text{types}([N : \mathit{Xns}\ \mathit{OAds}], N) \ = \ [\mathit{Xns}] + \text{types}([\mathit{OAds}], N) \qquad \text{[types-3]}$$

$$\frac{N' \neq N}{\text{types}([N'\ \mathit{OAds}], N) = \text{types}([\mathit{OAds}], N)} \qquad \text{[types-4]}$$

$$\frac{N' \neq N}{\mathsf{types}([N' : Xns\ OAds], N) = \mathsf{types}([OAds], N)}$$ [types-5]

$[OIps_1] + [OIps_2] \ = \ [OIps_1\ OIps_2]$ [types-union-1]

Ip in $[OIps_1\ Ip\ OIps_2] \ = \ \mathsf{true}$ [in-types-1]

$[]$ sub-of $Types \ = \ \mathsf{true}$ [sub-of-1]

$$\begin{aligned}[Ip\ OIps] \text{ sub-of } Types = {} & Ip \text{ in } Types \\ & \&\ [OIps] \text{ sub-of } Types\end{aligned}$$ [sub-of-2]

$$\begin{aligned}\mathsf{equal}(Types_1, Types_2) = {} & Types_1 \text{ sub-of } Types_2 \\ & \&\ Types_2 \text{ sub-of } Types_1\end{aligned}$$ [equal-1]

$\mathsf{make\text{-}var\text{-}sequence}([]) \ = \ []$ [make-var-sequence-1]

$$\begin{aligned}\mathsf{make\text{-}var\text{-}sequence}([N : N', OSvds]) = {} & [N : N'] \\ & + \mathsf{make\text{-}var\text{-}sequence}([OSvds])\end{aligned}$$ [make-var-sequence-2]

$\mathsf{var\text{-}sorts}([N : N']) \ = \ N'$ [var-sorts-1]

$$\frac{Xns = \mathsf{var\text{-}sorts}([Svds])}{\mathsf{var\text{-}sorts}([N : N', Svds]) = N' \# Xns}$$ [var-sorts-2]

N in$[OCns_1, N, OCns_2] \ = \ \mathsf{true}$ [in-name-list-1]

$\mathsf{is\text{-}renamed}([ORds_1, N \rightarrow N', ORds_2], N) \ = \ \mathsf{true}$ [is-renamed-1]

A.11 SSC

imports SSC-aux$^{A.10}$
exports
  context-free syntax
    ssc(Specification) → Errors
hiddens
  context-free syntax

```
ssc-spec(Signature, Specification)                                      → Errors

uniq-sorts(Names)                                                       → Errors

uniq-fun(Fun)                                                           → Errors
scan-fun(Names, Fun)                                                    → Errors
sorts-defined-in-fun "(" Names "," "[" {Name "#"}* "]" ","
Function-declaration ")"                                                → Errors

ssc-var(Signature, Variable-declaration-section)                        → Errors
scan-rew "(" Signature "," Var "," "[" Rewrite-rule* "]" ")"            → Errors
check-rw(Signature, Var, Rewrite-rule)                                  → Errors
vars-right-also-left(Var, Var, Rewrite-rule)                            → Errors

ssc-t(Signature, Var, Data-term)                                        → Errors
ssc-terms "(" Signature "," Var "," "[" {Data-term ","}* "]" ")"        → Errors

uniq-act(Act)                                                           → Errors
scan-act(Signature, Act)                                                → Errors
check-ad(Signature, Action-declaration)                                 → Errors
sorts-defined-in-act "(" Names "," "[" {Name "#"}* "]" ","
Action-declaration ")"                                                  → Errors

uniq-comm(Comm)                                                         → Errors
scan-comm(Act, Comm)                                                    → Errors
check-cd(Act, Communication-declaration)                                → Error

uniq-proc(Proc)                                                         → Errors
scan-proc(Signature, Proc)                                              → Errors
check-pd(Signature, Process-declaration)                                → Errors

ssc-pre(Signature, Var, Process-expression)                             → Errors
ssc-pae(Signature, Var, Parallel-expression)                            → Errors
ssc-mpe(Signature, Var, Merge-parallel-expression)                      → Errors
ssc-cpe(Signature, Var, Comm-parallel-expression)                       → Errors
ssc-ce(Signature, Var, Cond-expression)                                 → Errors
ssc-de(Signature, Var, Dot-expression)                                  → Errors
ssc-be(Signature, Var, Basic-expression)                                → Errors

uniq-encaps "(" "[" {Name ","}* "]" ")"                                 → Errors
scan-encaps "(" Act "," "[" {Name ","}* "]" ")"                         → Errors
check-encap(Act, Name)                                                  → Error

uniq-hides "(" "[" {Name ","}* "]" ")"                                  → Errors
scan-hides "(" Act "," "[" {Name ","}* "]" ")"                          → Errors
check-hide(Act, Name)                                                   → Error
```

uniq-renamings "(" "[" {Renaming-declaration ","}* "]" ")" → Errors
scan-renamings "(" Act "," "[" {Renaming-declaration ","}* "]" ")" → Errors
check-rd(Act, Renaming-declaration) → Error

no-sort-overlap(Names, Names) → Errors
no-fun-overlap(Fun, Fun) → Errors
no-act-overlap(Act, Act) → Errors
no-comm-overlap(Comm, Comm) → Errors
no-proc-overlap(Proc, Proc) → Errors

equations

ssc-spec(*Sig*, sort *Ns*) = uniq-sorts([*Ns*]) [ssc-spec-1]

uniq-sorts([]) = No errors were found [uniq-sorts-1]

$$\frac{N \text{ in } [ONs] \neq \text{true}}{\text{uniq-sorts}([N\ ONs]) = \text{uniq-sorts}([ONs])} \quad \text{[uniq-sorts-2]}$$

$$\frac{N \text{ in } [ONs] = \text{true}}{\text{uniq-sorts}([N\ ONs]) = \text{join(Sort } N \text{ has been defined more than once, uniq-sorts}([ONs]))} \quad \text{[uniq-sorts-3]}$$

ssc-spec(*Sig*, func *Fds*) = join(uniq-fun([*Fds*]),
scan-fun(sigsort(*Sig*), [*Fds*])) [ssc-spec-2]

uniq-fun([]) = No errors were found [uniq-fun-1]

$$\frac{\text{is-function}([OFds], N) \neq \text{true}}{\text{uniq-fun}([N : \ \rightarrow N'\ OFds]) = \text{uniq-fun}([OFds])} \quad \text{[uniq-fun-2]}$$

$$\frac{\text{is-function}([OFds], N) = \text{true}}{\text{uniq-fun}([N : \ \rightarrow N'\ OFds]) = \text{join(Function } N \text{ has been defined more than once, uniq-fun}([OFds]))} \quad \text{[uniq-fun-3]}$$

$$\frac{\text{is-function}([OFds], N : Xns) \neq \text{true}}{\text{uniq-fun}([N : Xns \rightarrow N'\ OFds]) = \text{uniq-fun}([OFds])} \quad \text{[uniq-fun-4]}$$

$$\frac{\text{is-function}([OFds], N : Xns) = \text{true}}{\text{uniq-fun}([N : Xns \rightarrow N'\ OFds]) = \text{join(Function } N : Xns \text{ has been defined more than once, uniq-fun}([OFds]))} \quad \text{[uniq-fun-5]}$$

scan-fun(*Names*, []) = No errors were found [scan-fun-1]

$$\text{scan-fun}(\mathit{Names},\ [N : \mathit{OXns} \rightarrow N'\ \mathit{OFds}]) = \text{join}(\text{sorts-defined-in-fun}(\mathit{Names},\ [\mathit{OXns}\#\ N'],\ N : \mathit{OXns} \rightarrow N'),\ \text{scan-fun}(\mathit{Names},\ [\mathit{OFds}])) \qquad \text{[scan-fun-2]}$$

$$\text{sorts-defined-in-fun}(\mathit{Names},\ [],\ \mathit{Fd}) = \text{No errors were found} \qquad \text{[sorts-defined-in-fun-1]}$$

$$\frac{N \text{ in } \mathit{Names} = \text{true}}{\text{sorts-defined-in-fun}(\mathit{Names},\ [N\#\ \mathit{OXns}],\ \mathit{Fd}) = \text{sorts-defined-in-fun}(\mathit{Names},\ [\mathit{OXns}],\ \mathit{Fd})} \qquad \text{[sorts-defined-in-fun-1]}$$

$$\frac{N \text{ in } \mathit{Names} \neq \text{true}}{\text{sorts-defined-in-fun}(\mathit{Names},\ [N\#\ \mathit{OXns}],\ \mathit{Fd}) = \text{join}(\text{Unknown sort } N \text{ in function declaration } \mathit{Fd},\ \text{sorts-defined-in-fun}(\mathit{Names},\ [\mathit{OXns}],\ \mathit{Fd}))} \qquad \text{[sorts-defined-in-fun-1]}$$

$$\text{ssc-spec}(\mathit{Sig},\ \mathit{Var\text{-}sec}\ \text{rew}\ \mathit{Rws}) = \text{join}(\text{ssc-var}(\mathit{Sig},\ \mathit{Var\text{-}sec}),\ \text{scan-rew}(\mathit{Sig},\ \text{vars}(\mathit{Var\text{-}sec}),\ [\mathit{Rws}])) \qquad \text{[ssc-spec-3]}$$

$$\text{ssc-var}(\mathit{Sig},\ ) = \text{No errors were found} \qquad \text{[ssc-var-1]}$$

$$\text{ssc-var}(\mathit{Sig},\ \text{var}\ \mathit{Vds}) = \text{var-over-sig}(\mathit{Sig},\ [\mathit{Vds}]) \qquad \text{[ssc-var-2]}$$

$$\text{scan-rew}(\mathit{Sig},\ \mathit{Var},\ []) = \text{No errors were found} \qquad \text{[scan-rew-1]}$$

$$\text{scan-rew}(\mathit{Sig},\ \mathit{Var},\ [\mathit{Rw}\ \mathit{ORws}]) = \text{join}(\text{check-rw}(\mathit{Sig},\ \mathit{Var},\ \mathit{Rw}),\ \text{scan-rew}(\mathit{Sig},\ \mathit{Var},\ [\mathit{ORws}])) \qquad \text{[scan-rew-2]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N) = \text{true},\quad \text{sort}(\mathit{Sig},\ \mathit{Var},\ N) = \text{sort}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trm})}{\text{check-rw}(\mathit{Sig},\ \mathit{Var},\ N = \mathit{Trm}) = \text{ssc-t}(\mathit{Sig},\ [],\ \mathit{Trm})} \qquad \text{[check-rw-1]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N) \neq \text{true}}{\text{check-rw}(\mathit{Sig},\ \mathit{Var},\ N = \mathit{Trm}) = \text{Unknown function } N \text{ on left-hand-side of rewrite-rule}} \qquad \text{[check-rw-2]}$$

$$\frac{\text{sort}(\mathit{Sig},\ \mathit{Var},\ N) \neq \text{sort}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trm})}{\text{check-rw}(\mathit{Sig},\ \mathit{Var},\ N = \mathit{Trm}) = \text{Left-hand-side and right-hand-side of rewrite-rule } N = \mathit{Trm} \text{ have different types}} \qquad \text{[check-rw-3]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N : \text{sort-list}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trms})) = \text{true},\quad \text{sort}(\mathit{Sig},\ \mathit{Var},\ N(\mathit{Trms})) = \text{sort}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trm})}{\text{check-rw}(\mathit{Sig},\ \mathit{Var},\ N(\mathit{Trms}) = \mathit{Trm}) = \text{join}(\text{vars-right-also-left}(\text{var-list}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trms}),\ \text{var}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trm}),\ N(\mathit{Trms}) = \mathit{Trm}),\ \text{join}(\text{ssc-t}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trm}),\ \text{ssc-terms}(\mathit{Sig},\ \mathit{Var},\ [\mathit{Trms}])))} \qquad \text{[check-rw-4]}$$

$$\frac{\begin{array}{r}\textit{Xns} = \text{sort-list}(\textit{Sig}, \textit{Var}, \textit{Trms}),\\ \text{is-function}(\text{sigfun}(\textit{Sig}), N : \textit{Xns}) \neq \text{true}\end{array}}{\begin{array}{l}\text{check-rw}(\textit{Sig}, \textit{Var}, N(\textit{Trms}) = \textit{Trm}) =\\ \text{Unknown function } N(\textit{Xns}) \text{ on left-hand-side of rewrite-rule}\end{array}} \quad \text{[check-rw-5]}$$

$$\frac{\text{sort}(\textit{Sig}, \textit{Var}, N(\textit{Trms})) \neq \text{sort}(\textit{Sig}, \textit{Var}, \textit{Trm})}{\begin{array}{l}\text{check-rw}(\textit{Sig}, \textit{Var}, N(\textit{Trms}) = \textit{Trm}) =\\ \text{Left-hand-side and right-hand-side of rewrite-rule } N(\textit{Trms}) = \textit{Trm} \text{ have different types}\end{array}} \quad \text{[check-rw-6]}$$

$$\text{vars-right-also-left}(\textit{Var}, [], \textit{Rw}) = \text{No errors were found} \quad \text{[vars-right-also-left-1]}$$

$$\frac{N : N' \text{ in } \textit{Var} = \text{true}}{\begin{array}{l}\text{vars-right-also-left}(\textit{Var}, [N : N' \ \textit{OVds}], \textit{Rw}) =\\ \text{vars-right-also-left}(\textit{Var}, [\textit{OVds}], \textit{Rw})\end{array}} \quad \text{[vars-right-also-left-2]}$$

$$\frac{N : N' \text{ in } \textit{Var} \neq \text{true}}{\begin{array}{l}\text{vars-right-also-left}(\textit{Var}, [N : N' \ \textit{OVds}], \textit{Rw}) =\\ \text{join(Variable } N : N' \text{ in rewrite-rule } \textit{Rw} \text{ appears in right-hand-side only},\\ \qquad \text{vars-right-also-left}(\textit{Var}, [\textit{OVds}], \textit{Rw}))\end{array}} \quad \text{[vars-right-also-left-3]}$$

$$\frac{\text{is-variable}(\textit{Var}, N) = \text{true}}{\text{ssc-t}(\textit{Sig}, \textit{Var}, N) = \text{No errors were found}} \quad \text{[ssc-t-1]}$$

$$\frac{\text{is-function}(\text{sigfun}(\textit{Sig}), N) = \text{true}}{\text{ssc-t}(\textit{Sig}, \textit{Var}, N) = \text{No errors were found}} \quad \text{[ssc-t-2]}$$

$$\frac{\begin{array}{r}\text{is-variable}(\textit{Var}, N) \neq \text{true},\\ \text{is-function}(\text{sigfun}(\textit{Sig}), N) \neq \text{true}\end{array}}{\text{ssc-t}(\textit{Sig}, \textit{Var}, N) = \text{Unknown function or variable } N \text{ in data-term}} \quad \text{[ssc-t-3]}$$

$$\frac{\text{is-function}(\text{sigfun}(\textit{Sig}), N : \text{sort-list}(\textit{Sig}, \textit{Var}, \textit{Trms})) = \text{true}}{\text{ssc-t}(\textit{Sig}, \textit{Var}, N(\textit{Trms})) = \text{ssc-terms}(\textit{Sig}, \textit{Var}, [\textit{Trms}])} \quad \text{[ssc-t-4]}$$

$$\frac{\begin{array}{r}\textit{Xns} = \text{sort-list}(\textit{Sig}, \textit{Var}, \textit{Trms}),\\ \text{is-function}(\text{sigfun}(\textit{Sig}), N : \textit{Xns}) \neq \text{true}\end{array}}{\text{ssc-t}(\textit{Sig}, \textit{Var}, N(\textit{Trms})) = \text{Unknown function } N(\textit{Xns}) \text{ in data-term}} \quad \text{[ssc-t-5]}$$

$$\text{ssc-terms}(\textit{Sig}, \textit{Var}, []) = \text{No errors were found} \quad \text{[ssc-terms-1]}$$

$$\begin{array}{l}\text{ssc-terms}(\textit{Sig}, \textit{Var}, [\textit{Trm}, \textit{OTrms}]) = \text{join}(\text{ssc-t}(\textit{Sig}, \textit{Var}, \textit{Trm}),\\ \qquad\qquad\qquad \text{ssc-terms}(\textit{Sig}, \textit{Var}, [\textit{OTrms}]))\end{array} \quad \text{[ssc-terms-2]}$$

$$\text{ssc-spec}(\mathit{Sig}, \text{act } \mathit{Ads}) = \text{join}(\text{uniq-act}([\mathit{Ads}]),\ \text{scan-act}(\mathit{Sig}, [\mathit{Ads}])) \quad \text{[ssc-spec-4]}$$

$$\text{uniq-act}([]) = \text{No errors were found} \quad \text{[uniq-act-1]}$$

$$\frac{\mathit{Ad} \text{ in } [\mathit{OAds}] \neq \text{true}}{\text{uniq-act}([\mathit{Ad}\ \mathit{OAds}]) = \text{uniq-act}([\mathit{OAds}])} \quad \text{[uniq-act-2]}$$

$$\frac{\mathit{Ad} \text{ in } [\mathit{OAds}] = \text{true}}{\text{uniq-act}([\mathit{Ad}\ \mathit{OAds}]) = \text{join}(\text{Action } \mathit{Ad} \text{ has been defined more than once},\ \text{uniq-act}([\mathit{OAds}]))} \quad \text{[uniq-act-3]}$$

$$\text{scan-act}(\mathit{Sig}, []) = \text{No errors were found} \quad \text{[scan-act-1]}$$

$$\text{scan-act}(\mathit{Sig}, [\mathit{Ad}\ \mathit{OAds}]) = \text{join}(\text{check-ad}(\mathit{Sig}, \mathit{Ad}),\ \text{scan-act}(\mathit{Sig}, [\mathit{OAds}])) \quad \text{[scan-act-2]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}), N) \neq \text{true}}{\text{check-ad}(\mathit{Sig}, N) = \text{No errors were found}} \quad \text{[check-ad-1]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}), N) = \text{true}}{\text{check-ad}(\mathit{Sig}, N) = N \text{ has been defined as both a function and an action}} \quad \text{[check-ad-2]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}), N : \mathit{Xns}) \neq \text{true}}{\text{check-ad}(\mathit{Sig}, N : \mathit{Xns}) = \text{sorts-defined-in-act}(\text{sigsort}(\mathit{Sig}), [\mathit{Xns}], N : \mathit{Xns})} \quad \text{[check-ad-3]}$$

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}), N : \mathit{Xns}) = \text{true}}{\text{check-ad}(\mathit{Sig}, N : \mathit{Xns}) = N(\mathit{Xns}) \text{ has been defined as both a function and an action}} \quad \text{[check-ad-4]}$$

$$\text{sorts-defined-in-act}(\mathit{Names}, [], \mathit{Ad}) = \text{No errors were found} \quad \text{[sorts-defined-in-act-1]}$$

$$\frac{N \text{ in } \mathit{Names} = \text{true}}{\text{sorts-defined-in-act}(\mathit{Names}, [N\#\ \mathit{OXns}], \mathit{Ad}) = \text{sorts-defined-in-act}(\mathit{Names}, [\mathit{OXns}], \mathit{Ad})} \quad \text{[sorts-defined-in-act-2]}$$

$$\frac{N \text{ in } \mathit{Names} \neq \text{true}}{\text{sorts-defined-in-act}(\mathit{Names}, [N\#\ \mathit{OXns}], \mathit{Ad}) = \text{join}(\text{Unknown sort } N \text{ in action declaration } \mathit{Ad},\ \text{sorts-defined-in-act}(\mathit{Names}, [\mathit{OXns}], \mathit{Ad}))} \quad \text{[sorts-defined-in-act-3]}$$

$$\text{ssc-spec}(\mathit{Sig}, \text{comm } \mathit{Cds}) = \text{join}(\text{uniq-comm}([\mathit{Cds}]),\ \text{scan-comm}(\text{sigact}(\mathit{Sig}), [\mathit{Cds}])) \quad \text{[ssc-spec-5]}$$

uniq-comm([]) = No errors were found [uniq-comm-1]

$$\frac{\text{is-comm}([\mathit{OCds}],\ N_1 \mid N_2) \neq \text{true}}{\text{uniq-comm}([N_1 \mid N_2 = N_3\ \mathit{OCds}]) = \text{uniq-comm}([\mathit{OCds}])}$$ [uniq-comm-2]

$$\frac{\text{is-comm}([\mathit{OCds}],\ N_1 \mid N_2) = \text{true}}{\begin{array}{l}\text{uniq-comm}([N_1 \mid N_2 = N_3\ \mathit{OCds}]) = \\ \text{join(Communication } N_1 \mid N_2 \text{ has been defined more than once,} \\ \quad \text{uniq-comm}([\mathit{OCds}]))\end{array}}$$ [uniq-comm-3]

scan-comm(Act, []) = No errors were found [scan-comm-1]

$$\begin{array}{l}\text{scan-comm}(\mathit{Act},\ [\mathit{Cd}\ \mathit{OCds}]) = \text{join(check-cd}(\mathit{Act},\ \mathit{Cd}), \\ \qquad\qquad \text{scan-comm}(\mathit{Act},\ [\mathit{OCds}]))\end{array}$$ [scan-comm-2]

$$\frac{\begin{array}{c}\mathit{Types}_1 = \text{types}(\mathit{Act},\ N_1), \\ \mathit{Types}_1 \neq [], \\ \text{equal}(\mathit{Types}_1,\ \text{types}(\mathit{Act},\ N_2)) = \text{true}, \\ \mathit{Types}_1 \text{ sub-of types}(\mathit{Act},\ N_3) = \text{true}\end{array}}{\text{check-cd}(\mathit{Act},\ N_1 \mid N_2 = N_3) = \text{No errors were found}}$$ [check-cd-1]

$$\frac{\text{types}(\mathit{Act},\ N_1) = []}{\begin{array}{l}\text{check-cd}(\mathit{Act},\ N_1 \mid N_2 = N_3) = \text{Unknown action-name } N_1 \\ \qquad \text{in communication declaration } N_1 \mid N_2 = N_3\end{array}}$$ [check-cd-2]

$$\frac{\text{equal(types}(\mathit{Act},\ N_1),\ \text{types}(\mathit{Act},\ N_2)) \neq \text{true}}{\begin{array}{l}\text{check-cd}(\mathit{Act},\ N_1 \mid N_2 = N_3) = \\ \text{Input-patterns of action-names in communication declaration } N_1 \mid N_2 = N_3 \\ \text{do not match}\end{array}}$$ [check-cd-3]

$$\frac{\text{types}(\mathit{Act},\ N_1) \text{ sub-of types}(\mathit{Act},\ N_3) \neq \text{true}}{\begin{array}{l}\text{check-cd}(\mathit{Act},\ N_1 \mid N_2 = N_3) = \\ \text{Input-patterns of action-names in communication declaration } N_1 \mid N_2 = N_3 \\ \text{do not match}\end{array}}$$ [check-cd-4]

$$\begin{array}{l}\text{ssc-spec}(\mathit{Sig},\ \text{proc } \mathit{Pds}) = \text{join(uniq-proc}([\mathit{Pds}]), \\ \qquad\qquad \text{scan-proc}(\mathit{Sig},\ [\mathit{Pds}]))\end{array}$$ [ssc-spec-6]

uniq-proc([]) = No errors were found [uniq-proc-1]

$$\frac{\text{is-process}([\mathit{OPds}],\ N) \neq \text{true}}{\text{uniq-proc}([N = \mathit{Pre}\ \mathit{OPds}]) = \text{uniq-proc}([\mathit{OPds}])}$$ [uniq-proc-2]

$$\frac{\text{is-process}([\mathit{OPds}],\ N) = \text{true}}{\text{uniq-proc}([N = \mathit{Pre}\ \mathit{OPds}]) = \text{join}(\text{Process } N \text{ has been defined more than once},\ \text{uniq-proc}([\mathit{OPds}]))}$$ [uniq-proc-3]

$$\frac{\text{is-process}([\mathit{OPds}],\ N(\text{var-sorts}([\mathit{Svds}]))) \neq \text{true}}{\text{uniq-proc}([N(\mathit{Svds}) = \mathit{Pre}\ \mathit{OPds}]) = \text{uniq-proc}([\mathit{OPds}])}$$ [uniq-proc-4]

$$\frac{\mathit{Xns} = \text{var-sorts}([\mathit{Svds}]),\quad \text{is-process}([\mathit{OPds}],\ N(\mathit{Xns})) = \text{true}}{\text{uniq-proc}([N(\mathit{Svds}) = \mathit{Pre}\ \mathit{OPds}]) = \text{join}(\text{Process } N(\mathit{Xns}) \text{ has been defined more than once},\ \text{uniq-proc}([\mathit{OPds}]))}$$ [uniq-proc-5]

$$\text{scan-proc}(\mathit{Sig},\ []) = \text{No errors were found}$$ [scan-proc-1]

$$\text{scan-proc}(\mathit{Sig},\ [\mathit{Pd}\ \mathit{OPds}]) = \text{join}(\text{check-pd}(\mathit{Sig},\ \mathit{Pd}),\ \text{scan-proc}(\mathit{Sig},\ [\mathit{OPds}]))$$ [scan-proc-2]

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N) \neq \text{true},\quad \text{is-action}(\text{sigact}(\mathit{Sig}),\ N) \neq \text{true}}{\text{check-pd}(\mathit{Sig},\ N = \mathit{Pre}) = \text{ssc-pre}(\mathit{Sig},\ [],\ \mathit{Pre})}$$ [check-pd-1]

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N) = \text{true}}{\text{check-pd}(\mathit{Sig},\ N = \mathit{Pre}) = N \text{ has been defined as both a function and a process}}$$ [check-pd-2]

$$\frac{\text{is-action}(\text{sigact}(\mathit{Sig}),\ N) = \text{true}}{\text{check-pd}(\mathit{Sig},\ N = \mathit{Pre}) = N \text{ has been defined as both an action and a process}}$$ [check-pd-3]

$$\frac{\text{is-function}(\text{sigfun}(\mathit{Sig}),\ N : \text{var-sorts}([\mathit{Svds}])) \neq \text{true},\quad \text{is-action}(\text{sigact}(\mathit{Sig}),\ N : \text{var-sorts}([\mathit{Svds}])) \neq \text{true},\quad \mathit{Var} = \text{make-var-sequence}([\mathit{Svds}])}{\text{check-pd}(\mathit{Sig},\ N(\mathit{Svds}) = \mathit{Pre}) = \text{join}(\text{var-over-sig}(\mathit{Sig},\ \mathit{Var}),\ \text{ssc-pre}(\mathit{Sig},\ \mathit{Var},\ \mathit{Pre}))}$$ [check-pd-4]

$$\frac{\mathit{Xns} = \text{var-sorts}([\mathit{Svds}]),\quad \text{is-function}(\text{sigfun}(\mathit{Sig}),\ N : \mathit{Xns}) = \text{true}}{\text{check-pd}(\mathit{Sig},\ N(\mathit{Svds}) = \mathit{Pre}) = N(\mathit{Xns}) \text{ has been defined as both a function and a process}}$$ [check-pd-5]

$$\frac{\mathit{Xns} = \text{var-sorts}([\mathit{Svds}]),\quad \text{is-action}(\text{sigact}(\mathit{Sig}),\ N : \mathit{Xns}) = \text{true}}{\text{check-pd}(\mathit{Sig},\ N(\mathit{Svds}) = \mathit{Pre}) = N(\mathit{Xns}) \text{ has been defined as both an action and a process}}$$ [check-pd-6]

$$\text{ssc-pre}(\mathit{Sig}, \mathit{Var}, \mathit{Pae}) = \text{ssc-pae}(\mathit{Sig}, \mathit{Var}, \mathit{Pae}) \qquad \text{[ssc-pre-1]}$$

$$\text{ssc-pre}(\mathit{Sig}, \mathit{Var}, \mathit{Pae} + \mathit{Pre}) = \text{join}(\text{ssc-pae}(\mathit{Sig}, \mathit{Var}, \mathit{Pae}), \text{ssc-pre}(\mathit{Sig}, \mathit{Var}, \mathit{Pre})) \qquad \text{[ssc-pre-2]}$$

$$\text{ssc-pae}(\mathit{Sig}, \mathit{Var}, \mathit{Mpe}) = \text{ssc-mpe}(\mathit{Sig}, \mathit{Var}, \mathit{Mpe}) \qquad \text{[ssc-pae-1]}$$
$$\text{ssc-pae}(\mathit{Sig}, \mathit{Var}, \mathit{Cpe}) = \text{ssc-cpe}(\mathit{Sig}, \mathit{Var}, \mathit{Cpe}) \qquad \text{[ssc-pae-2]}$$
$$\text{ssc-pae}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}) = \text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}) \qquad \text{[ssc-pae-3]}$$

$$\text{ssc-pae}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_1 \parallel\_\ \mathit{Ce}_2) = \text{join}(\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_1), \text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_2)) \qquad \text{[ssc-pae-4]}$$

$$\text{ssc-mpe}(\mathit{Sig}, \mathit{Var}, \mathit{Ce} \parallel \mathit{Mpe}) = \text{join}(\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}), \text{ssc-mpe}(\mathit{Sig}, \mathit{Var}, \mathit{Mpe})) \qquad \text{[ssc-mpe-1]}$$

$$\text{ssc-mpe}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_1 \parallel \mathit{Ce}_2) = \text{join}(\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_1), \text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_2)) \qquad \text{[ssc-mpe-2]}$$

$$\text{ssc-cpe}(\mathit{Sig}, \mathit{Var}, \mathit{Ce} \mid \mathit{Cpe}) = \text{join}(\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}), \text{ssc-cpe}(\mathit{Sig}, \mathit{Var}, \mathit{Cpe})) \qquad \text{[ssc-cpe-1]}$$

$$\text{ssc-cpe}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_1 \mid \mathit{Ce}_2) = \text{join}(\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_1), \text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{Ce}_2)) \qquad \text{[ssc-cpe-2]}$$

$$\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{De}) = \text{ssc-de}(\mathit{Sig}, \mathit{Var}, \mathit{De}) \qquad \text{[ssc-ce-1]}$$

$$\frac{\text{sort}(\mathit{Sig}, \mathit{Var}, \mathit{Trm}) = \text{name}(\texttt{"B" "o" "o" "l"})}{\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{De}_1 \lhd \mathit{Trm} \rhd \mathit{De}_2) = \text{join}(\text{ssc-t}(\mathit{Sig}, \mathit{Var}, \mathit{Trm}), \text{join}(\text{ssc-de}(\mathit{Sig}, \mathit{Var}, \mathit{De}_1), \text{ssc-de}(\mathit{Sig}, \mathit{Var}, \mathit{De}_2)))} \qquad \text{[ssc-ce-2]}$$

$$\frac{\text{sort}(\mathit{Sig}, \mathit{Var}, \mathit{Trm}) \neq \text{name}(\texttt{"B" "o" "o" "l"})}{\text{ssc-ce}(\mathit{Sig}, \mathit{Var}, \mathit{De}_1 \lhd \mathit{Trm} \rhd \mathit{De}_2) = \text{Condition } \mathit{Trm} \text{ is not of sort `Bool'}} \qquad \text{[ssc-ce-3]}$$

$$\text{ssc-de}(\mathit{Sig}, \mathit{Var}, \mathit{Be}) = \text{ssc-be}(\mathit{Sig}, \mathit{Var}, \mathit{Be}) \qquad \text{[ssc-de-1]}$$

$$\text{ssc-de}(\mathit{Sig}, \mathit{Var}, \mathit{Be} \,.\, \mathit{De}) = \text{join}(\text{ssc-be}(\mathit{Sig}, \mathit{Var}, \mathit{Be}), \text{ssc-de}(\mathit{Sig}, \mathit{Var}, \mathit{De})) \qquad \text{[ssc-de-2]}$$

$$\text{ssc-be}(\mathit{Sig}, \mathit{Var}, \text{delta}) = \text{No errors were found} \qquad \text{[ssc-be-1]}$$
$$\text{ssc-be}(\mathit{Sig}, \mathit{Var}, \text{tau}) = \text{No errors were found} \qquad \text{[ssc-be-2]}$$

$$\text{ssc-be}(\mathit{Sig}, \mathit{Var}, \text{encap}(\{\mathit{Cns}\}, \mathit{Pre})) = \text{join}(\text{uniq-encaps}([\mathit{Cns}]), \text{join}(\text{scan-encaps}(\text{sigact}(\mathit{Sig}), [\mathit{Cns}]), \text{ssc-pre}(\mathit{Sig}, \mathit{Var}, \mathit{Pre}))) \qquad \text{[ssc-be-3]}$$

$$\text{uniq-encaps}([]) = \text{No errors were found} \qquad \text{[uniq-encaps-1]}$$

$$\frac{N \text{ in}[\mathit{OCns}] \neq \text{true}}{\text{uniq-encaps}([N, \mathit{OCns}]) = \text{uniq-encaps}([\mathit{OCns}])} \qquad \text{[uniq-encaps-2]}$$

$$\frac{N \text{ in}[\mathit{OCns}] = \text{true}}{\text{uniq-encaps}([N, \mathit{OCns}]) = \text{join}(\text{Action-name } N \text{ appears more than once in one encapsulation-set}, \text{uniq-encaps}([\mathit{OCns}]))} \qquad \text{[uniq-encaps-3]}$$

$$\text{scan-encaps}(\mathit{Act}, []) = \text{No errors were found} \qquad \text{[scan-encaps-1]}$$

$$\text{scan-encaps}(\mathit{Act}, [N, \mathit{OCns}]) = \text{join}(\text{check-encap}(\mathit{Act}, N), \text{scan-encaps}(\mathit{Act}, [\mathit{OCns}])) \qquad \text{[scan-encaps-2]}$$

$$\frac{\text{types}(\mathit{Act}, N) \neq []}{\text{check-encap}(\mathit{Act}, N) = \text{No errors were found}} \qquad \text{[check-encap-1]}$$

$$\frac{\text{types}(\mathit{Act}, N) = []}{\text{check-encap}(\mathit{Act}, N) = \text{Unknown action-name } N \text{ in encapsulation-set}} \qquad \text{[check-encap-2]}$$

$$\text{ssc-be}(\mathit{Sig}, \mathit{Var}, \text{hide}(\{\mathit{Cns}\}, \mathit{Pre})) = \text{join}(\text{uniq-hides}([\mathit{Cns}]), \text{join}(\text{scan-hides}(\text{sigact}(\mathit{Sig}), [\mathit{Cns}]), \text{ssc-pre}(\mathit{Sig}, \mathit{Var}, \mathit{Pre}))) \qquad \text{[ssc-be-4]}$$

$$\text{uniq-hides}([]) = \text{No errors were found} \qquad \text{[uniq-hides-1]}$$

$$\frac{N \text{ in}[\mathit{OCns}] \neq \text{true}}{\text{uniq-hides}([N, \mathit{OCns}]) = \text{uniq-hides}([\mathit{OCns}])} \qquad \text{[uniq-hides-2]}$$

$$\frac{N \text{ in}[\mathit{OCns}] = \text{true}}{\text{uniq-hides}([N, \mathit{OCns}]) = \text{join}(\text{Action-name } N \text{ appears more than once in one hiding-set}, \text{uniq-hides}([\mathit{OCns}]))} \qquad \text{[uniq-hides-3]}$$

$$\text{scan-hides}(\mathit{Act}, []) = \text{No errors were found} \qquad \text{[scan-hides-1]}$$

$$\text{scan-hides}(\mathit{Act}, [N, \mathit{OCns}]) = \text{join}(\text{check-hide}(\mathit{Act}, N), \text{scan-hides}(\mathit{Act}, [\mathit{OCns}])) \qquad \text{[scan-hides-2]}$$

$$\frac{\text{types}(Act,\ N) \neq []}{\text{check-hide}(Act,\ N) = \text{No errors were found}} \quad \text{[check-hide-1]}$$

$$\frac{\text{types}(Act,\ N) = []}{\text{check-hide}(Act,\ N) = \text{Unknown action-name } N \text{ in hiding-set}} \quad \text{[check-hide-2]}$$

$$\begin{aligned}\text{ssc-be}(Sig,\ Var,\ \text{rename}(\{Rds\},\ Pre)) = \text{join}(&\text{uniq-renamings}([Rds]),\\ &\text{join}(\text{scan-renamings}(\text{sigact}(Sig),\ [Rds]),\\ &\quad \text{ssc-pre}(Sig,\ Var,\ Pre)))\end{aligned} \quad \text{[ssc-be-5]}$$

$$\text{uniq-renamings}([]) \ = \ \text{No errors were found} \quad \text{[uniq-renamings-1]}$$

$$\frac{\text{is-renamed}([ORds],\ N) \neq \text{true}}{\text{uniq-renamings}([N \to N',\ ORds]) = \text{uniq-renamings}([ORds])} \quad \text{[uniq-renamings-2]}$$

$$\frac{\text{is-renamed}([ORds],\ N) = \text{true}}{\begin{array}{l}\text{uniq-renamings}([N \to N',\ ORds]) =\\ \text{join}(\text{Action-name } N \text{ is being renamed more than once in one renaming-set},\\ \qquad \text{uniq-renamings}([ORds]))\end{array}} \quad \text{[uniq-renamings-3]}$$

$$\text{scan-renamings}(Act,\ []) \ = \ \text{No errors were found} \quad \text{[scan-renamings-1]}$$

$$\begin{aligned}\text{scan-renamings}(Act,\ [Rd,\ ORds]) = \text{join}(&\text{check-rd}(Act,\ Rd),\\ &\text{scan-renamings}(Act,\ [ORds]))\end{aligned} \quad \text{[scan-renamings-2]}$$

$$\frac{\begin{array}{c}Types = \text{types}(Act,\ N),\\ Types \neq [],\\ Types \text{ sub-of types}(Act,\ N') = \text{true}\end{array}}{\text{check-rd}(Act,\ N \to N') = \text{No errors were found}} \quad \text{[check-rd-1]}$$

$$\frac{\text{types}(Act,\ N) = []}{\begin{array}{l}\text{check-rd}(Act,\ N \to N') = \text{Unknown action-name } N\\ \qquad\qquad\qquad\qquad\quad \text{in renaming declaration } N \to N'\end{array}} \quad \text{[check-rd-2]}$$

$$\frac{\text{types}(Act,\ N) \text{ sub-of types}(Act,\ N') \neq \text{true}}{\begin{array}{l}\text{check-rd}(Act,\ N \to N') =\\ \text{Input-patterns of action-names in renaming declaration } N \to N' \text{ do not match}\end{array}} \quad \text{[check-rd-3]}$$

$$\begin{aligned}\text{ssc-be}(Sig,\ Var,\ \text{sum}(N : N',\ Pre)) = \text{join}(&\text{var-over-sig}(Sig,\ Var \mid N + [N : N']),\\ &\text{ssc-pre}(Sig,\ Var \mid N + [N : N'],\ Pre))\end{aligned} \quad \text{[ssc-be-6]}$$

$$\frac{\text{is-process}(\text{sigproc}(Sig),\ N) = \text{true}}{\text{ssc-be}(Sig,\ Var,\ N) = \text{No errors were found}} \quad \text{[ssc-be-7]}$$

$$\frac{\text{is-action}(\text{sigact}(\mathit{Sig}),\ \mathit{N}) = \text{true}}{\text{ssc-be}(\mathit{Sig},\ \mathit{Var},\ \mathit{N}) = \text{No errors were found}} \quad \text{[ssc-be-8]}$$

$$\frac{\begin{array}{c}\text{is-process}(\text{sigproc}(\mathit{Sig}),\ \mathit{N}) \neq \text{true},\\ \text{is-action}(\text{sigact}(\mathit{Sig}),\ \mathit{N}) \neq \text{true}\end{array}}{\text{ssc-be}(\mathit{Sig},\ \mathit{Var},\ \mathit{N}) = \text{Unknown process or action } \mathit{N} \text{ in process expression}} \quad \text{[ssc-be-9]}$$

$$\frac{\text{is-process}(\text{sigproc}(\mathit{Sig}),\ \mathit{N}(\text{sort-list}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trms}))) = \text{true}}{\text{ssc-be}(\mathit{Sig},\ \mathit{Var},\ \mathit{N}(\mathit{Trms})) = \text{ssc-terms}(\mathit{Sig},\ \mathit{Var},\ [\mathit{Trms}])} \quad \text{[ssc-be-10]}$$

$$\frac{\text{is-action}(\text{sigact}(\mathit{Sig}),\ \mathit{N} : \text{sort-list}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trms})) = \text{true}}{\text{ssc-be}(\mathit{Sig},\ \mathit{Var},\ \mathit{N}(\mathit{Trms})) = \text{ssc-terms}(\mathit{Sig},\ \mathit{Var},\ [\mathit{Trms}])} \quad \text{[ssc-be-11]}$$

$$\frac{\begin{array}{c}\mathit{Xns} = \text{sort-list}(\mathit{Sig},\ \mathit{Var},\ \mathit{Trms}),\\ \text{is-process}(\text{sigproc}(\mathit{Sig}),\ \mathit{N}(\mathit{Xns})) \neq \text{true},\\ \text{is-action}(\text{sigact}(\mathit{Sig}),\ \mathit{N} : \mathit{Xns}) \neq \text{true}\end{array}}{\text{ssc-be}(\mathit{Sig},\ \mathit{Var},\ \mathit{N}(\mathit{Trms})) = \text{Unknown process or action } \mathit{N}(\mathit{Xns}) \text{ in process expression}} \quad \text{[ssc-be-12]}$$

$$\text{ssc-be}(\mathit{Sig},\ \mathit{Var},\ (\mathit{Pre})) \ = \ \text{ssc-pre}(\mathit{Sig},\ \mathit{Var},\ \mathit{Pre}) \quad \text{[ssc-be-13]}$$

$$\begin{array}{l}\text{ssc-spec}(\mathit{Sig},\ \mathit{SE}\ \mathit{SEs}) = \qquad \text{[ssc-spec-7]}\\ \text{join}(\text{no-sort-overlap}(\text{sigsort}(\text{sig}(\mathit{SE})),\ \text{sigsort}(\text{sig}(\mathit{SEs}))),\\ \quad \text{join}(\text{no-fun-overlap}(\text{sigfun}(\text{sig}(\mathit{SE})),\ \text{sigfun}(\text{sig}(\mathit{SEs}))),\\ \quad\quad \text{join}(\text{no-act-overlap}(\text{sigact}(\text{sig}(\mathit{SE})),\ \text{sigact}(\text{sig}(\mathit{SEs}))),\\ \quad\quad\quad \text{join}(\text{no-comm-overlap}(\text{sigcomm}(\text{sig}(\mathit{SE})),\ \text{sigcomm}(\text{sig}(\mathit{SEs}))),\\ \quad\quad\quad\quad \text{join}(\text{no-proc-overlap}(\text{sigproc}(\text{sig}(\mathit{SE})),\ \text{sigproc}(\text{sig}(\mathit{SEs}))),\\ \quad\quad\quad\quad\quad \text{join}(\text{ssc-spec}(\mathit{Sig},\ \mathit{SE}),\\ \quad\quad\quad\quad\quad\quad \text{ssc-spec}(\mathit{Sig},\ \mathit{SEs})))))))\end{array}$$

$$\text{no-sort-overlap}([\,],\ \mathit{Names}) \ = \ \text{No errors were found} \quad \text{[no-sort-overlap-1]}$$

$$\frac{\mathit{N} \text{ in } \mathit{Names} \neq \text{true}}{\text{no-sort-overlap}([\mathit{N}\ \mathit{ONs}],\ \mathit{Names}) = \text{no-sort-overlap}([\mathit{ONs}],\ \mathit{Names})} \quad \text{[no-sort-overlap-2]}$$

$$\frac{\mathit{N} \text{ in } \mathit{Names} = \text{true}}{\begin{array}{r}\text{no-sort-overlap}([\mathit{N}\ \mathit{ONs}],\ \mathit{Names}) = \text{join}(\text{Sort } \mathit{N} \text{ has been defined more than once},\\ \text{no-sort-overlap}([\mathit{ONs}],\ \mathit{Names}))\end{array}} \quad \text{[no-sort-overlap-3]}$$

$$\text{no-fun-overlap}([\,],\ \mathit{Fun}) \ = \ \text{No errors were found} \quad \text{[no-fun-overlap-1]}$$

$$\frac{\text{is-function}(\mathit{Fun},\ \mathit{N}) \neq \text{true}}{\text{no-fun-overlap}([\mathit{N} :\ \rightarrow \mathit{N}'\ \mathit{OFds}],\ \mathit{Fun}) = \text{no-fun-overlap}([\mathit{OFds}],\ \mathit{Fun})} \quad \text{[no-fun-overlap-2]}$$

$$\frac{\text{is-function}(\mathit{Fun},\ N) = \text{true}}{\begin{array}{l}\text{no-fun-overlap}([N:\ \to N'\ \mathit{OFds}],\ \mathit{Fun}) = \\ \text{join}(\text{Function } N \text{ has been defined more than once,} \\ \quad \text{no-fun-overlap}([\mathit{OFds}],\ \mathit{Fun}))\end{array}} \quad \text{[no-fun-overlap-3]}$$

$$\frac{\text{is-function}(\mathit{Fun},\ N : \mathit{Xns}) \neq \text{true}}{\text{no-fun-overlap}([N : \mathit{Xns} \to N'\ \mathit{OFds}],\ \mathit{Fun}) = \text{no-fun-overlap}([\mathit{OFds}],\ \mathit{Fun})} \quad \text{[no-fun-overlap-4]}$$

$$\frac{\text{is-function}(\mathit{Fun},\ N : \mathit{Xns}) = \text{true}}{\begin{array}{l}\text{no-fun-overlap}([N : \mathit{Xns} \to N'\ \mathit{OFds}],\ \mathit{Fun}) = \\ \text{join}(\text{Function } N : \mathit{Xns} \text{ has been defined more than once,} \\ \quad \text{no-fun-overlap}([\mathit{OFds}],\ \mathit{Fun}))\end{array}} \quad \text{[no-fun-overlap-5]}$$

$$\text{no-act-overlap}([],\ \mathit{Act}) \ = \ \text{No errors were found} \quad \text{[no-act-overlap-1]}$$

$$\frac{\mathit{Ad} \text{ in } \mathit{Act} \neq \text{true}}{\text{no-act-overlap}([\mathit{Ad}\ \mathit{OAds}],\ \mathit{Act}) = \text{no-act-overlap}([\mathit{OAds}],\ \mathit{Act})} \quad \text{[no-act-overlap-2]}$$

$$\frac{\mathit{Ad} \text{ in } \mathit{Act} = \text{true}}{\begin{array}{l}\text{no-act-overlap}([\mathit{Ad}\ \mathit{OAds}],\ \mathit{Act}) = \text{join}(\text{Action } \mathit{Ad} \text{ has been defined more than once,} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{no-act-overlap}([\mathit{OAds}],\ \mathit{Act}))\end{array}} \quad \text{[no-act-overlap-3]}$$

$$\text{no-comm-overlap}([],\ \mathit{Comm}) \ = \ \text{No errors were found} \quad \text{[no-comm-overlap-1]}$$

$$\frac{\text{is-comm}(\mathit{Comm},\ N_1 \mid N_2) \neq \text{true}}{\begin{array}{l}\text{no-comm-overlap}([N_1 \mid N_2 = N_3\ \mathit{OCds}],\ \mathit{Comm}) = \\ \text{no-comm-overlap}([\mathit{OCds}],\ \mathit{Comm})\end{array}} \quad \text{[no-comm-overlap-2]}$$

$$\frac{\text{is-comm}(\mathit{Comm},\ N_1 \mid N_2) = \text{true}}{\begin{array}{l}\text{no-comm-overlap}([N_1 \mid N_2 = N_3\ \mathit{OCds}],\ \mathit{Comm}) = \\ \text{join}(\text{Communication } N_1 \mid N_2 \text{ has been defined more than once,} \\ \quad \text{no-comm-overlap}([\mathit{OCds}],\ \mathit{Comm}))\end{array}} \quad \text{[no-comm-overlap-3]}$$

$$\text{no-proc-overlap}([],\ \mathit{Proc}) \ = \ \text{No errors were found} \quad \text{[no-proc-overlap-1]}$$

$$\frac{\text{is-process}(\mathit{Proc},\ N) \neq \text{true}}{\text{no-proc-overlap}([N = \mathit{Pre}\ \mathit{OPds}],\ \mathit{Proc}) = \text{no-proc-overlap}([\mathit{OPds}],\ \mathit{Proc})} \quad \text{[no-proc-overlap-2]}$$

$$\frac{\text{is-process}(\mathit{Proc},\ N) = \text{true}}{\begin{array}{l}\text{no-proc-overlap}([N = \mathit{Pre}\ \mathit{OPds}],\ \mathit{Proc}) = \\ \text{join}(\text{Process } N \text{ has been defined more than once,} \\ \quad \text{no-proc-overlap}([\mathit{OPds}],\ \mathit{Proc}))\end{array}} \quad \text{[no-proc-overlap-3]}$$

$$\frac{\text{is-process}(\mathit{Proc},\ \mathit{N}(\text{var-sorts}([\mathit{Svds}]))) \neq \text{true}}{\text{no-proc-overlap}([\mathit{N}(\mathit{Svds}) = \mathit{Pre}\ \mathit{OPds}],\ \mathit{Proc}) = \text{no-proc-overlap}([\mathit{OPds}],\ \mathit{Proc})} \quad \text{[no-proc-overlap-4]}$$

$$\frac{\begin{array}{r}\mathit{Xns} = \text{var-sorts}([\mathit{Svds}]),\\ \text{is-process}(\mathit{Proc},\ \mathit{N}(\mathit{Xns})) = \text{true}\end{array}}{\begin{array}{l}\text{no-proc-overlap}([\mathit{N}(\mathit{Svds}) = \mathit{Pre}\ \mathit{OPds}],\ \mathit{Proc}) =\\ \text{join}(\text{Process } \mathit{N}(\mathit{Xns}) \text{ has been defined more than once},\\ \quad \text{no-proc-overlap}([\mathit{OPds}],\ \mathit{Proc}))\end{array}} \quad \text{[no-proc-overlap-5]}$$

$$\text{ssc}(\mathit{E}) \ = \ \text{ssc-spec}(\text{sig}(\mathit{E}),\ \mathit{E}) \quad \text{[ssc-1]}$$

A.12 Well-formed-aux

imports SSC$^{A.11}$
exports
context-free syntax

$$\begin{array}{ll}
\text{comm-star(Comm)} & \rightarrow \text{Comm} \\
\text{thirds(Communication-declaration, Comm)} & \rightarrow \text{Comm} \\
\text{assoc(Communication-declaration,} & \\
\text{Communication-declaration, Comm)} & \rightarrow \text{BOOL} \\
\text{make-sort-sequence “(” “[” \{Name “\#”\}* “]” “)”} & \rightarrow \text{Names} \\
\text{remove-fd(Fun, Function-declaration)} & \rightarrow \text{Fun}
\end{array}$$

equations

$$\text{comm-star}([]) \;=\; [] \qquad \text{[comm-star-1]}$$

$$\text{comm-star}([N_1 \mid N_2 = N_3 \; OCds]) = [N_1 \mid N_2 = N_3 \; N_2 \mid N_1 = N_3] + \text{comm-star}([OCds]) \qquad \text{[comm-star-2]}$$

$$\text{thirds}(N_1 \mid N_2 = N, []) \;=\; [] \qquad \text{[thirds-1]}$$

$$\text{thirds}(N_1 \mid N_2 = N, [N \mid N_3 = N' \; OCds]) = [N \mid N_3 = N'] + \text{thirds}(N_1 \mid N_2 = N, [OCds]) \qquad \text{[thirds-2]}$$

$$\frac{N'' \neq N}{\text{thirds}(N_1 \mid N_2 = N, [N'' \mid N_3 = N' \; OCds]) = \text{thirds}(N_1 \mid N_2 = N, [OCds])} \qquad \text{[thirds-3]}$$

$$\frac{\begin{array}{l} Comm = [OCds_1 \; N_2 \mid N_3 = N'' \; OCds_2], \\ Comm = [OCds_1{}' \; N_1 \mid N'' = N' \; OCds_2{}'] \end{array}}{\text{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', Comm) = \text{true}} \qquad \text{[assoc-1]}$$

$$\text{make-sort-sequence}([]) \;=\; [] \qquad \text{[make-sort-sequence-1]}$$

$$\text{make-sort-sequence}([N\# \; OXns]) = [N] + \text{make-sort-sequence}([OXns]) \qquad \text{[make-sort-sequence-2]}$$

$$\text{remove-fd}([], Fd) \;=\; [] \qquad \text{[remove-fd-1]}$$

$$\text{remove-fd}([Fd \; OFds], Fd) \;=\; \text{remove-fd}([OFds], Fd) \qquad \text{[remove-fd-2]}$$

$$\frac{Fd' \neq Fd}{\text{remove-fd}([Fd' \; OFds], Fd) = [Fd'] + \text{remove-fd}([OFds], Fd)} \qquad \text{[remove-fd-3]}$$

A.13 Well-formed

imports Well-formed-aux$^{A.12}$
exports
  context-free syntax
    well-formed(Specification) → Errors
hiddens
  context-free syntax
    comm-assoc(Specification) → Errors
    scan-comm-star(Comm, Comm) → Errors
    scan-thirds(Communication-declaration, Comm, Comm) → Errors
    check-third(Communication-declaration,
    Communication-declaration, Comm) → Error

    no-empty-sorts(Specification) → Errors
    scan-sorts(Names, Fun) → Errors
    check-sort(Fun, Name, Fun) → Error

    bool-declared(Specification) → Error

    true-declared(Specification) → Error

    false-declared(Specification) → Error
equations

$$\text{comm-assoc}(E) = \text{scan-comm-star}(\text{comm-star}(\text{sigcomm}(\text{sig}(E))), \text{comm-star}(\text{sigcomm}(\text{sig}(E)))) \quad \text{[comm-assoc-1]}$$

$$\text{scan-comm-star}([], \mathit{Comm}) = \text{No errors were found} \quad \text{[scan-comm-star-1]}$$

$$\frac{\mathit{Comm}' = \text{thirds}(\mathit{Cd}, \mathit{Comm})}{\text{scan-comm-star}([\mathit{Cd}\ \mathit{OCds}], \mathit{Comm}) = \text{join}(\text{scan-thirds}(\mathit{Cd}, \mathit{Comm}', \mathit{Comm}), \text{scan-comm-star}([\mathit{OCds}], \mathit{Comm}))} \quad \text{[scan-comm-star-2]}$$

$$\text{scan-thirds}(\mathit{Cd}, [], \mathit{Comm}) = \text{No errors were found} \quad \text{[scan-thirds-1]}$$

$$\text{scan-thirds}(\mathit{Cd}, [\mathit{Cd}'\ \mathit{OCds}], \mathit{Comm}) = \text{join}(\text{check-third}(\mathit{Cd}, \mathit{Cd}', \mathit{Comm}), \text{scan-thirds}(\mathit{Cd}, [\mathit{OCds}], \mathit{Comm})) \quad \text{[scan-thirds-2]}$$

$$\frac{\text{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', \mathit{Comm}) = \text{true}}{\text{check-third}(N_1 \mid N_2 = N, N \mid N_3 = N', \mathit{Comm}) = \text{No errors were found}} \quad \text{[check-third-1]}$$

$$\frac{\text{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', \mathit{Comm}) \neq \text{true}}{\text{check-third}(N_1 \mid N_2 = N, N \mid N_3 = N', \mathit{Comm}) = \text{Communication } N_1 \mid N_2 \mid N_3 \text{ is not associative}} \quad \text{[check-third-2]}$$

The next equation describes a hypothetical case.
It has been added to make the function 'check-third' complete.

$$\frac{N'' \neq N}{\text{check-third}(N_1 \mid N_2 = N,\, N'' \mid N_3 = N',\, \mathit{Comm}) = \text{No errors were found}} \quad \text{[check-third-3]}$$

$$\text{no-empty-sorts}(E) = \text{scan-sorts}(\text{sigsort}(\text{sig}(E)),\, \text{sigfun}(\text{sig}(E))) \quad \text{[no-empty-sorts-1]}$$

$$\text{scan-sorts}([\,],\, \mathit{Fun}) = \text{No errors were found} \quad \text{[scan-sorts-1]}$$

$$\text{scan-sorts}([N\ \mathit{ONs}],\, \mathit{Fun}) = \text{join}(\text{check-sort}(\mathit{Fun},\, N,\, \mathit{Fun}),\ \text{scan-sorts}([\mathit{ONs}],\, \mathit{Fun})) \quad \text{[scan-sorts-2]}$$

$$\text{check-sort}([\,],\, N,\, \mathit{Fun}) = \text{Sort } N \text{ is empty} \quad \text{[check-sort-1]}$$

$$\frac{N'' \neq N}{\text{check-sort}([N' : \mathit{OXns} \to N''\ \mathit{OFds}],\, N,\, \mathit{Fun}) = \text{check-sort}([\mathit{OFds}],\, N,\, \mathit{Fun})} \quad \text{[check-sort-2]}$$

$$\frac{\text{scan-sorts}(\text{make-sort-sequence}([\mathit{OXns}]),\ \text{remove-fd}(\mathit{Fun},\, N' : \mathit{OXns} \to N)) = \text{No errors were found}}{\text{check-sort}([N' : \mathit{OXns} \to N\ \mathit{OFds}],\, N,\, \mathit{Fun}) = \text{No errors were found}} \quad \text{[check-sort-3]}$$

$$\frac{\text{scan-sorts}(\text{make-sort-sequence}([\mathit{OXns}]),\ \text{remove-fd}(\mathit{Fun},\, N' : \mathit{OXns} \to N)) \neq \text{No errors were found}}{\text{check-sort}([N' : \mathit{OXns} \to N\ \mathit{OFds}],\, N,\, \mathit{Fun}) = \text{check-sort}([\mathit{OFds}],\, N,\, \mathit{Fun})} \quad \text{[check-sort-4]}$$

$$\frac{\text{name}(\texttt{"B" "o" "o" "l"}) \text{ in sigsort}(\text{sig}(E)) = \text{true}}{\text{bool-declared}(E) = \text{No errors were found}} \quad \text{[bool-declared-1]}$$

$$\frac{\text{name}(\texttt{"B" "o" "o" "l"}) \text{ in sigsort}(\text{sig}(E)) \neq \text{true}}{\text{bool-declared}(E) = \text{Sort 'Bool' should be defined}} \quad \text{[bool-declared-2]}$$

$$\frac{\texttt{T} : \ \to \text{name}(\texttt{"B" "o" "o" "l"}) \text{ in sigfun}(\text{sig}(E)) = \text{true}}{\text{true-declared}(E) = \text{No errors were found}} \quad \text{[true-declared-1]}$$

$$\frac{\texttt{T} : \ \to \text{name}(\texttt{"B" "o" "o" "l"}) \text{ in sigfun}(\text{sig}(E)) \neq \text{true}}{\text{true-declared}(E) = \text{Function 'T:}\to\text{Bool' should be defined}} \quad \text{[true-declared-2]}$$

$$\frac{\texttt{F} : \ \to \text{name}(\texttt{"B" "o" "o" "l"}) \text{ in sigfun}(\text{sig}(E)) = \text{true}}{\text{false-declared}(E) = \text{No errors were found}} \quad \text{[false-declared-1]}$$

$$\frac{\texttt{F :} \ \rightarrow \mathsf{name}(\texttt{"B" "o" "o" "l"}) \ \mathsf{in} \ \mathsf{sigfun}(\mathsf{sig}(E)) \neq \mathsf{true}}{\mathsf{false\text{-}declared}(E) = \mathsf{Function\ `F{:}{\rightarrow}Bool'\ should\ be\ defined}} \qquad \text{[false-declared-2]}$$

$$\begin{aligned}\mathsf{well\text{-}formed}(E) = \mathsf{join}(&\mathsf{ssc}(E),\\ &\mathsf{join}(\mathsf{comm\text{-}assoc}(E),\\ &\quad \mathsf{join}(\mathsf{no\text{-}empty\text{-}sorts}(E),\\ &\qquad \mathsf{join}(\mathsf{bool\text{-}declared}(E),\\ &\qquad\quad \mathsf{join}(\mathsf{true\text{-}declared}(E),\\ &\qquad\qquad \mathsf{false\text{-}declared}(E))))))\end{aligned} \qquad \text{[well-formed-1]}$$