University of Amsterdam
Programming Research Group

Reuse of ASF+SDF Specifications by means of Renaming

Merijn de Jonge

Report P9718 october 1997

University of Amsterdam

Department of Computer Science

Programming Research Group

Reuse of asf+sdf specifications by means of renaming

Merijn de Jonge

Report P9718

october 1997

M. de Jonge

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7586
e-mail: mdejonge@wins.uva.nl

Reuse of ASF+SDF Specifications by means of Renaming

Merijn de Jonge

Doctoraalscriptie Informatica, Universiteit van Amsterdam, Faculteit WINS, sectie Programmatuur.

Afstudeerdocent:	prof. dr. P. Klint
Begeleiders:	dr. E. Visser
	dr. M. G. J. van den Brand

Contents

Chapter 1

Introduction

Reuse shortens the time to develop software systems and it promotes standardization. Often, it is difficult to extract elements from one project that can be easily used in an unrelated project because of dependencies between portions of code. Decreasing the dependencies between components is therefore considered a good design principle for the promotion of reuse (Thunem and Sindre, 1992). In Stevens *et al.* (1974) relations between components are described in terms of *coupling* and *cohesion*. These terms describe the relationships *between* components and the relationships *within* a component respectively. In order to promote reuse, a component should be designed as orthogonal as possible (Thunem and Sindre, 1992) by lowering the coupling between modules and increasing the internal cohesion.

Several techniques exist for the development of reusable orthogonal components. Examples are *higher-order* programming techniques in functional languages, *inheritance* in object-oriented languages and *import* and *parameterization* mechanisms in modular languages. This thesis will concentrate on reuse in the algebraic specification formalism ASF+SDF. Because it is a modular formalism, we will concentrate on import and parameterization as techniques for the design and reuse of orthogonal modules.

1.1 ASF+SDF

ASF+SDF is a modular algebraic specification formalism with first-order, many-sorted signatures and conditional equations. It is the union of the algebraic specification formalism ASF (Bergstra *et al.*, 1989) and the syntax definition formalism SDF (Heering *et al.*, 1989).

ASF is a modular formalism for specifying abstract data types. An ASF module consist of a signature defining sorts and first-order functions. The semantical part of a module is formed by a set of conditional equations defining the properties of the data types. To support the reuse of specifications, ASF contains an import and parameterization mechanism.

SDF allows the simultaneous definition of concrete (i.e., lexical and context-free syntax) and abstract syntax and it implicitly defines a translation from text strings to abstract syntax trees.

Together, ASF and SDF form a powerful formalism for the definition of syntax and semantics of programming languages. The SDF part is used to define the concrete syntax of a language. The equations forming the ASF part of a specification can be defined directly on the concrete constructs. Hence, SDF serves as front-end for ASF in the formalism. Being a modular formalism, ASF+SDF supports (like ASF) an import mechanism. However, unlike ASF, it does not provide a parameterization mechanism.

Development of specifications is supported by the ASF+SDF Meta-Environment (Klint, 1993). It is called a *meta-environment* since it supports editing, type-checking, debugging and testing of specifications. Moreover, it supports the generation and testing of programming environments for the language described by a specification. We refer to Klint (1995) for the use of the ASF+SDF Meta-Environment. Good introductions in the ASF+SDF formalism can be found in Visser (1997) and Moonen (1996a). The book Van Deursen *et al.* (1996) gives an introduction to ASF+SDF as language prototyping formalism.

1.2 Problem Description

Reuse of modules in the ASF+SDF formalism is difficult. The reason is twofold. First, we should observe that the formalism is many-sorted. As a consequence, several instances of the same data type (like lists of integers and lists of strings) may exist. Since the formalism does not provide a parameterization mechanism, these instances have to be implemented separately. In case of a list of integers, this yields an implementation in which the definition of integers and the definition of the data type list are tied together. Modules in an ASF+SDF specification are therefore often strongly coupled and have less internal cohesion, which makes them unsuitable for reuse.

ASF+SDF has free syntax. This often results in different syntax used for the same data type. Hence, to promote the reuse of a module, one should be able to adapt the syntax of the reusable module for a particular need. Because the ASF+SDF formalism does not provide a mechanism for the adaptation of syntax, free syntax becomes the second reason why reuse in ASF+SDF is difficult.

In this thesis we try to find an answer for two questions: 1) Is it possible to introduce a mechanism for parameterization and for the adaptation of syntax without extending the ASF+SDF formalism? 2) Is it possible to define (complex) ASF+SDF modules that are reusable in many different projects.

The first question is the subject of Part I of this thesis. In the corresponding chapters we will develop a renaming mechanism as an external tool. This tool is defined to support renaming of syntax as well as parameterization. Part II of this thesis answers the second question. We will introduce the notions of parameterized modules and parameterized data types in ASF+SDF and we will develop a framework for the specification of hierarchical data types. As a case study we will develop part of a library and show its practical use. At the end of Part I and II a concluding chapter discusses the results, possible improvements and future work.

1.3 Related Work

In the past years several people have concentrated on the idea of a parameterization mechanism for ASF+SDF by means of renaming. We mention the renaming mechanism introduced in Walters (1994), that defines renaming on the AsFix representation of a module. Another renaming tool, that defines renaming on the textual representation of ASF+SDF modules is described in Visser (1992). Unfortunately, both tools have turned out not to be satisfactionary for use in practice.

The syntax definition formalism SDF2 (Visser, 1997) is a generalization of SDF and introduces renaming and parameterization. Despite these features, the formalism is not used in practice because no implementation of the formalism yet exists.

Despite the lack of a proper parameterization mechanism, several research projects were dedicated to the development of reusable modules for ASF+SDF. We mention the specification of Integer and Real arithmetic in Van Deursen (1991), the specification of data types based on four valued logic in Bergstra and Sellink (1996) and the library of ATerms (Van den Brand *et al.*, 1997c).

Although parameterization is a good mechanism for reuse in first order algebraic specification formalisms, considerable research has concentrated on other techniques for reuse in algebraic specifications. We mention the use of higher-order techniques in algebraic specifications (Walters and Kamperman, 1993; Heering, 1992). Another approach, called *Multi-Level Algebraic Specification* is described in Visser (1997). Multi-Level algebraic specification formalisms extend first-order formalisms with polymorphism and higher order functions.

1.4 Acknowledgments

First of all I want to thank Eelco Visser, who inspired and supported me during the whole project. His ideas and answers always helped a lot. The initial discussions with Paul Klint and Mark van den Brand resulted in a clear definition of this research project. Their comments on early versions of the text have resulted in several improvements. Many thanks to my colleague students and friends at the Programming Research Group of the University of Amsterdam for all coffee breaks, lunches and the like. My *type writers*: Jorien Kaper, Ramin Monajemi and my brothers Niels and Joost did a great job during the last phase of the project when my arm refused further typing. Not to forget my parents, for their great, never decreasing support.

Part I

A Renaming Tool for ASF+SDF

Chapter 2

Renaming

Renaming is a mechanism that promotes reuse of modules. In many formalisms (like Bergstra *et al.* (1989)), renaming serves two purposes: 1) The adaptation of the syntax of a module for a particular need and 2) resolving *name clashes*. A name clash occurs for example, when two modules both declaring a sort with the same name, are included in a single specification. Such a name clash can be resolved by uniquely renaming one of the sorts. A renaming $A \Rightarrow B$ (stating that A should be renamed to B) applied to a module, results in the substitution of B for all occurrences of A.

In a *many-sorted* formalism several instances of the same data type (like lists of integers and lists of strings) may exist. Without a proper reusing mechanism these data types have to be implemented separately. For example, in case of a list of integers, the implementation of a list and its elements (integers) are tied together. To promote the reuse of modules, a formalism should provide a mechanism that enables the abstraction from particular instances during the implementation of modules. Many modular formalisms provide a parameterization mechanism for this abstraction. Parameterization enables the implementation of modules with formal parameters. Instantiation is used to substitute (replace) the formal parameters of the module by actual parameters.

In modular formalisms that promote reusability, renaming and parameterization are often provided as distinguishable language constructs. Renaming is used in such formalisms to modify the syntax. Instantiation on the other hand, is used to modify the semantics of the module. Because the instantiation of a module implies the replacement of (some of) its formal parameters by actual parameters, it can be implemented as renaming. For example, in the formalism SDF2 (Visser, 1997) the formal parameters $[\alpha]$ of a module $M[\alpha]$ are renamed by the actual parameters $[\beta]$ when the module is imported as $M[\beta]$.

In order to improve the reusability of modules in ASF+SDF (that does not provide renaming nor parameterization), we will develop a renaming mechanism in the next chapters. Since parameterization can be implemented as renaming, the renaming tool will also support the instantiation of modules. Including parameterization, the renaming tool that we will develop serves three purposes: *Modification of syntax*, *Avoiding name clashes* and *Parameterization*.

The mechanism defined in the next chapters operates on existing ASF+SDF modules and therefore does not require an adaptation of the formalism. As a consequence, it is not difficult to define renamings that result in incorrect modifications of the semantics of specifications. We will therefore give a rather informal definition of parameterization in §3.4 and consider renamings that violate this definition to be invalid renamings with undefined results.

The next chapters are organized as follows. We start in Chapter 3 by defining the syntax and semantics of renaming and the several anomalies that arise when renamings are applied to ASF+SDF modules. Next, we concentrate on the algebraic specification of module renaming in Chapter 4. In that chapter we explain how a renaming is applied to a single module. The complete specification of module renaming is included in Appendix A. Chapter 5 concentrates on the implementation of the complete renaming tool. It extends the renaming of a single module from Chapter 4 to the renaming of a module hierarchy. Finally, we give a short discussion about the results, possible improvements and extensions of the renaming tool in Chapter 6.

Chapter 3

Syntax and Semantics of Renaming

3.1 Introduction

In this chapter a renaming mechanism for ASF+SDF is introduced. Since SDF2, the successor of SDF, provides a renaming mechanism, compatibility with SDF2 forms an important design goal. Hence, the syntax and semantics of renaming are defined according to SDF2 whenever possible.

In general we consider a renaming to be a substitution. That is, when $\mathcal{A}$ has to be renamed to $\mathcal{B}$ in some module M, we substitute $\mathcal{B}$ for all occurrences of $\mathcal{A}$ in M. However, the number of language constructs makes the renaming of an ASF+SDF module more complicated. As stated in Visser (1997), it would be desirable that renamings preserve well-formedness, i.e., if a tree T is well-formed under some grammar $\mathcal{G}$, it should also be well-formed when renamed with some renaming ρ. That is, we want that

$$\mathcal{G} \vdash T \Rightarrow \rho(\mathcal{G}) \vdash \rho(T)$$

This well-formedness cannot be preserved in all cases however. Hence, we have to define how renamings can be defined that preserve well-formedness and when well-formedness is violated.

This chapter forms the basis for the implementation of a renaming tool in Chapter 4 and Chapter 5. First, we introduce the syntax used to define renamings on ASF+SDF modules. This syntax is very close to the syntax of SDF2 renamings, however complete compatibility could not be achieved (see §3.2). Next, we give some definitions of ASF+SDF modules and their internal structure. Then we introduce conventions and definitions that are required for a good understanding of renaming in general and instantiation in particular. In §3.6 and §3.7 we concentrate on the renaming of specific language constructs and warn for renamings that violate the well-formedness. In §3.8 the parameterization mechanism is introduced. Name clashes are described in §3.9 and

§3.10. Finally, we concentrate on the renaming of a hierarchy of modules and the persistent storage of renaming results in §3.11.

3.2 Syntax

This section will introduce the syntax of the renaming mechanism for ASF+SDF. Since the mechanism is based on renaming as defined in SDF2, we will first explain renamings in that formalism shortly. Unfortunately, it has turned out to be unavoidable to deviate from SDF2 at some points. Hence, we will mention briefly why and how we have to deviate from SDF2 before we give the syntax of the renaming mechanism for ASF+SDF. The section concludes with an example to demonstrate the use of renaming in ASF+SDF.

SDF2 is (like ASF+SDF) a modular formalism. Each module can use sorts and functions defined in other modules by *importing* these modules. Renaming is implemented in SDF2 as part of this import mechanism. For each imported module an optional renaming can be specified. The following example shows the use of the renaming mechanism in SDF2.

Example 3.2.1 (SDF2 Renaming) The following module defines the syntax of tables. A Table is defined as a sequence of key/value pairs between square brackets.

```
module Tables
exports
   sorts Key Value Pair Table
   context-free syntax
      "(" Key "," Value ")" -> Pair
              "[" Pair* "]" -> Table
          lookup( Table, Key ) -> Value
```

Using this module we can generate several other modules by specifying different renamings. To define an array for instance, we can define the module below that defines a renaming on the module `Tables`.

```
module Arrays
imports Naturals
        Table [   Key => Nat
                Table => Array
                lookup( Table, Key ) -> Value =>
                   Array "[" Nat "]" -> Value]
```

This renaming replaces the sorts `Key` and `Table` in the module `Tables` by `Nat` and `Array` respectively. Moreover, the syntax of the function `lookup` is changed to obtain the syntax that is normally associated with arrays. The example assumes that the sort `Nat` is defined in the module `Naturals`. □

Let us now briefly describe why we have to deviate from SDF2 renaming. Due to limitations of the ASF+SDF formalism and because we do not want to extend the syntax of the formalism, we have to deal with three difficulties: 1) In SDF2 a renaming can be expressed as part of the import mechanism. This cannot be done in ASF+SDF since the formalism does not provide renaming. Consequently, no syntax for the specification of renamings is defined. 2) In SDF2 renaming is applied dynamically during normalization (i.e., renamings are applied when each import is replaced by the body it refers to). Therefore, the result of renaming has never to be made persistent. In ASF+SDF however, renaming is defined outside the formalism and can therefore not be part of the normalization process. As a consequence, a renaming has to be applied before a specification is normalized and the result has to be made persistent. 3) The renaming in the example above results in the declaration of the sort `Nat` twice, once in the module `Naturals` and once in the renamed module `Tables` (since `Key` has been renamed to `Nat`). While such "*instantiations*" are valid in SDF2 (in which double declarations are merged), they result in ambiguous specifications in ASF+SDF (since it does not merge double declarations).

Summarizing: in order to introduce renaming in ASF+SDF we require new syntax (outside the formalism) to define a renaming. Furthermore, a renaming has to be applied before a specification is normalized and the result has to be made persistent. Finally, the mechanism should avoid the introduction of double declarations.

We have defined a renaming mechanism that satisfies all these requirements by introducing "*Renaming Modules*" and renaming with an optional "*Origin Specification*".

A *Renaming Module* is a module that contains the definition of a single renaming. A renaming module M (with `M.ren` as corresponding file name) defines how the ASF+SDF module M should be constructed by renaming. To simplify the renaming mechanism and to maximize the reusability of renamed modules, we have limited renaming modules to define a renaming on a single module only.

A renaming with *Origin Specification* is a renaming that specifies in which module a sort or production is defined. Such a renaming will remove the declaration from the module that is being renamed and import the module (i.e., the origin specification) that contains the desired declaration. Origin specifications are used to avoid double declarations during instantiation. In Example 3.2.2 the use of an origin specification is demonstrated in order to instantiate the module `Tables` from the previous example unambiguously.

After describing why we have to deviate from the syntax of SDF2 renaming, we will now introduce the syntax of renaming in ASF+SDF and show precisely how it deviates from SDF2.

In accordance with SDF2 we will define a renaming `A => B` to be either a symbol or production renaming that renames A to B. A renaming `A => B <M>` forms an extension to the syntax of SDF2 and it is used to specify a renaming

imports Sdf-syntax$^{A.1}$
exports
  sorts Import Renaming Renamings Symbol Production
  context-free syntax

Sdf-SortId	$\rightarrow$ Symbol
Sdf-SortId Sdf-Iterator	$\rightarrow$ Symbol
"{" Sdf-SortId Sdf-Literal "}" Sdf-Iterator	$\rightarrow$ Symbol
Sdf-LexicalFunction	$\rightarrow$ Production
Sdf-CfFunction	$\rightarrow$ Production
Sdf-Id Renamings	$\rightarrow$ Import
"[" Renaming$*$ "]"	$\rightarrow$ Renamings
Symbol "$\Rightarrow$" Symbol	$\rightarrow$ Renaming
Symbol "$\Rightarrow$" Symbol "<" Sdf-Id ">"	$\rightarrow$ Renaming
Production "$\Rightarrow$" Production	$\rightarrow$ Renaming
Production "$\Rightarrow$" Production "<" Sdf-Id ">"	$\rightarrow$ Renaming

Figure 3.1: Syntax of Renaming in ASF+SDF

with origin specification. The complete syntax of renaming in ASF+SDF is shown in Figure 3.1.

Example 3.2.2 (ASF+SDF Renaming) The renaming from Example 3.2.1 can be specified according to the syntax of Figure 3.1 as follows:

```
Tables [  Key => Nat <Naturals>
        Table => Array
        lookup( Table, Key ) -> Value =>
        Array "[" Nat "]" -> Value]
```

Due to the origin specification `Naturals`, a double sort declaration of `Nat` in the renamed module `Tables` is prevented. Hence, this renaming will result in an unambiguous specification. The renaming applied to the module `Tables` yields the following module:

```
imports Naturals
exports
   sorts Value Pair Array
context-free syntax
   "(" Nat "," Value ")" -> Pair
   "[" Pair* "]"         -> Array
   Array "[" Nat "]"     -> Value
```

Observe that in this module only three sorts are declared (due to the origin specification the sort declaration `Key` has been removed). Furthermore, observe that the module `Naturals` is imported as another result of the origin specification. □

3.3 Modules and Signatures

Before we can define the semantics of renaming, we need notation in order to be able to describe how a renaming will affect the different parts of a module. This section will therefore describe the structure of ASF+SDF modules briefly and introduce operators for the retrieval of the several parts of a module.

In ASF+SDF a module consists of two parts: a part defining syntax rules and an optional part defining semantic rules. The first part consists of a number of sections, that together form the signature of a module (i.e., a pair (S, Ω) of sets of which the elements are sorts and operations respectively (Loeckx *et al.*, 1996)). We distinguish three types of sections that form the syntactical part of a module which we will describe shortly. First, a module may define import sections. An import section contains a set of names of modules that are imported. The complete set of names defined in all import sections of a module M is denoted by Imports(M). These modules together form the *imported signature* (Imp(M)) of a module. Next, a module consists of zero or more export sections, forming the *exported signature* (Exp(M)) of the module. Finally, the module may define a set of hidden sections, that form the *hidden signature* (denoted by Hid(M)).

Hidden and exported sections may contain variable declarations. We will use V(M) to denote all variables that are declared in a module. Furthermore, we use E(M) to denote the set of equations (i.e., the semantical part) of a module.

In ASF+SDF the exported signatures of imported modules are automatically exported. Hence, the set of operations and sorts visible outside a module, depends not only on the exported signature, but on the imported signature as well. This is depicted in the definition of the visible signature below:

Definition 3.3.1 According to the definition in Bergstra (1987) we define the *visible signature* of a module M (denoted by Vis(M)) to be the union of the imported and exported signatures: $\mathrm{Imp}(M) \cup \mathrm{Exp}(M)$. The visible signature corresponds to the operations and the sorts visible outside a module.

Furthermore, we define the *internal signature* (Int(M)) to denote all sorts and operations visible within the module M. This corresponds to the union of the visible and hidden signature: $\mathrm{Vis}(M) \cup \mathrm{Hid}(M)$. □

3.4 Definitions and Conventions

The renaming mechanism that we will define introduces parameterization in ASF+SDF. Although the notion of a *parameter* is required for parameterization, we will not extend the ASF+SDF formalism. Instead, we will introduce parameters in ASF+SDF by means of a convention. This convention and some related definitions are the subject of this section. The use of parameterization and instantiation is described in detail in §3.8 and §8.2.

In formalisms with parameterization a *formal parameter* is used as *place holder*. During instantiation formal parameters are replaced by *actual parameters*. The use of formal parameters enables the correct definition of a specification even before instantiation. Since a formal parameter is replaced during instantiation no semantics should be assigned to it.

In order to support parameterization in ASF+SDF we should observe the desire to use both *parameter sorts* and *parameter functions*. We will use the term *undefined* for sorts and productions without semantics. A sort is undefined when it does not contain any constants (i.e., the sort does not occur as target sort of a function with zero arity). We will define a (non-constructor) function to be undefined when no equations are defined for it. Observe that for constructor functions also no equations are defined. We can therefore not distinguish between undefined and constructor functions from the context of a module. Using these observations, we will introduce the notion of a parameter in ASF+SDF by the following convention:

Convention 3.4.1 (Sort and Function Parameter) By convention we define a *parameter sort* to be an *undefined* sort that is *exported*. Likewise, we define a *parameter function* to be an *exported* and *undefined* function. Because constructor functions and parameter functions are indistinguishable, we will from now on indicate explicitly which functions are parameter functions. Instead of *parameter* sorts and functions the terms *generic* sorts and functions are often used. □

The use of parameters in an ASF+SDF module is demonstrated in the example below.

Example 3.4.2 (Sort and Function Parameters) The following module defines two parameters: a parameter sort 'X' and a parameter function 'f'.

```
exports
   sorts X Y
   context-free syntax
      "a"    -> Y

      f( X ) -> X
      g( X ) -> X
   variables
      "x" -> X
equations
[01] g( x ) = f( x )
```

Observe that the sort 'Y' is *defined* because it contains the constant 'a'. The sort 'Y' is therefore not a parameter function. Likewise, the function 'g' is not a parameter function since an equation is defined for it. □

3.5 Renaming Semantics

In this section we introduce the semantics of renaming. We describe how a renaming is applied to a module and discuss some properties that follow from our definition of renaming. The remaining sections in this chapter will discuss the renaming of the several grammar constructs of ASF+SDF modules in more detail.

A renaming replaces all symbols and productions in a module for which a renaming has been defined. Symbols and productions are renamed by recursively applying the renaming to all grammar constructs that form a module. For symbols and productions a renaming is found by looking up the symbol or production in the list of renamings. If such a renaming exists in the list, the symbol or production is replaced by the value according to the renaming. When a renaming does not exist, renaming is applied recursively to the sub-parts of the symbol or production. To each symbol or production at most one renaming is applied. When more then one renamings exist, the first renaming is applied.

After this informal definition of renaming, let us now discuss some properties that follow directly from this definition. First observe that a renaming is applied to a production *before* a renaming is applied to its argument and target sorts, since renaming is applied recursively according to the structure of a module. Symbol renamings will therefore not affect a production that has already been renamed. Thus, for some productions f and g, arbitrary symbols $S_0, \ldots, S_n$ and sorts A, E, B let:

$$\begin{array}{rcl} f(S_0, \ldots, A, \ldots, S_n) \rightarrow B & \in & M \\ A \Rightarrow E & \in & \rho \\ f(S_0, \ldots, A, \ldots, S_n) \rightarrow B \Rightarrow g(S_0, \ldots, A, \ldots, S_n) \rightarrow B & \in & \rho \end{array}$$

then

$$g(S_0, \ldots, A, \ldots, S_n) \rightarrow B \in \rho(M)$$

As a consequence, to rename the syntax of a production *and* its sorts both have to be specified in a single renaming. Likewise, to modify the syntax *and* type of list constructs (see the next section), these two modifications have to be specified in a single renaming as well.

Secondly, since the first renaming is selected in the case that more renamings are defined for a symbol or production, the order of such *overlapping* renamings becomes relevant. For two *non-overlapping* renamings this order is irrelevant. That is, for two non-overlapping renamings ρ_1 and ρ_2 and a module M, the following holds:

$$[\rho_1\ \rho_2](M) = [\rho_2\ \rho_1](M)$$

Finally, we should observe that a renaming $[B \Rightarrow C]$ applied to a renaming $[A \Rightarrow B]$ (i.e., the *external composition* of two renamings is not equivalent to the *internal composition* $[A \Rightarrow B\ B \Rightarrow C]$. Because at most one renaming is applied to a symbol or production, a symbol or production that has been renamed to B (according to $A \Rightarrow B$) cannot transitively be renamed to C (according to $B \Rightarrow C$) using internal composition of renamings.

After the definition of renamings and the discussion of several of its properties, the next sections concentrate on the renaming of several constructs of ASF+SDF modules. First, symbol and production renamings are explained. Next, name clashes, the renaming of hidden grammars and module hierarchies are discussed.

3.6 Symbol Renamings

A *symbol renaming* corresponds to the renaming of a sort or list construct. The renaming of a sort according to a renaming $A \Rightarrow B$ implies the replacement of occurrences of A by B. The renaming of list constructs is more complicated and is the subject of this section.

In ASF+SDF two list constructs are defined: Lists *with* and lists *without* separators:

$S*$	defines a list of *zero* or more repetitions of S.
$\{\ S\ ","\}+$	defines a list of *one* or more repetitions of S separated by ','.

From these examples we can see that both list constructs have syntax (i.e., the optional separator), type information (i.e., the type of its elements) and semantics (i.e., the iterator '+' or '*').

We will define the renaming of a list construct according to the three purposes for which renaming is defined (modification of syntax, avoiding name clashes and parameterization (see §2)). Therefore, we support the modification of the syntax of list constructs. That is, a separator may be introduced, modified or removed by renaming. However, renaming the iterator of a list is not supported since it implies a modification of the semantics. This differs from SDF2 where a *Regular Expression* (which is a generalization of an iterator in SDF2) can be renamed to another regular expression. Because it is not clear what the exact semantics of such renamings is, this type of renamings will probably be dropped in a future definition of SDF2.

To summarize: the renaming $[A* \Rightarrow A+]$ is not supported because it modifies the iterator (and hence the semantics). The renaming $[A* \Rightarrow \{A\ ","\}*]$ on the other hand, is allowed because it only modifies the syntax.

3.7 Production Renamings

Production renamings are used to modify the syntax and the type of productions. The need to include symbol renamings in production renamings to modify the type of productions was motivated in §3.5.

Renaming a production involves two issues that need further consideration. First, we should consider what happens when a production renaming includes the renaming of one of its argument or target sorts. Secondly, we should consider how a renaming modifies the attributes of a production.

3.7.1 Production Renamings with Symbol Renaming

Because a production consists of syntax, a target sort and zero or more argument sorts, the renaming of a production may result in a modification of syntax and sorts.

One has to be careful when syntax and sorts are changed by a single renaming. Such renamings can result in erroneous specifications, as is demonstrated by the following example:

Example 3.7.1 Consider the function `eq` defined on integers:

```
context-free syntax
   eq "(" Int "," Int ")" -> Bool
variables
   I[0-9]* -> Int
equations
[01] eq( I1, I2 ) = true  when I1  = I2
[02] eq( I1, I2 ) = false when I1 != I2
```

Applying the following renaming that defines the function `eq` on strings instead of integers will result in an erroneous specification.

```
M [eq "(" Int "," Int ")" -> Bool =>
   eq "(" Str "," Str ")" -> Bool
  ]
```

Since this renaming will not modify the type of the variables, they will have the wrong type after the renaming has been applied:

```
context-free syntax
   eq "(" Str "," Str ")" -> Bool
variables
   I[0-9]* -> Int
equations
[01] eq( I1, I2 ) = true  when I1  = I2
[02] eq( I1, I2 ) = false when I1 != I2
```

□

To avoid this kind of problems, we introduce the following requirement for production renamings:

Requirement 3.7.2 Let f, g denote arbitrary functions, $S_0 \dots S_n$ symbols, A, B, C and D sorts and a renaming ρ, then we require:
if $f(S_0, \dots, A, \dots S_n) \to B \Rightarrow g(S_0, \dots, C, \dots S_n) \to D \in \rho$ then also $A \Rightarrow C,\ B \Rightarrow D \in \rho$. □

Thus, whenever a sort is renamed by a production renaming, the corresponding sort renaming should be defined separately as well.

Of course, the renaming tool can be defined to automatically include the renamings $A \Rightarrow C$ and $B \Rightarrow D$. To remain compatible with the SDF2 formalism however, we have chosen not to implement this. The automatic addition of symbol renamings may be included in a future definition of SDF2.

The renaming mechanism does not forbid production renamings that introduce or change the order of existing arguments. Such renamings are considered invalid, because they will modify the semantics of a specification and are therefore ignored in this text.

3.7.2 Production Renamings with Attributes

Specifying the attributes of a production in a renaming is optional. For example, the production `Nat "+" Nat -> Nat {left}` can be renamed leaving the attributes unchanged using the renaming:

```
[Nat "+" Nat -> Nat => Nat "++" Nat -> Nat]
```

But in order to adapt the attributes of a production, one has to specify the production completely:

```
[ Nat "+" Nat -> Nat {left} => Nat "++" Nat -> Nat {right}]
```

Some care has to be taken when modifying the attributes, since it can change the semantics of a specification. When, for example, the associativity of a production is changed from left to right, the semantics of a specification might be affected significantly. Modifying the attributes of a production is therefore dangerous and one should be very careful with it. Despite its possible adaptation of semantics, the modification of attributes becomes indispensable to disambiguate a production that is renamed from prefix to infix notation.

For example, to rename the prefix function `plus(Nat,Nat)` $\to$ `Nat` to infix correctly, an attribute is required to disambiguate the production. The renaming should therefore look like:

```
[ plus(Nat, Nat) -> Nat => Nat "+" Nat -> Nat {left} ]
```

In order to restrict a renaming to the three goals that we designed renaming for (solving name clashes, modification of syntax and instantiation), we suggest that attributes should be modified only to disambiguate a function that is renamed from prefix to infix notation and vice versa. These modifications require the introduction or the removal of an attribute respectively. That is, when changing a function from prefix to infix notation, the attributes `left`, `right` or `non-assoc` may be introduced. When prefix notation is obtained these attributes may be removed. all other modifications of attributes are discouraged.

Obviously, the restricted use of attribute renaming could have been forced by the renaming tool itself, but once again, to remain compatible with SDF2 we have decided not to restrict renaming in this way.

Besides associativity, *priorities* are used to disambiguate a specification. By renaming it is not possible however, to change the priorities of functions.

Example 3.7.3 Consider the specification consisting of the two functions `plus` (for addition) and `mul` (for multiplication), both in prefix notation:

```
plus(Int, Int) -> Int
 mul(Int, Int) -> Int
```

Next, consider the renaming that changes the syntax of these functions to obtain infix notation:

```
[plus(Int, Int) -> Int => Int "+" Int -> Int {left}
  mul(Int, Int) -> Int => Int "*" Int -> Int {left}]
```

This renaming yields an ambiguous specification since the resulting specification does not provide priorities. For a term like `1 + 2 * 3` this results in two possible calculations (either `(1+2)*3` or `1+(2*3)`) with different outcomes (9 and 7 respectively). □

In general, one has to be careful with modifying the syntax of a function from prefix to infix notation when priorities have not been defined.

3.8 Instantiation

In a parameterized module the parameter plays the role of "*place holder*", i.e., the parameters are *used* rather then *defined* or *redefined*. During instantiation the formal parameters of a module are replaced by their corresponding actual parameters.

In ASF+SDF parameterized modules do not exist, therefore we have introduced the rather informal definition of parameter and parameterized modules in §3.4. In that section a parameter is defined as either an undefined sort or production. With this requirement a parameter is simulated by a sort or production declaration, since it has no semantics assigned and its use is therefore restricted as place holder only.

Now consider the parameterized module `List` specifying the data type `List` on a parameter sort `Item`:

```
exports
   sorts List Item

   context-free syntax
      Item* -> List
      "insert" "(" List "," Item ")" -> List
```

Suppose we want to construct a list of integers by instantiating this module with the sort `Int`. Suppose furthermore that the sort `Int` and operations on integers are defined in the module `Integers`.

How should we instantiate the module `List`? Just replacing all occurrences of `Item` by `Int` according to the renaming `[Item => Int]` results in the double declaration of the sort `Int` (once in the renamed module `List`, and once in the module `Integers`):

```
exports
   sorts List Int

   context-free syntax
      Int* -> List
      "insert" "(" List "," Int ")" -> List
```

This yields an erroneous specification when both modules are contained in the module hierarchy of a single specification, since ASF+SDF does not merge two equivalent sort (or production) declarations.

Of course, the intended meaning of the instantiation is to obtain a list with elements of type `Int`, defined in the module `Integers`. As suggested in Bergstra (1987), the instantiation of a module (or parameter binding) should perform the following tasks:

1. Add the module defining the actual parameter (`Integers` in the above example) to the list of imported modules.
2. Remove the sort (or production) declaration of the formal parameter (`Elem` in the example).
3. Replace all occurrences of the formal parameters by their corresponding actual parameters (this corresponds to the replacement of all occurrences of `Elem` by `Int` in the example).

Since the module in which an actual parameter is defined cannot be determined automatically, we have introduced the notion of "*renaming with origin specification*" to implement instantiation of modules. The actual parameter does not have to be defined in the module that is imported itself, but may be defined in some module that is imported directly or indirectly. To instantiate a formal parameter A by B, defined in a module hierarchy rooted at module M, one uses the renaming `[A => B <M>]`. Using renaming with origin specification, we can instantiate the list module correctly by applying the renaming `[Item => Int <Integers>]` to the module List. This results in the following module:

```
imports Integers
exports
  sorts List

  context-free syntax
    Int* -> List
    "insert" "(" List "," Int ")" -> List
```

The need to specify an originating module for the correct instantiation of a module comes from the limitation of ASF+SDF, which is not able to merge two equal sort or production declarations. In SDF2 however, such declarations are merged and as a consequence, the removal of formal declarations and the import of an originating module is superfluous. Hence, renaming with origin specification is the only (unavoidable) renaming construct that resulted in a deviation from the syntax of renaming as defined for SDF2. Observe that the module hierarchy of a specification can be changed by a renaming with origin specification. This is an important semantical deviation from the renaming as defined in SDF2, which has consequences for the development of compatible module hierarchies (see §8.2).

3.9 Name Clashes

A name clash occurs when two initial different sorts or productions are made equal. A renaming $[A \Rightarrow B]$ applied to a module M thus yields a name clash when $A, B \in \mathrm{Int}(M)$ and $A \neq B$, where $\mathrm{Int}(M)$ is the internal signature of M (see §3.3).

In Bergstra *et al.* (1990) a *permutative* renaming is defined: if a is renamed to b, then b is simultaneously renamed to a. This mechanism assures that different names will never be made equal by renaming. In other words, using this permutative renaming will prevent name clashes. Because avoiding name clashes is not the only purpose of our renaming tool, the permutative strategy does not suffice and will even introduce incorrect results, as is demonstrated in Example 3.9.1.

Example 3.9.1 (Permutative Renaming) Consider a generic list defined on items and a function `length` that returns the length of the list:

```
Item* -> List
length "(" List ")" -> Int
```

Suppose we want to instantiate this definition to create a list of Integers using the renaming `[Item => Int]`. Applying a permutative renaming to the above specification will yield the following result:

```
Int* -> List
length "(" List ")" -> Item
```

Using a permutative mechanism, the renaming `[Item => Int]` will replace the target sort of the function `length` by `Item` to solve the name clash introduced by the renaming. Obviously this is not what we expect from the instantiation of a module. □

To be able to instantiate a module correctly we will not prevent name clashes resulting from a renaming automatically. Our renaming mechanism will therefore not be permutative. Observe however that the behavior of a permutative renaming `[A => B]` can be simulated by specifying the renaming `[A => B B => A]`.

Despite name clashes that are harmless, some name clashes may result in errornous specifications and have to be prevented. Since such name clashes have to be prevented by hand, let us describe the situations in which they can occur and how to prevent them.

We can distinguish two possible types of name clashes: the first type of name clashes occur as a result of normal renamings (i.e., renamings without origin specifications). Since these renamings do not instantiate a module, they should not change the semantics of a module but only its syntax. To solve these name clashes such renamings need to be modified slightly to introduce new syntax.

The second type of name clashes occur as a result of renamings with origin specifications. These renamings do instantiate a module and as a result, they modify the semantics of a module. Instantiation of a module M according to the renaming `[A => B <N>]` implies the import of module N in M.

Suppose that $M \cup N$ is un-ambiguous (otherwise we construct a new module M' by renaming each double declared sort or production in $M \cup N$ uniquely). Whenever the instantiation `[A => B <N>]` introduces a name clash we should note the following:

- According to the discussion of instantiation in §3.8, the declaration of A is removed from M (see also §3.11). Hence, Both N and M will share the same sort or production declaration B.
- The sort or production B is not affected by the instantiation. In case of the instantiation of a parameter sort no new elements are added to the sort (since the sort A is undefined by Convention 3.4.1). In case of the instantiation of a function parameter no new equations are added (since by Convention 3.4.1 no equations exist for the production A).

Hence, the name clash is harmless.

To summarize: in Bergstra *et al.* (1990) all name clashes introduced by renaming are undesired and avoided. The example above however, showed that for instantiation such name clashes should not be prevented. When modules are specified according to the conventions mentioned in §3.4, these name clashes are harmless. Because these conventions are not enforced, it is possible to introduce name clashes that are harmful. These are considered as invalid renamings though and we will not pay any attention to them in the rest of this text.

3.10 Renaming of Hidden Grammars

Since a hidden grammar is by definition not visible outside its corresponding module, one could argue about the semantics of the application of a renaming to such a grammar. One could follow one of two strategies in order to handle renamings of hidden grammars.

No Renaming of Hidden Grammars. Although theoretically probably the most perfect solution, it introduces problems due to name clashes. Because renaming is not applied to hidden grammars, the user does not have any control of symbol and production names in these grammars. This can easily lead to name clashes when a renaming is applied to the corresponding module. We need to provide a naming scheme for hidden sorts and productions that prevents such name clashes. We can, for example, prefix each hidden sort and production by the module name in which it is defined. A hidden production `f( A ) -> B` defined in module M then becomes `M_f( A ) -> B`. Of course a naming scheme like this cannot prevent name clashes completely.

Applying renamings to hidden grammars. Not distinguishing renamings of sorts and productions defined in hidden or exported grammars enables the user to solve name clashes with hidden sorts and productions by hand. Moreover, this strategy is more easy to implement because both grammars can be handled in the same way. The renaming of a hidden sort or production does however introduce a problem when an origin specification has been specified. Because of an origin specification, an extra module is added to the set of imported modules. Since in ASF+SDF no hidden import mechanism exists, the imported module is automatically exported. As a result, the renamed sort (or production) f, that was originally hidden in a module M (i.e., $f \in \mathrm{Hid}(M)$), becomes visible now (i.e., $f \in \mathrm{Vis}(M)$). However, according to the conventions in §3.4, a renaming with origin specification is defined to be an instantiation of formal parameters. A formal parameter is either an *exported* sort or an *exported* production. Instantiating a *hidden* sort is therefore not defined. Moreover, because a parameter sort or production is undefined by definition, it makes no sense to hide such a parameter since it remains undefined.

Due to the simplicity of its implementation and the flexibility that it provides for the user to prevent name clashes, we have chosen to implement the second strategy. Thus, renaming of productions and sorts declared in hidden grammars is allowed, but its only purpose should be the prevention of name clashes.

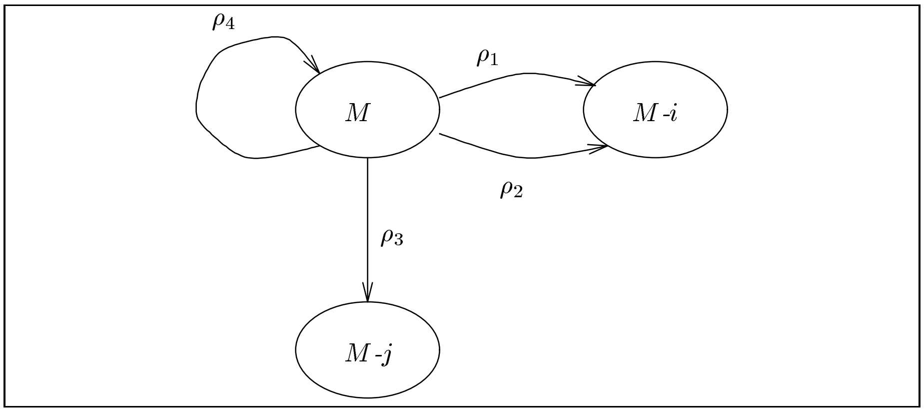


Figure 3.2: Module Identifiers can be shared to avoid redundancy.

3.11 Renaming a Module Hierarchy

When a renaming `[A => B]` is applied to a module M, we have to distinguish three different situations. These situations are exclusive, either one of them happens, but no combinations:

1. $A \notin \mathrm{Int}(M)$. The renaming does not affect M or one of its imported modules. This situation corresponds to $\forall_{m \in (\mathrm{Imports}(M) \cup M)} \langle [A \Rightarrow B](m) = m \rangle$

2. $A \in \mathrm{Exp}(M) \cup \mathrm{Hid}(M)$. That is, A is defined in the module M itself, not in one of its imported modules. In this case we just apply the renaming and we are done.

3. $A \in \mathrm{Imp}(M)$. A is declared in some module N in the module hierarchy of M.

The first two situations are trivial, the third one is the subject of this section.

Being an operation external to the ASF+SDF system itself, the result of the application of a renaming to a module should be made persistent. That is, the application of a renaming to a module M should yield a module M' containing the result of the renaming. What name should we give to this renamed module? Obviously a module M, a renaming ρ and the resulting module M' are related. Consider however the sets of renamings $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, the renamings $\rho = \mathcal{A} \cup \mathcal{B}$ and $\rho' = \mathcal{A} \cup \mathcal{C}$ and the modules M and M' with the following properties:

$$
\begin{aligned}
\mathcal{A}(M) &= M' \\
\mathcal{B}(M) &= M \\
\mathcal{C}(M) &= M
\end{aligned}
$$

Then clearly $\rho(M) = \rho'(M)$. To avoid redundancy the result of these renamings should be stored only once. In order to allow a renamed module to be related to more then one renaming, we introduce a naming scheme that appends a unique *module identifier* (represented by a natural number) to the module name. Furthermore, we introduce the (finite) function $M \xrightarrow{\rho} M\text{-}i$ that maps a module M and a renaming ρ to the corresponding renamed module $M\text{-}i$, satisfying the following property:

Property 3.11.1 (Sharing of Module Identifiers)

$$\left.\begin{array}{l} M \xrightarrow{\rho} M\text{-}i \\ M \xrightarrow{\rho'} M\text{-}j \\ M\text{-}i \equiv M\text{-}j \end{array}\right\} \Rightarrow i = j$$

□

In Figure 3.2 the use of identifiers and the mappings between module renamings and result modules is depicted.

For the implementation of this mapping mechanism we use a table. The table contains for each module M and the renaming ρ that has been applied to the module an identifier id, indicating that the result of $\rho(M)$ is stored in the module $M\text{-}id$. When the result of two different renamings has yielded two equivalent modules, their corresponding entries in the table will share the same module identifier. We introduce the *zero* identifier for the case that no identifier has been attached to M and $\rho(M) = M$. Without this convention we cannot add the information according to the mapping $M \xrightarrow{\rho} M$ to the module table.

Example 3.11.2 (Module Table) Consider the following table:

M	ρ_1	i
M	ρ_2	i
M	ρ_3	j
M	ρ_4	0

This table states that the result of renaming ρ_1 applied to module M is stored in module $M\text{-}i$. The application of ρ_2 to the module has yielded an equivalent module. Consequently, both renamings share the same module $M\text{-}i$. The application of ρ_3 to M has resulted in the module $M\text{-}j$ ($i \neq j$). Finally, the table shows that the renaming ρ_4 did not affect the module M, hence the zero identifier in the table. □

Whenever a module M is contained in a module hierarchy of a specification (but is not the root module), then M is imported by some module N. When the module is modified by a renaming, the result of the renaming is stored in $M\text{-}i$. Obviously, N should import the module $M\text{-}i$ after the renaming has been applied to the module hierarchy. The import section of N should therefore be updated to import the module $M\text{-}i$ rather than M. As a consequence,

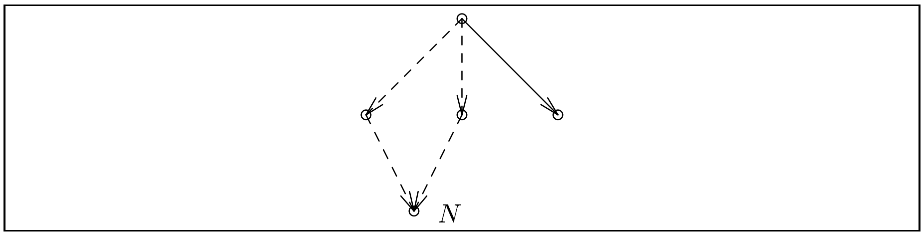


Figure 3.3: All modules contained in a path of dashed arcs should be updated when module N has been modified by a renaming.

N becomes affected by the renaming too, resulting in a module N-j. Thus, whenever some module is affected by a renaming, all modules that directly or indirectly import the affected module become affected too (see Figure 3.3).

The information in the module table can be updated to include all correct module mappings prior to the renaming of a single module. When a renaming is applied to a single module, this up-to-date information from the module table will be used to correctly update the names of imported modules.

Chapter 4

Specification of Module Renaming

4.1 Introduction

After the semantics has been defined in the previous chapter, this chapter concentrates on the specification of a renaming mechanism accordingly. The specification defines single module renaming only. Recursive renamings and renaming of a complete module hierarchy are described in the next chapter. The complete specification can be found in Appendix A.

The specification takes as input the AsFix representation of a module, the textual representation of the renaming (i.e., the contents of the renaming module) and a table containing module to file name mappings. The specification generates as output the renamed module represented in AsFix.

To understand how a renaming modifies an ASF+SDF module, some knowledge of ATerms and AsFix is required since renaming operates on the AsFix representation of ASF+SDF modules. Hence, this chapter starts with a brief introduction of ATerms in §4.2. The use of AsFix as representation for ASF+SDF modules is motivated in §4.3.

Since a renaming should only modify *matching* symbols and productions, we give a definition of matching renamings in §4.4. The remaining sections describe the renaming process in more detail. In §4.5 the global renaming process is described, §4.6 concentrates on the renaming of ATerms and the different renamings that have been defined for different language constructs. Finally, in §4.7 the renaming of symbol and production declarations according to renamings with origin specifications is described.

4.2 ATerms

This section contains a brief introduction of ATerms (short for Annotated Terms). Some knowledge of ATerms is required since it forms the base for the AsFix representation of ASF+SDF modules. While this section concentrates on the basics of ATerms only, the next section motivates the use of ATerms and AsFix as representation of ASF+SDF modules for renaming.

ATerms (Van den Brand *et al.*, 1997c) are defined as intermediate data representation. They are used to represent structured information that is to be exchanged between a heterogeneous collection of tools. ATerms are designed to accommodate the representation of all data that is to be exchanged. Furthermore, ATerms support the addition of information to terms in the form of *annotations* to the original ATerms.

ATerms are either literal strings (e.g., `"abc"`), numerals (integer numbers or floating point constants) or they are constructed using function application of the form $F(T_1, \ldots, T_n)$, list construction (i.e., $[T_1, \ldots, T_n]$) or annotation (i.e., $T\{T_1, \ldots, T_n\}$).

An advantage of ATerms is its support for higher-order functions. A simulation of higher-order functions in the setting of the first-order formalism ASF+SDF is possible because all operations on ATerms are expressed using a single fixed sort `ATerm`. A framework of higher-order functions is introduced by separating the name of a function (as literal string) and its application. A higher-order function will get the name of a function as argument. The user provides functions that form a mapping between the name of a function and its application. Within the framework of higher order functions these mappings are defined by means of predicates that determine whether some condition *holds* for an ATerm and functions that *apply* some operation to an ATerm. The use of higher-order functions is demonstrated by the example below:

Example 4.2.1 (Higher-Order Functions) Consider the function `is-bool` that determines whether the value of a term is either the boolean constant `true` or `false`:

```
[01] is-bool(T) = T == true \/ T == false
```

Next suppose that we want to determine whether some term in a list is a boolean. Instead of traversing the list ourselves, we will use the higher-order function `for-some` defined in the ATerm library. First, we define the predicate `holds` that forms the mapping between the name of the function (i.e., `"is-bool"`) and its application:

```
[02] holds( "is-bool", T ) = is-bool(T)
```

The higher-order function `for-some` can now be used as follows to search a list Tl of elements for the existence of a boolean value:

```
for-some("is-bool", Tl)
```

□

The ATerm library provides many higher-order functions for list operations and for the traversal and modification of trees. Among others, the availability of these functions formed a strong motivation for the use of AsFix (that is based on ATerms) as module representation for the application of renamings. This motivation is further discussed in the next section.

4.3 Module Representation

To rename a module we have to apply the renaming to some representation of the module. Since we cannot depend on the internal representation of the ASF+SDF system (because we do not have access to the abstract syntax tree of the module directly), we have to use a representation that provides enough information about the structure of a module and is easy to manipulate. The textual representation (this is the representation used by the ASF+SDF system to store modules) for example, lacks any type information and is therefore not appropriate.

The AsFix (Van den Brand *et al.*, 1997b) formalism does provide a good representation for modules. It does contain all information of a module (it contains type information for variables, white space, and all productions in equations contain links to their corresponding declarations). An AsFix module is represented internally using ATerms. It provides constructor functions for the representation of ASF+SDF modules. The internal representation (using ATerms) is hidden by appropriate access functions and predicates for the manipulation and inspection of AsFix modules.

Due to these access functions and predicates, renaming can almost completely be implemented using these functions. Although little knowledge is required of the internal representation using ATerms, precise knowledge of the structure of ASF+SDF modules is a prerequisite. To implement renaming using these access functions one has to unpack the module according to its structure until the single productions and symbols can be reached.

For example, in order to apply a symbol renaming to a sort declaration defined in an exported grammar of module M, the module has to be split in sections, grammars and sorts successively.

Using these functions for the implementation of renaming would imply writing many functions that have nothing to do with the renaming itself, but with unpacking a module and constructing it again after a renaming has been applied.

Besides this complexity, most of the structure of an ASF+SDF module is not important for renaming. To rename a sort S to S' for instance, we would just rename all occurrences of S. For renaming it is therefore irrelevant to which section the sort belongs, or whether it occurs in a function or in a variable declaration. Unfortunately, things are a bit complicated due to renamings with origin specifications. These renamings do require some knowledge of the structure of a module because the behavior depends on the context of a matching sort or production (i.e., being a declaration or not).

The observation that only little knowledge of the structure of a module is required and the availability of many higher order functions for the manipulation of ATerms, have made us decide to implement renaming on ATerms directly. AsFix functions are used only for the construction of new terms and for the destruction of terms in their corresponding components.

4.4 Matching

A renaming is applied to a term when a matching renaming can be found. A matching renaming $[A \Rightarrow B]$ is found for a term T whenever the following conditions hold:

1. T and A are *matching* terms.
2. The renaming $[A \Rightarrow B]$ is *valid* with respect to T.

To determine whether a renaming $[A \Rightarrow B]$ matches a term, we have to define when two symbols and two productions match:

- Two symbols T_1 and T_2 match when they are equivalent modulo white space.
- A symbol T_1 matches a list application T_2 (for example the list: $A; B; C$), when the symbol of T_2 (i.e., the type of the list) matches T_1.
- A production P_1 matches P_2 when all symbols of P_1 match the symbols of P_2 discarding any white space.

Matching alone is not sufficient to determine whether a matching renaming exists. There are some conditions that should hold depending on the term that is to be renamed. Whenever these conditions hold we say that a renaming $[A \Rightarrow B]$ is *valid* with respect to a term T. Depending on the term T the following conditions should hold for a renaming $[A \Rightarrow B]$:

- T is a production (i.e., its function symbol is `prod`). The attributes of T, A and B should match. That is, either no attributes are specified in the renaming or the attributes of T equals the attributes of A. This condition states that specifying the attributes in a production renaming is optional. However, when attributes are specified, these attributes and the attributes of a production declaration in the module should match exactly (see also §3.7.2).
- T is a list symbol (its function symbol is either `iter` or `iter-sep`). We require that the iterators of T and B match (see §3.6). This condition states that we can only change the syntax of a list, not its semantics (i.e., $A*$ may be renamed to $\{A \text{ "," }\}*$ but not to $A+$).

- T is a list (its function symbol is `list`). Here too, we only allow syntactical changes. Thus, the iterator of a list (i.e., `T[sym]`) should match the iterator of B (see §3.6).

For all other terms T only a match between T and A is required to determine whether a renaming $[A \Rightarrow B]$ can be applied to the term T.

4.5 Renaming

Renaming the AsFix representation of a module is done in a number of steps. The global process of renaming (the different steps involved and the order in which they are passed through) is specified in the following equation:

$$
[01] \quad \frac{\begin{array}{rcl} \text{asfix}(I) & = & \mathit{Tl}, \\ M[\text{abbrevs}] & = & \mathit{Abbrevs}, \\ \text{rename-aterm}(\mathit{Tl}, \mathit{Abbrevs}, M) & = & M_1, \\ \text{imports}(M_1, \mathit{MT}) & = & M_2, \\ \text{remove-empty-grammars}(M_2) & = & M_3 \end{array}}{\text{Ren}(I, M, \mathit{MT}) \ = \ M_3}
$$

The renaming I is first converted to AsFix. Representing both the module and the renaming in AsFix simplifies the matching and substitution of terms. An AsFix module contains an abbreviation section, the purpose of which is to limit the size of an AsFix module. The section maps numbers to production declarations. These numbers are used to link the functions used in equations to their corresponding declarations. To rename an equation correctly, the declarations corresponding to the functions in the equation are retrieved from the abbreviation table (M[abbrevs] = Abbrevs). These declarations are used to find matching renamings.

Next, the renaming is applied to the AsFix representation of the module by calling the function 'rename-aterm' (see §4.6). After the module has been renamed, modules according to renamings with origin specifications are added to the list of imported modules (see §4.7). Due to renamings with origin specifications (and the corresponding removal of sort and production declarations), exported or hidden grammars may have become empty. Such empty grammars are not allowed in ASF+SDF. Therefore, as a last step, the renamed module is filtered to remove empty grammars.

4.6 Renaming ATerms

An ASF+SDF module is represented in AsFix as a tree. Each node in the tree corresponds to a component of the module, its children correspond to subcomponents. In Figure 4.1 part of this tree structure is depicted. Renaming the AsFix representation of a module consists of a traversal of the AsFix tree and the replacement of all terms that match a renaming.

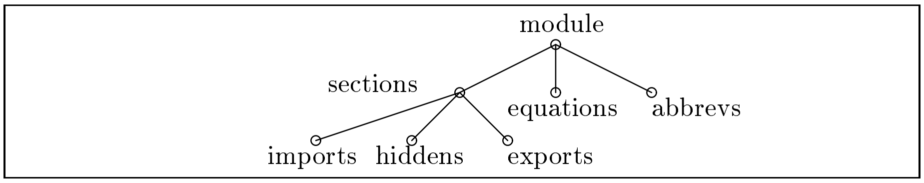


Figure 4.1: (Part of) the tree structure of an ASF+SDF module.

AsFix introduces constructors for the different components that form an ASF+SDF module. These constructors can be distinguished by their unique function symbols. Because only a subset of all different components can be renamed (because we only have symbol and production renamings), the function symbols of the AsFix terms are used to determine *if* a term is a candidate for renaming. Moreover, since renaming of each type of component might differ, the function symbols are also used to determine *how* a term should be renamed. Renaming of an ATerm tree can now be defined as follows:

$$
\begin{array}{ll}
[01] & \text{rename-aterm}(\mathit{Tl},\ \mathit{Abbrevs},\ T) \\
 & =\ \text{treplace-ann}(\texttt{"should-rename"}(\mathit{Tl},\ \mathit{Abbrevs}), \\
 & \qquad\qquad\qquad \texttt{"rename"}(\mathit{Tl},\ \mathit{Abbrevs}),\ T)
\end{array}
$$

The function 'treplace-ann' is based on 'treplace' (Van den Brand *et al.*, 1997c), but modified in order to collect annotations (the use of annotations is described in §4.7). The function is parameterized with a predicate (`should-rename`) and a function (`rename`) that applies a renaming to a single term. The predicate determines whether a renaming should be applied to a term. When no renaming needs to be applied to a term (i.e., 'should-rename' yields 'false') renaming is applied recursively to all its sub-terms. Whenever a matching renaming does exist ('should-rename' yields 'true'), the function 'rename' is used to replace the term according to a matching renaming. In this case the renaming is not applied recursively to the new term.

What remains is the definitions of the *overloaded* functions 'should-rename' and 'rename' for each construct that can be renamed. The specification in Appendix A contains separate modules for each construct. To illustrate the use of 'should-rename' we give the definition of the predicate for function application (the function symbol of the corresponding term is `appl`):

$$
[01]\quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{appl}, \\ \text{abbreviation}(T[\text{prod}],\ \mathit{Abbrevs}) & = & T', \\ \text{exists-matching-renaming}(T',\ \mathit{Tl}) & = & \text{true} \end{array}}{\text{should-rename}(\mathit{Tl},\ \mathit{Abbrevs},\ T)\ =\ \text{true}}
$$

For this construct we need to "*lookup*" the corresponding function declaration in the list of abbreviations. This declaration is then used to determine whether a matching renaming exists (by calling 'exists-matching-renaming'). For symbols we use the following definition:

$$
[01] \quad \frac{\begin{array}{l} \text{fun}(T) \equiv \text{sort} \\ \vee\ \text{fun}(T) \equiv \text{list} \\ \vee\ \text{fun}(T) \equiv \text{iter} \\ \vee\ \text{fun}(T) \equiv \text{iter-sep} \end{array} = \text{true}, \\ \text{exists-matching-renaming}(T,\ Tl) = \text{true}}{\text{should-rename}(Tl,\ Abbrevs,\ T) = \text{true}}
$$

For other constructs we use similar definitions (see the specification in Appendix A). To illustrate the use of the overloaded function 'rename' we give the definition for symbol renaming. In Appendix A the definitions for all other constructs are given. Note the use of the function symbol ('fun(T)') to distinguish the different constructs:

$$
[02] \quad \frac{\begin{array}{l} \text{fun}(T) \equiv \text{sort} \\ \vee\ \text{fun}(T) \equiv \text{iter} \\ \vee\ \text{fun}(T) \equiv \text{iter-sep} \end{array} = \text{true}, \\ \text{term}(\text{lookup}(T,\ Tl)) = [T_1,\ T_2]}{\text{rename}(Tl,\ Abbrevs,\ T) = T_2}
$$

This equation defines renaming of symbols (except for the renaming of lists). The renaming $[T_1, T_2]$ (that corresponds to the renaming $[T_1 \Rightarrow T_2]$ is retrieved). Since the function 'rename' is only executed for T when 'should-rename' yielded 'true', we know that there exists a matching renaming. The function 'term' is used to remove any annotation from the matching renaming. This symbol renaming (which is the simplest form of renaming), just implies the replacement of T by T_2.

4.7 Origin Specification

Remember from §3.8 that renaming with origin specification was introduced to support parameterized modules. For the instantiation of such modules, the parameter sort or production is removed from the module and a new hierarchy containing the definition of the actual parameter is imported.

For the removal of a sort or production declaration we should observe the following:

- The AsFix representation of a module makes it hard to distinguish a (sort or production) declaration from its corresponding use. Context information is therefore required to distinguish a sort declaration (in a 'sorts' section) from its use in, for example, a variable declaration. Likewise, we have to use context information to distinguish between a production declaration in a 'lexical syntax' or 'context free syntax' section and a declaration in the list of abbreviations.

- After removing a sort or production declaration from a 'sorts', 'lexical syntax' or 'context-free syntax' section, the section might become empty. Since ASF+SDF does not allow empty sections they should be removed from the module.

To implement renaming with origin specification, different renaming functions should be defined for renamings with and without origin specifications. The context of a term should be used to determine which renaming function should be used. Moreover, functions are required for the renaming of 'sorts', 'lexical syntax' and 'context-free syntax' sections. They should rename all declarations in the section and remove any section that becomes empty. Clearly, we can use the context of a section to distinguish between the declaration and use of sorts and productions. We can therefore combine the detecting of emptiness and the selection of the proper renaming function. For productions the following equation is used:

$$
[02]\quad \frac{\begin{array}{rcl} \begin{array}{l}\mathrm{fun}(T) \equiv \text{lexical-syntax} \\ \vee\ \mathrm{fun}(T) \equiv \text{context-free-syntax}\end{array} & = & \text{true}, \\ \text{map-sons-ann}(\texttt{"rename-prod"}(Tl,\ Abbrevs),\ T) & = & T', \\ \mathrm{term}(T'[\text{productions}]) & = & Tl_1 \end{array}}{\begin{array}{l} \mathrm{rename}(Tl,\ Abbrevs,\ T)\ = \\ \text{if is-empty}(Tl_1) \\ \text{then set-ann}(\mathrm{w}(\texttt{""}),\ \mathrm{ann}(T')) \\ \text{else } T' \text{ fi} \end{array}}
$$

This renaming function is applied to 'lexical syntax' and 'context-free syntax' sections. It applies the renaming recursively to all declarations (using 'map-sons-ann'). The function `rename-prod`, that will remove declarations according to origin specifications, is used here for the renaming of production declarations. The result of 'rename' is either empty white space ('w("")') to denote that the section has become empty and therefore has been removed, or the (non-empty) renamed section `T'`. The term that is returned is an annotated term. The use of annotations is described below.

The recursive approach of the application of a renaming to the AsFix representation of a module makes it impossible to update different terms at different levels in the tree simultaneously. Declarations of sorts and productions and import sections occur at different levels in the tree. In the case of a renaming with origin specification it is therefore not possible to remove a declaration and, at the same time, add the corresponding module name to an import section.

We should therefore collect the set of module names corresponding to renamings with origin specifications and construct an import section as a separate phase in the renaming process. For the collection if these module names we can choose among three different approaches:

- *Collection of module names prior to renaming.* This approach will collect the module names corresponding to matching renamings with origin specifications during a traversal of the tree prior to renaming. During the renaming phase, the corresponding declarations are removed. This is an expensive operation because it requires the traversal of the tree twice (once to collect the module names and once to apply the renaming to the tree).

- *Collecting the modules after renaming.* This approach is even more expensive. If we want to know whether the declaration of B in a renamed module M' is the result of the renaming `[A => B <N>]` we have to determine whether the declaration of A belongs to the original module M. Furthermore, in order to find matching declarations, these declarations cannot be removed during the renaming phase. As a result, the tree has to be traversed at least three times: once to apply the renaming to the module, once to collect module names according to matching renamings with origin specifications and once to remove their declarations.

- *Collect the modules during renaming.* This approach has the advantage that the collection of module names, the renaming of the module and the removal of declarations can be combined.

To modify the AsFix representation of a module and to collect the set of module names simultaneously, renaming a module should yield both a modified module and a list of module names. Here again, ATerms prove to be very useful since it enables the use of annotation to combine the list of module names and the modified ATerm. Whenever a declaration is removed according to a renaming with origin specification, an empty term is returned and the corresponding module name is attached to it by means of an annotation (hence the use of 'set-ann' in the equation above). The tree replacement function ('treplace-ann') will move the annotations from the children of a tree to its root. As a consequence, the AsFix representation of a module will contain the list of module names that needs to be imported as an annotation after the renaming has been applied. From this list a new imports section is created and appended to the renamed module.

Chapter 5

Implementation of a Renaming Tool

5.1 Introduction

The specification described in the previous chapter is able to apply a renaming to the AsFix representation of a single module only. The complete process of renaming a module including all modules that it imports is far more complicated. This process includes the conversion to and from AsFix, the renaming of imported modules and the prevention of redundancy resulting from different renamings yielding equivalent modules. This chapter concentrates on several of these topics. In §5.2 we describe how we can improve the performance of the tool by taking dependency information into account. In §5.3 we describe what actions are involved for the renaming of a single module. In §5.4 we explain how the renaming is extended in order to apply a renaming to a complete module hierarchy, §5.5 concentrates on avoiding redundant modules. In §5.6 we give some suggestions for the organization of modules within a directory structure. Finally, in §5.7 we describe shortly how renaming is applied to a module hierarchy (i.e., in what order the renaming is applied etc.).

5.2 Dependencies

Several steps involved in the renaming process of a module are rather time consuming. In a module hierarchy containing many modules, some of which are renamings themselves, the performance of renaming will decrease dramatically due to the recursive application of the renaming to all modules involved. Executing the time consuming steps only sparingly can save much time.

Dependency information helps to increase the performance because it provides information about the minimal set of modules to which a renaming should be applied.

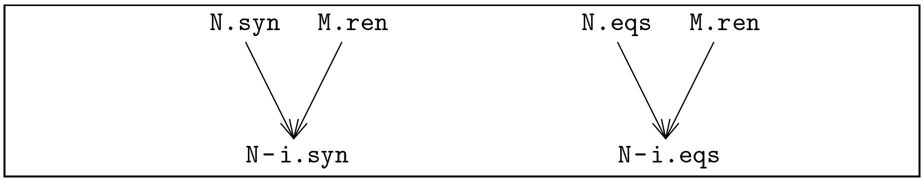


Figure 5.1: Dependency graphs corresponding to the file dependencies for `M-i.syn` and `M-i.eqs`.

Definition 5.2.1 (Dependency Relation) We define a binary dependency relation ($\rightarrow$) between two files (or modules):

$$N \rightarrow M = \begin{cases} true & \text{whenever the contents of } M \text{ depends on the contents of } N, \\ false & \text{otherwise.} \end{cases}$$

Furthermore, we define the transitive relation $\stackrel{*}{\rightarrow}$ for some files K, M and N:

$$N \stackrel{*}{\rightarrow} M = \begin{cases} true & \text{when } N = M \text{ or } N \rightarrow K \text{ and } K \stackrel{*}{\rightarrow} M, \\ false & \text{otherwise.} \end{cases}$$

□

We define a dependency graph for M to be a directed graph. The nodes of the graph are the files (or modules) for which $N \stackrel{*}{\rightarrow} M$ holds. An arrow from N to M is drawn whenever $N \rightarrow M$ holds.

To avoid unnecessary execution of steps in the renaming process we will construct a file/module M in the (directed) graph only when some file/module N ($N \neq M$) above in the dependency graph has changed. That is, whenever N has changed and $N \stackrel{*}{\rightarrow} M$ holds.

We distinguish two types of dependencies:

- *File dependencies.* The construction of `M-i.syn` (i.e., the file containing the syntactical part of the renamed module M) depends on `M.syn` and `R.ren` (the file in which the renaming is defined). The file dependencies for `M-i.eqs` are likewise. The file dependencies for the files `M-i.syn` and `M-i.eqs` are depicted in Figure 5.1. In this text we silently ignore the fact that the AsFix representation of both the original and the renamed module should be included in the graph too. Note that the dependency graph only contains file names, hence the name File Dependency.

- *Module Dependencies.* A renaming applied to a module M is also applied to all modules imported by M. The resulting module depends therefore on its imported modules, since the module name of an imported module may have changed as a result of the renaming. Note that only renamed modules have module dependencies. Therefore, we define $N \rightarrow M\text{-}i$ to hold whenever $M\text{-}i$ imports N. The corresponding dependency graph contains only module names, hence the name Module Dependency.

These dependencies are used to update the module table (see §3.11). The module table, that is stored in the file `.module-table` satisfies the property:

Property 5.2.2 (Module Table) Whenever an entry M, R, i is contained in the module table then the module M-i is the result of applying the renaming R to the module M and M-i is up-to-date. □

According to the type of the dependency the module table is updated as follows:

- *File Dependencies.* Whenever the files `M.syn` or `M.eqs` have changed, all entries in the module table relating M to some module M-j are removed. They are removed because the modules M-j are no longer up-to-date and the renamings have therefore to be applied again. When a renaming module `N.ren` has changed, all entries referring to that renaming are removed from the module table.

  Whenever a file F in the dependency graph of the files `M.syn` or `M.eqs` has been changed, all files occurring in a path rooted at F in these dependency graphs are removed. That is, all files K (with $K \neq F$) for which $F \xrightarrow{*} K$ holds are removed.

- *Module Dependencies.* When a module N in the module dependency graph of M-i has changed we remove the entry from the module table that maps the module M to M-i. Unrelated renamed modules (i.e., renamed modules for which the module table does not contain a mapping) will be removed.

The removal of files and unrelated modules guarantees that all out-dated information is removed during the dependency checks. As a result, some of them will be constructed again during the remaining renaming process (this time by using up-to-date information). The dependency checks increase the performance of renaming because a module/file will only be reconstructed if one or more of the files/modules in its dependency graph have been modified.

5.3 Single Module Renaming

The renaming of a single module takes place in several steps. Since renaming is defined on the AsFix representation of a module but because modules forming a specification are stored in their textual representation, many steps are concerned with the conversion from and to AsFix. These steps are displayed in Figure 5.2. The figure shows the process of applying a renaming M (stored in the renaming module `M.ren`) to the module N. The names of the tools that are involved are displayed within the boxes. The file formats, the tools take as input and produce as output, are displayed next to the arrows.

The rest of this section will concentrate on these conversions and on the execution of the renaming specification.

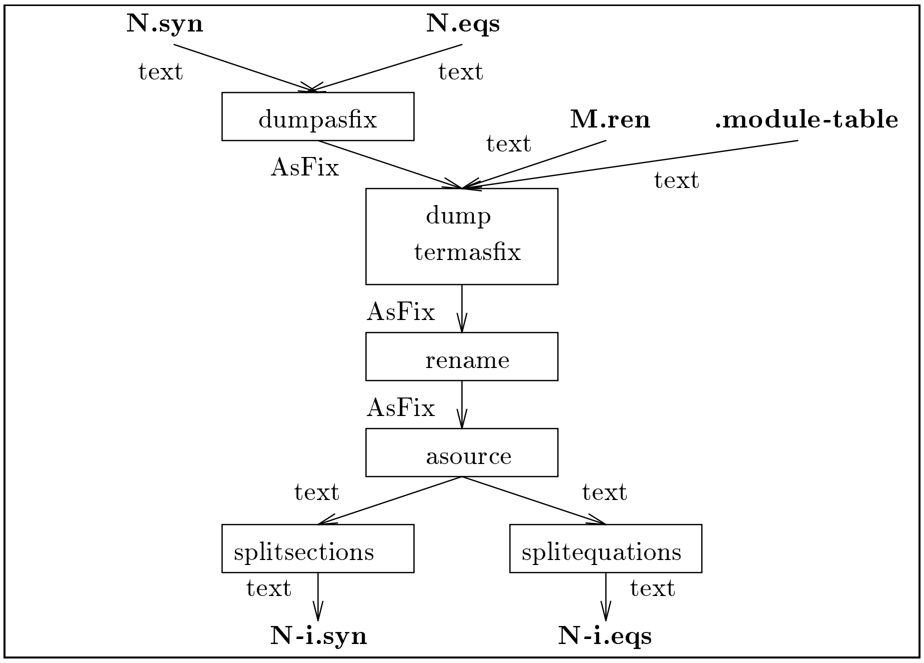


Figure 5.2: Overview of the renaming of a single module.

5.3.1 Conversion to and from AsFix

The AsFix representation of a specification to which a renaming is applied is just an encoding of the parse tree of the specification. The Meta-Environment is currently required to perform this encoding. Fortunately, the stand-alone tool `dumpasfix` exists that automates the encoding of the specification and the dump to a file without user intervention. To re-obtain the textual representation after a renaming has been applied, the tool `asource` ("AsFix source") exists. This tool takes an AsFix term as input and reconstructs the source string that was used to create the AsFix term (i.e., it decodes the AsFix term). However, the source text that `asource` gives as output does not contain the division of a module in its syntactical and equational part. To be able to store a module correctly (i.e., the syntactical part of a module M in the file `M.syn` and its equational part in `M.eqs`), the text has to be split. The tools `splitsections` and `splitequations` are used to obtain the syntactical part and the equational part of a module respectively. However, the output of `asource` cannot be divided correctly in all cases (see §5.5).

5.3.2 A Stand-Alone Specification

The specification of module renaming (included in Appendix A) is compiled to a stand-alone term rewriting system. The decision to use the compiled specification was motivated by the increase of performance compared to the execution of the specification in the Meta-Environment. Moreover, by using the Meta-Environment we would have to deal with difficulties in order to control the Meta-Environment without user intervention since it is an interactive system.

A stand-alone term rewriting system (TRS) is constructed by compiling the AsFix representation of the specification using the AsFix compiler `ac`(1). The result is the component `rename` in Figure 5.2.

The stand-alone TRS requires as input the AsFix representation of a term which is valid for the specification. Like for the construction of the AsFix representation of a specification, the Meta-Environment is required to encode a term in AsFix. The tool `dumptermasfix` (like `dumpasfix`) automates the process of encoding a term in AsFix and dumping it on a file.

5.4 Renaming a Module Hierarchy

Since a renaming can affect any module in the module hierarchy of a specification, a renaming should be applied to all modules in the module hierarchy. When a renaming affects a module M the name of the module can change to M'. Either by appending an identifier (in case M does not already have an identifier appended to its module name) or by changing the identifier. In both cases, a module that imports M should update its import section to import M' instead of M. Figure 3.3 showed that all modules that import M directly or indirectly should be renamed, in order to update their import sections.

In order to rename the modules contained in a single hierarchy correctly, the following requirement should be satisfied:

Requirement 5.4.1 A renaming should not yield a specification containing ambiguous instances of the same module. This implies:

1. If the module table contains an entry M, R, i before the renaming R has been applied to the module M and the table contains the entry M, R, j afterwards ($i \neq j$), then all occurrences of M-i in a single hierarchy should be replaced by M-j during renaming.

2. When two instances of the same module M-i and M-j ($i \neq j$) are included in a single module hierarchy then $M\text{-}i \cup M\text{-}j$ should be unambiguous. This forms a limitation for the contents of reusable modules, since it implies that an instantiation that affects a module should affect all sorts and productions in the module. For the modules M-i and M-j this corresponds to: $\mathrm{Vis}(M\text{-}i) \cap \mathrm{Vis}(M\text{-}j) = \emptyset$.

□

The second implication cannot be satisfied by the renaming tool because it depends on the design of modules. Chapter 8 and Chapter 9 concentrate on the correct design of reusable modules.

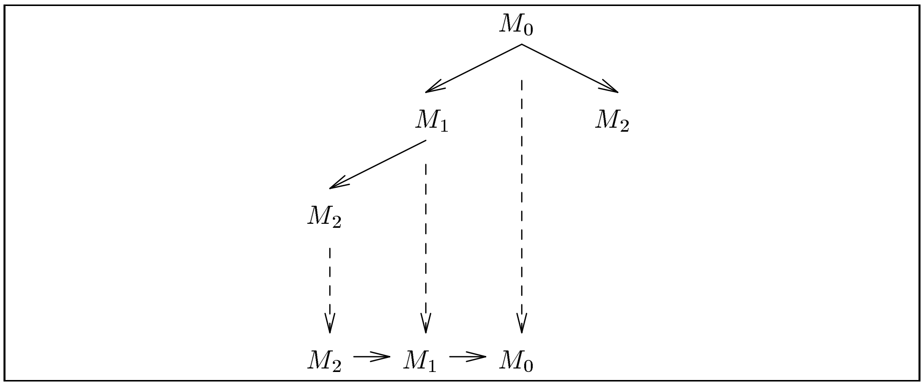


Figure 5.3: Depth first order in which a renaming should be applied to the modules included in the import graph of M_0.

The first implication of Requirement 5.4.1 guarantees that after a renaming has been applied, a module hierarchy has been updated according to the information in the module table. The property is required in ASF+SDF to avoid ambiguous sort and production declarations (see §3.8 and §3.11).

It is not before renaming has been applied to all modules imported by M that the import section of M can be updated correctly. Thus, the renaming of a module should be applied in a fixed order according to the imported modules. This depth first order is demonstrated in Figure 5.3.

The tool `get-import-graph` that is contained in the *graph-tools* package (Moonen, 1996b) uses the textual representation of modules to collect the set of modules that are imported. It operates recursively on all imported modules and gives as output the import graph in a format that represents the hierarchical structure. However, the tool does not take module identifiers into account.

A module name M-*id* in an import section of a module may not correspond to a valid module during the renaming. Due to the dependency checks the module may have been removed, or the entry that maps M to M-*id* might have been removed. As a result, the tool `get-import-graph` is not able to define the module hierarchy correctly. We have therefore implemented a tool (that is based on `get-import-graph`) that does account for module identifiers and accesses the module table to obtain correct module names. Moreover, the tool outputs the modules in depth first order directly (i.e., no graph is returned). The list that is generated by the tool contains each module from the module hierarchy exactly once. That is why the module M_2 imported by M_0 in Figure 5.3 is included in the list only once.

5.4.1 Recursive Renaming

Since an imported module N may be the result of a renaming itself (i.e., the import of a renamed module), we should note the following possible problems:

- When the order is determined in which the renaming should be applied to the modules of a module hierarchy, the module N does not exist. This indicates that N has not yet been constructed. As a consequence, the module hierarchy cannot be determined completely.

- N does exist, but its contents are not up-to-date. According to its dependencies (see §5.2), either the renaming module `N.ren` or the module that the renaming is defined on has been modified. As a consequence, the set of imported modules might be incorrect. Either it contains too many, too few or invalid modules. In the latter case some of the imported modules have incorrect identifiers and the module hierarchy can therefore not be determined correctly.

To resolve these problems we have to execute the renaming tool recursively for each module in the module hierarchy that is based on a renaming. The module hierarchy can be constructed correctly after these renamings have been applied.

5.4.2 Origin Specification

A renaming with origin specification N can extend the module hierarchy of a specification M. Since both module hierarchies M and N can overlap (that is, $\exists_m \langle m \in \text{Imports}(M) \wedge m \in \text{Imports}(N)\rangle$), a renaming should also be applied to an origin specification in order to satisfy Requirement 5.4.1.

Now consider the following modules M and N:

```
module M          module N
exports           exports
   sorts A           sorts A
```

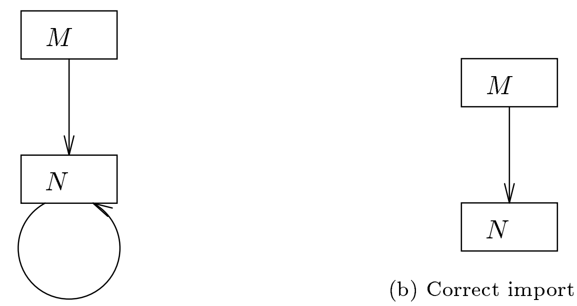
and a renaming with origin specification (i.e., an instantiation of M): `M[A => A <N>]`.

When we apply this renaming to the module M, the renaming will also be applied to N (as required). Observe that the renaming `[A => A <N>]` is a matching renaming for module N too. The origin specification will therefore result in a cyclic import since the sort A will be removed from N and the module N will be imported. This results in the module hierarchy shown in Figure 5.4(a).

Of course, the renaming should have left the module N unchanged. In general, when a renaming is applied to a module hierarchy corresponding to an origin specification, that origin specification should be removed from the set of renamings. That is, the renaming `[A => A <N>]` should have been removed, yielding an empty renaming which would have left N unchanged (Figure 5.4(b)).

5.5 Reusing Modules

The module table that is used by the renaming tool contains module mappings. A module mapping indicates in which module the result of a renaming is stored.



(a) Renaming yielded a cyclic import

(b) Correct import graph

Figure 5.4: Applying a renaming `[A => A <N>]` to the origin specification N yields a cyclic import (Figure 5.4(a)). In order to obtain a correct import graph (Figure 5.4(b)), this renaming should be removed from the set of renamings before the renaming is applied to N.

The motivation for the use of a module table was described in §3.11. The table serves three purposes:

1. It is accessed to retrieve valid module names in order to update the import sections of a module correctly (see §5.4).

2. It is used to determine whether a renaming needs to be applied to a module. According to Property 5.2.2, a renaming needs to be applied to a module when the module table does not contain an entry for that module and renaming (see §5.2).

3. Finally, the table is used to promote the reuse of equivalent modules. The table is thus used to prevent redundancy.

Avoiding redundancy by reusing modules is the subject of this section.

Definition 5.5.1 (Base name) Let us define the base name of a module to be the module name without a module identifier:

$$\begin{aligned} basename(M) &= M \\ basename(M\text{-}id) &= M \end{aligned}$$

□

For each module that has been renamed, an entry is appended to the module table. We should consider the following cases:

- The application of a renaming R to a module M has resulted in some module $M' \neq M$. In this case we allocate a unique identifier (> 0) and append the entry $basename(M), R, id$ to the table.

- A renaming R applied to a module M did not modify the module. When $basename(M) = M$, we add the entry $M, R, 0$ to the module table (remember from §3.11 that a zero identifier is used to indicate that a module without a module identifier was not affected by a renaming). When $basename(M) \neq M$, we reuse the module identifier i of M and add the entry $basename(M), R, i$ to the table.

Now consider a module M and the renamings R_1 and R_2:

```
module M
exports
   sorts S
```

```
M[S => S']
```

R_1

```
M[S => S'
  B => B']
```

R_2

The application of R_1 and R_2 to M results in two modules M-i and M-j respectively ($i \neq j$). Since both renamings will affect module M, i and j will not be zero identifiers. Moreover, although both renamings are different, their application to the module M will yield the same result, introducing redundancy.

In order to resolve this redundancy we allow module identifiers to be shared among different renamings. In the example above only one identifier id will be allocated and shared by both modules. The module table is updated by the two entries M, R_1, id and M, R_2, id.

Of course we cannot and will not detect all redundancies. We only prevent redundancies that result from the application of different renamings to the same module (according to Property 3.11.1). Due to the naming mechanism that is used, these equivalent modules will have equal base names. Redundancies due to equivalent modules with different base names are not resolved.

To promote the reuse of modules, the following steps are required for the detection and prevention of redundancies:

1. Obtain a module identifier. According to a renaming R and a module M, locate its corresponding identifier in the module table. When no such entry exists, allocate a new (and unique) identifier. A unique identifier is obtained by increasing the maximum module identifier that is in use by one.

2. Apply the renaming R to the module M and store the result in the module M-i (using the module identifier i that has just been obtained).

3. Retrieve all identifiers $(id_1, \ldots, id_n)$ from the module table according to M (i.e., retrieve all identifiers according to the entries $M, *, *$ in the table). In order to detect redundancy we compare the contents of the module M-i with all modules M-$id_1, \ldots, M$-id_n and the original module M. When we

find an equivalent module N, we have found redundancy. When $N = M$, the renaming did not affect the original module M. We therefore add the entry $M, R, 0$ to the table. In all other cases we will share the identifier (say id_i, $1 \leq i \leq n$) of N and add the entry M, R, id_i to the table. Note that also in the case when id_i equals zero, the renaming did not affect the module M. Because we have just reused an existing module, we can remove M-i in order to resolve the redundancy.

It is not difficult to see that this mechanism works and prevents redundancy. Consider a consistent set of modules (i.e., a set of modules without redundancies). The renaming of a module M according to a renaming R can yield a unique or a redundant module M-i. In the first case, the set of modules remains consistent since of course no redundancy is introduced. In the latter case, there exists exactly one module M' that is equivalent to M-i. According to the mechanism described above, the module M-i will be removed and either the module identifier of M' will be shared or the zero identifier will be used (in case $basename(M') = M'$). Because the redundant module M-i has been removed, the set of modules will remain consistent after the renaming has been applied.

To determine whether two modules are equivalent the textual representations of the modules are compared. This check will detect any difference between two modules. Thus, even when two modules differ in white space only, they will be considered unequal.

The executable specification that applies a renaming to a module, will output an AsFix term representing the renamed module (see Figure 5.2). Unfortunately, the AsFix representation does not maintain the exact division of a module in its syntactical and equational part correctly when the latter contains initial white space. The division is made implicit in the AsFix representation by the occurrence of the keyword `equations` (or `no-equations` in case of an empty equational part). Initial comments in the equational part of a module will therefore be appended to the end of the syntactical part. As a consequence, a module of which the equational part contains initial white space, cannot be converted back correctly once it has been converted to AsFix.

When two modules have to be compared for equivalence we should consider the following:

- We can use the AsFix representation as a base. That is, we take the AsFix representation of two modules and use the tool `asource` to convert it to text. This textual representation is used to compare the two modules. Since `asource` outputs a single module (i.e., it does not split the module in a syntactical and equational part), the problem described above does not exist. However, `asource` requires the existence of the AsFix representation of a module. Because we consider AsFix modules as intermediate results during the renaming process (they may be removed at any time), we think that this requirement is too strong.

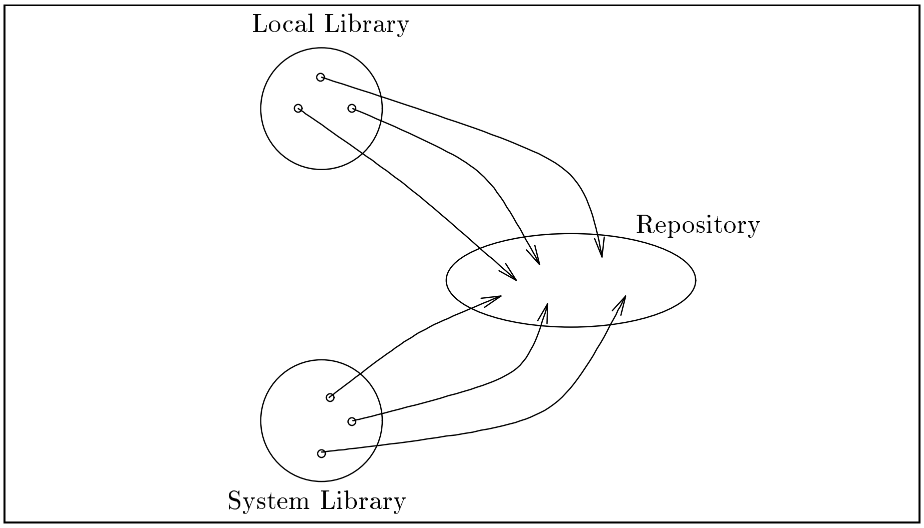


Figure 5.5: Example of a Module Repository

- We can use the module representation as a base. To compare two modules, we compare their syntactical parts and their equational parts separately. In this case however, we may get wrong results due to an incorrect division of a module in its syntactical and equational part as described above.

Therefore, we will combine the two approaches by using the module representation and joining the syntactical and equational part. The syntactical and equational part together are used for the comparison.

5.6 Module Storage and Retrieval

Using the renaming tool, one is able to construct libraries of generic modules. These modules can be instantiated by means of renaming. The renaming of a module results in the creation of one or more new modules (unless the renaming does not affect a module of course). Ideally, one would share these instantiated modules among different users in order to promote reuse (and prevent redundancy). However, this will introduce serious concurrency problems. For example, we would have to deal with the potential application of the same renaming simultaneously. Moreover, each user should have write access to the directories in which the generated modules are stored. Due to these difficulties, we have decided not to support any concurrency in the renaming tool.

To promote individual reuse of modules we suggest the use of a personal *repository*. Due to the mechanism by which the Meta-Environment finds its modules (using the search paths defined in `ctasdf.conf`), the use of more complex module hierarchies (for example the storage of modules that form a single library in different sub-directories) becomes very hard to configure (and main-

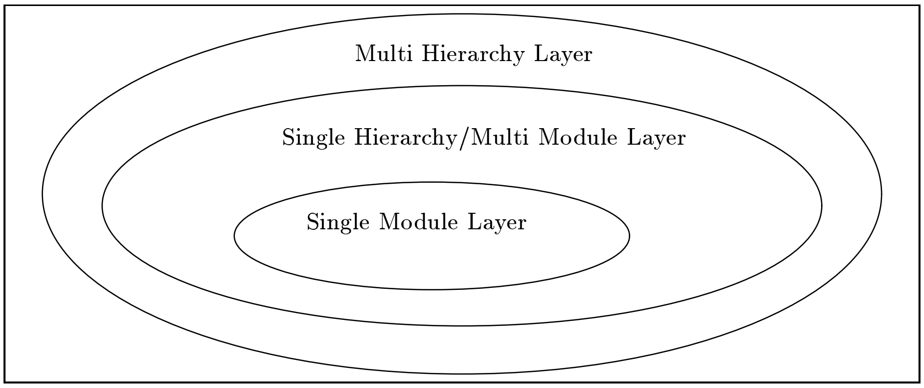


Figure 5.6: The three layers Layers of Renaming.

tain). For each specification we should include all directories in the configuration file of each library that we want to use.

The use of a single repository in which all modules and instances of modules are stored, makes the configuration of the Meta-Environment very easy because only one directory needs to be present in the configuration file. In the repository one can store links to any (system wide or local) library. By using links the libraries themselves can be constructed using any hierarchy. The suggested use of the repository is shown in Figure 5.5. We have provided some tools for the creation (and removal) of links from a library hierarchy to a repository.

5.7 Putting Things Together

After having described most of the components that form the renaming tool and after discussed most important issues involved in the renaming process, we will now describe the application of the renaming process. The complete renaming process can be divided in three layers corresponding to the renaming of multiple module hierarchies, the renaming of a single module hierarchy and the renaming of a single module (see Figure 5.6).

Below we will briefly describe how the actions executed in each layer will contribute to the renaming process.

- Multi Hierarchy Layer:
  The steps performed at this level are:

  1. Performing dependency checks on the renaming module. Whenever the renaming module has changed, all entries in the module table that reference this renaming are removed.
  2. The application of the renaming to each individual module hierarchy. A renaming module defines at least one module hierarchy (namely the hierarchy to which the renaming should be applied). Other hierarchies are defined by origin specifications. Whenever a renaming

contains origin specifications, the renaming is applied to the origin specifications first.

3. After a renaming R has been applied to all module hierarchies, the file `R.syn` is constructed (see §5.3).

- Single Hierarchy / Multi Module Layer:
  At this level the depth first order at which the renaming should be applied is determined (see §5.4). Furthermore, the renaming tool is executed recursively for any module included in the hierarchy that is based on a renaming module. Finally, the renaming is applied in depth first order to each module in the hierarchy.

- Single Module Layer:
  At this level a renaming is applied to a single module (by executing the executable specification), redundancy is removed and the module table is updated. Furthermore, the conversions to and from AsFix takes place.

Chapter 6

Conclusions

The renaming tool as described in the previous chapters has been implemented as a set of shell scripts, perl scripts, Makefiles and a stand-alone specification. The set of tools that form the renaming tool together with a short description is listed in Appendix C. The complete specification is included in Appendix A. We have used the renaming tool for several experiments to test its usefulness including the instantiation of several data types defined in Bergstra and Sellink (1996). The use of the renaming tool for the development of a library of hierarchical data types is the subject of Chapter 9.

Despite its usefulness, the current implementation has several drawbacks. First, we should note that the performance is really poor. This poor performance is caused by two factors. For each application of a renaming to a module an AsFix term needs to be constructed. Currently the only way to obtain the AsFix representation of a term over a specification is by using the Meta-Environment. Hence, for each application of a renaming the complete Meta-Environment has to be started and a term has to be constructed. For small modules this is the most time consuming operation of the renaming part. The second factor that is responsible for the poor performance is the execution of the stand-alone specification. It decreases the performance because it takes much time to load all modules corresponding to a complete specification. The execution time is increased even more due to the growing number of reduction steps involved for the renaming of larger modules.

The requirement of the Meta-Environment for the application of renaming is another drawback of the current implementation. Although the renaming tool is an external tool, its use becomes restricted to the very limited architectures supported by the Meta-Environment.

Finally, we mention the restrictions with respect to the configuration of the tool. Because the renaming mechanism consists of several tools, we have to deal with some configuration and installation issues. Items that need to be configured include path names, names of tools and their locations. The tools that we have developed are all configured using environment variables. All variables required by the tools are configured in a single shell script. Concentrating the parameters of the system makes it more easy to move the system to another directory structure since only one module has to be modified. Unfortunately, the renaming tool makes use of several third party tools. Many of these tools contain hard-coded paths. To avoid redundant tools we have decided to use existing tools whenever possible instead of adapting them to remove the hard-coded paths. This decision however, makes the movement of the system to another directory structure more difficult.

6.1 Future Work

Because the performance of the renaming tool restricts its practical use, the improvement of the performance has high priority. The performance of the tool can be improved by removing the two bottlenecks in the renaming process (the dump of an AsFix term and the execution of the stand-alone specification). To remove the first bottleneck, we should investigate whether it is possibly to bypass the Meta-Environment during the construction of AsFix terms. For the removal of the second bottleneck we mention the development of a new compiler that will probably increase the performance significantly. This compiler translates an ASF+SDF specification to C directly and would make the current ARM-interpreter (Kamperman and Walters, 1993, 1995) obsolete.

In order to improve the portability of the tool, we mention the current development of the new Environment (Van den Brand *et al.*, 1997a, 1997d). To improve the tools that are used by the renaming tool (in particular by removing hard-coded paths), we suggest a re-implementation and a standard distribution of tools that are related to the ASF+SDF formalism.

Part II

Reuse of ASF+SDF Specifications

Chapter 7

Reusable Components

Software development productivity can be increased by reusing standard components. In general, by using reusable components, constructing a software system can be seen as piecing together a number of standard components that are adapted to meet the particular requirements of the new system.

Development with reuse involves answering the question *whether* a particular component is reusable within the new system. In order to maximize reuse, one should be able to select as many suitable components from a collection (or repository) of reusable components as possible. Because such repositories might become quite large after some time, the process of selecting the correct components becomes difficult. Many articles in the literature concentrate therefore on the storage and retrieval of components. Retrieval mechanisms are based on a notion of *specification matching* (Rouaix, 1992). Hence, for the storage of components detailed specifications of the components have to be stored as well. Such specifications can either be informal (for example, by giving a textual characterization of the aspects of the component (Thunem and Sindre, 1992; Constantopoulos *et al.*, 1992)) or formal (for example, by means of algebraic specification techniques (Bert and Jacquet, 1992)). For the retrieval of components, several tools have been developed that helps the user to find suitable components in the repository (Constantopoulos *et al.*, 1992; Moineau and Gaudel, 1992).

We observed that many articles consider a repository to be an unordered collection of components (i.e., no internal relation between the components is assumed). When reuse has to be maximized within the repository however, components should be developed by reusing existing ones. The components then become related and one might wonder whether a repository can be ordered into clusters of such related components. Reusing components within a repository involves answering the question whether a new component *can* be developed by reusing existing ones.

We found several formalisms for which a library of such related components has been developed. These libraries are structured hierarchically, such that modules lower in the hierarchy are designed by reusing components higher in the hierarchy. Examples of these formalisms and their libraries that we have studied include: AFSL (Groenboom, 1997; Groenboom and Renardel de Lavalette, 1996), Larch (Guttag and Horning, 1993), OPAL (Pepper, 1991; Didrich, 1994) and PSF (Mauw and Veltink, 1990; Van Wamel, 1993). Although these libraries are structured hierarchically, they all have in common that the hierarchical structure is limited by the formalism (i.e., several unrelated components exist that could have been designed by means of reuse). Furthermore, the hierarchical structure of these libraries is not motivated. As a consequence, it is not always clear why these libraries are structured differently. Hence, the question when a component $\mathcal{A}$ can be developed by reusing a component $\mathcal{B}$ remains.

In the next chapters we will concentrate on this hierarchical structuring of components. After we discuss some general ideas, we will focus on the hierarchical structuring of data types specified in ASF+SDF using the renaming and parameterization mechanism developed in Part I. The goal is to maximize the hierarchical structuring without being restricted by the formalism.

In Chapter 8 we will define a framework for the development of reusable modules in ASF+SDF and we discuss some ideas that concentrate on the question when a component can be included in a hierarchy of data types (we restrict ourselves to data types only). These ideas are demonstrated in a case study showing how a library of reusable data types can be developed in Chapter 9. The chapter concludes with an extensive example in §9.10 demonstrating how the renaming tool is used to instantiate parameterized data types and how new data types can be derived from existing ones. Finally, in Chapter 10 some results and possible improvements are discussed.

Chapter 8

Design of Reusable Modules

8.1 Introduction

Using the renaming tool defined in part I, we are now able to define reusable modules. The renaming mechanism enables us to adapt the syntax of a module for its particular use. With its parameterization mechanism we can define generic modules, that we can instantiate to modify the semantics of the module.

This chapter concentrates on maximizing reuse in the design of new data types (the discussion in this chapter is limited to the development of data types but can easily be extended to components in general). The chapter discusses ideas about the development of data types by reusing existing ones (i.e., the development of data types *based on* existing data types). These ideas are used to define a framework for the development of a hierarchy of data types in ASF+SDF.

The chapter starts with a description of parameterized modules and data types in §8.2. The importance of parameterization for the development of reusable modules and data types becomes clear in the remaining sections and in Chapter 9. In §8.3 we discuss in what way the functionality of a derived module may differ from its base. We introduce an informal notion for the ordering of data types. This ordering will be used for the hierarchical structuring of data types. Three factors are distinguished that motivate the adaptation of functionality derived from a data type: *extending* the functionality, *restricting* the functionality and *adapting* the functionality (by changing the semantics). In §8.4 we introduce a layered module structure for the development of data types in ASF+SDF that supports the modification of derived functionality.

Since ASF+SDF has user definable syntax, reusable modules should not depend on any representation. Therefore, we discuss in §8.5 how a module can be defined independently of a representation, by introducing parameterized *access* functions.

8.2 Parameterized Modules

In a formalism with parameterization sorts and productions can be defined as formal parameter of the module (Mauw and Veltink, 1990; Bergstra *et al.*, 1989; Guttag and Horning, 1993). The formal parameter in the module stands for an arbitrary sort or production. An instantiation of a parameterized module consist in the replacement of each occurrence of a formal parameter by an actual one.

A data type that is being defined in several modules is defined to be a *parameterized data type* when one of its modules is parameterized. Instantiating all parameters of a parameterized data type yields a *non-parameterized data type* (Loeckx *et al.*, 1996).

In ASF+SDF, parameterization does not exist. This section concentrates on a convention for the simulation of parameterized modules and parameterized data types in ASF+SDF.

Since a formal parameter stands for an arbitrary sort or production, no semantics is assigned to the parameter. This observation and the definitions of undefined sorts and functions in §3.4 that do not have a semantics either, formed the motivation for the (informal) definition of parameter sorts and productions (i.e., undefined exported sorts or productions respectively, see Convention 3.4.1). For the instantiation of modules we follow the mechanism described in Bergstra (1987) which requires renamings with origin specification (§3.8). We define an instantiation to be *complete*, when the module (or data type) becomes non-parameterized afterwards. When one or more parameters remain uninstantiated, the instantiation will be called *incomplete*.

Example 8.2.1 Consider the module below. It defines a function `sample` using both a parameter sort `S` and a parameter function `eq`:

```
imports Bool
exports
   sorts S
   context-free syntax
      "sample" "(" S "," S ")" -> Bool
          "eq" "(" S "," S ")" -> Bool
   variables
      S[12] -> S
equations
[01] sample( S1, S2 ) = eq( S1, S2 )
```

Using a renaming we can instantiate this module. The renaming below uses the sorts `Int` and the (infix) function '`==`', both defined in the module `Integer`, to instantiate the module. Observe that the instantiation is complete because it will instantiate all formal parameters of the module:

```
[
   S => Int <Integer>
   "eq" "(" S "," S ")" -> Bool =>
```

```
      Int "==" Int -> Bool {left} <Integer>
   ]
```

Because of the origin specifications, the declarations of the sort `S` and the function `eq` will vanish from the module. Instead, the module `Integer` will be imported. The definitions of the sort `Int` and the function '`==`' from this module are used. All occurrences of the sort `S` will be replaced by `Int` and the occurrence of `eq` at the right hand side of equation 1 will be replaced by '`==`':

```
imports Integer
imports Bool
exports
   context-free syntax
      "sample" "(" Int "," Int ")" -> Bool
   variables
      S[12] -> Int
equations
[01] sample( S1, S2 ) = S1 == S2
```

□

Due to the definition of parameter sorts and productions, parameterized specifications can be type-checked, even when the specification has not (or only partly) been instantiated. Moreover, since a parameter function is declared, the number of its arguments is known. An uninstantiated specification can therefore be checked for correct function calls (with respect to the number and types of its arguments).

Due to the lack of a formal notion of parameterization in the ASF+SDF formalism, there is no way to restrict renamings with origin specifications to operate on parameters only. Since such renamings are not conform our definition of instantiation (Convention 3.4.1), they are not considered to be instantiations and are therefore considered as invalid renamings.

8.3 Extending and Restricting Functionality

In a hierarchy of data types (for example as described in Uhl and Schmid (1990)), data types down in the hierarchy are based on modules above. That is, they are constructed by reusing (part of) the definition of the data types above. This is called *inheritance* in an object oriented world.

Definition 8.3.1 (Ordering of Data Types) Let us associate with a data type D a set of properties (prop(D)). Furthermore, consider a hierarchy of data types H and let depth(D, H) denote the depth at which D occurs in the hierarchy H. Now consider two data types D_1 and D_2 (both contained in a single path in H) and their corresponding sets of properties. We will define a partial ordering $\prec$, satisfying:

$$\mathrm{depth}(D_1, H) \leq \mathrm{depth}(D_2, H)) \leftrightarrow \mathrm{prop}(D_1) \prec \mathrm{prop}(D_2)$$

In words this definition states that data types higher in the hierarchy are more generic (i.e., they enforce less restrictions on its elements). A new data type D_n can be derived from a data type D when its associated set of properties forms an extension to the properties of D (thus, $prop(D) \subseteq prop(D_n)$). □

We should emphasize that this definition is rather informal because we did not define how the set of properties for a data type can be defined. Nevertheless, the definition gives some intuition for the hierarchical structuring of data types. The exact definition of the properties associated with a data type requires further investigation.

Example 8.3.2 (Ordering of Data Types) This example demonstrates several properties that can be associated with different data types. Furthermore, we demonstrate the use of the sets of properties for the hierarchical structuring of data types.
A set S in which the elements are indexed uniquely, satisfies the property:

$$e_i, e_j \in S \wedge e_i = e_j \rightarrow i = j$$

Two properties that are satisfied by a List L are:

$$\begin{array}{lll} insert(L,e) = L' & \rightarrow & e \in L' \\ length(L) = n & \rightarrow & length(insert(L,e)) = n+1 \end{array}$$

The first property states that after inserting an element, that element is contained somewhere in the list. Hence, this property does not enforce the exact position where the element will be inserted. The second property states that the length of a list is always increased by inserting an element. Observe that the latter property is not satisfied by the data type Set defined above.

The data type Stack introduces the new property:

$$top(push(S,e)) = e$$

This property states that an element is pushed *on top* of a stack S and therefore *does* enforce the position where an element is inserted. Observe that both properties associated with lists are still satisfied. This property forms therefore an extension to the set of properties associated with the data type List. As a consequence, the data type Stack can be derived from List according to Definition 8.3.1. □

Obviously, the functions corresponding to a data type should satisfy all properties of the data type. When a new data type D_n is derived from a data type D we should therefore note the following with respect to the functions of D_n and its new properties (i.e., the set of properties $\text{prop}(D_n) \setminus \text{prop}(D)$):

- Using the new properties we can define new functions that make sense for D_n but not for D. In this case the signature of D_n should be *extended* with respect to the signature of D (see §8.3.1).

- A function defined on D may conflict with the properties of the new data type D_n. In this case the signature of D_n needs to be *restricted* (see §8.3.2).

- A function in D does not work correctly in D_n because not all properties are satisfied. However, these properties can be satisfied by *adapting* the derived function definition (see §8.3.3).

In the following paragraphs we will concentrate on these three situations. In §8.4 we will develop a framework for the development of data types in ASF+SDF. This framework supports the hierarchical construction of new data types by reusing functionality from base data types. A layered structure is introduced to restrict and extend the set of operations and to adapt the behavior of functions.

8.3.1 Extending Functionality

Different behavior, due to new properties, motivates the derivation of a new data type from an existing one. A derived data type is more restricted than its base data type as a result of its new properties. For example, from the common data type Bag (or multi-set) we can derive the data types List and Set. The first is restricted because the order of its elements becomes relevant, whereas the latter is restricted since it requires uniqueness of its elements.

Due to the new properties, new functions become useful. The operation `head` (that returns the first element of a list), for instance, does make sense in a list but is meaningless in its base data type Bag.

Creating a new data type in ASF+SDF by reusing a (renamed) base data type and extending the functionality (by declaring new operations) cannot be done in a single module. Therefore a separate module M is required that imports the renamed data type and declares new functions. The exported signature of the module M (Exp(M)) forms the extension of functionality with respect to its base data type.

It is the rather restricted form of renaming that we have introduced that makes it necessary to implement the extended functionality in a separate module. For example, in ASF (Bergstra *et al.*, 1989) and SDF2 (Visser, 1997) the renaming is part of the import mechanism. New functionality and the renaming on the base data type can therefore be specified in a single module.

8.3.2 Restricting Functionality

Since a base data type does enforce less restrictions on its elements (because of the smaller set of properties), it often contains operations that are meaningless in (or do not belong to) the derived data type. This corresponds to operations that result in conflicting properties.

Example 8.3.3 Consider the data type List that contains a sequence of elements $e_0 \ldots e_{n-1}$. Suppose that the function `reverse` is defined on lists and satisfies the following property:

$$\forall_{i<n}\langle e_i = x \in L \rightarrow e_{n-i-1} = x \in reverse(L)\rangle \tag{8.1}$$

It would probably be a good choice to derive from List the data type SortedList in which the elements are ordered. Suppose that the elements are ordered in descending order, which is stated by the following property:

$$\forall_{i<n-1}\langle e_i < e_{i+1}\rangle \tag{8.2}$$

Both properties cannot be satisfied for $n > 1$. The operation `reverse` should therefore be removed from the signature of SortedList. □

In practice, the existence of conflicting properties often motives the construction of a new hierarchy for the new data type. For example, in the PSF library (Van Wamel, 1993) the data types Strings, Queue and Table are implemented as separate modules, although, at least part of their functionality overlaps. The library of Larch (Guttag and Horning, 1993) as another example, defines the containers Set, Bag, Stack and Queue as separate abstract data types. Both formalisms have constructed their libraries according to the following observation:

Observation 8.3.4 When, given a renaming ρ and two data types D and D' the following holds:

$$\Sigma(\rho(D)) \subseteq \Sigma(D')$$

then the data type D' can be based on the data type D. □

In the Larch library, the signatures of the data types Deque (not Dequeue) and List and the data types String and Sequence are ordered this way. Hence, List is derived from Deque and Sequence is based on String.

In order to develop a library that maximizes its hierarchical structure, we should be able to *restrict* the visible signature with respect to the operations from the imported signature (i.e., $\mathrm{Vis}(M) \subseteq \mathrm{Exp}(M) \cup \mathrm{Imp}(M)$). Many formalisms (like ASF (Bergstra *et al.*, 1989) and OPAL (Pepper, 1991)) however, export their imported signatures automatically, making it hard to restrict the functionality that is derived.

When an operation is removed from the visible signature, we should note that the operation cannot be removed from the internal signature since operations higher in the hierarchy may depend on them. Hence, the operation will not be removed from the specification completely but becomes an *internal* (or *implementation dependent*) function. Due to the use of internal functions properties of a data type might by violated temporarily in operations defined higher in the hierarchy.

Example 8.3.5 In a sorted list (as in Example 8.3.3) the function `reverse` (that violates the ordering property 8.2) might be used internally in the implementation. The function `last` for example, returning the last element of a list, might be implemented using `reverse` as follows:

```
[01] last( SL ) = first( reverse( SL ) )
```

The function `reverse` is used here as internal function. Hence, the properties of SortedList are violated only temporarily. Whenever the function `last` returns, all properties are again satisfied. □

Thus, although the function `reverse` should be removed from the visible signature of SortedList, it should not be removed from the internal signature. Hence, to restrict the visible signature of a new data type D_n, derived from D, with respect to a function f, a renaming ρ should be applied that satisfies:

Property 8.3.6 (Restricting Visible Signature)

$$f \in \mathrm{Vis}(D) \rightarrow \rho(f) \in \mathrm{Hid}(\rho(D))$$

□

In SDF2 *hidden imports* are introduced. As a result, the complete signature of a module can be imported without being exported. Moreover, two equal productions, of which one is declared as a hidden production, are merged into a single hidden production in SDF2. Thus, hidden imports and double production declarations give the user fine control over which imported productions should be exported and which should be removed from the visible signature.

In ASF+SDF no hidden imports exists. An imported signature will therefore be exported automatically and as a result, Property 8.3.6 cannot be satisfied. In our effort of designing a hierarchical library for ASF+SDF, we have experimented with a convention to simulate the removal of a production from the visible signature.

Convention 8.3.7 (Restricting Visible Signature in ASF+SDF) Consider a module hierarchy rooted at N, a function f and a renaming ρ that constructs the module M from N. Furthermore, consider:

$$\begin{array}{rcl} f & \in & \mathrm{Vis}(N), \text{ and} \\ f \Rightarrow @M\text{-}f & \in & \rho \end{array}$$

then (by convention), we assume the following to hold afterwards:

$$f \notin \mathrm{Vis}(\rho(N))$$

□

These renamings are of course obsolete in SDF2 since the production f can be removed from the visible signature formally by copying the declaration of f to a hidden section.

8.3.3 Adapting Functionality

Typically, in a hierarchical environment several data types will share a subset of their operations. However, due to the difference of the data types, the behavior of these operations may differ.

Example 8.3.8 Consider a hierarchical structure of data types in which the data type Bag forms the base for the data types Set and SortedList. These data types require a different behavior of the operation `insert` that is derived from Bag. In a set, each element occurs only once and the `insert` operation should therefore preserve this uniqueness constraint. In a sorted list on the other hand, the correct ordering of the elements should be maintained by the insert operation. □

In order to support multiple implementations for the same operation in different derived data types, we will simulate the Object Oriented approach of *deferred methods* (sometimes called *generic methods*; in C++ called *pure virtual methods*). In a superclass the behavior of a deferred method is essential null, a place holder, and *all* useful activity is defined as part of the overriding method (Budd, 1991). Such methods are called *abstract* in the programming language Java. We will introduce the term *incomplete* for similar functions that are declared but that are defined in a derived data type. We will define functions to be *complete* when a definition exists for a function. Similar to virtual classes in C++ and abstract classes in Java, we will use the term *incomplete data type* to denote data types that contain incomplete functions. When a data type does not contain incomplete functions it will be called a *complete data type.*

During the design of a data type we thus have to determine for which operations the behavior remains *fixed* and for which operations the behavior might differ. Besides parameter functions that depends on a particular instantiation of the data type we thus also have functions that depend on particular derivations. Hence, we divide the set of operations of a data type in three subsets:

- *Complete Functions.* The set of functions for which the behavior remains fixed in derived data types and in different instances.
- *Incomplete Functions.* The set of functions that are declared in some base data type, but that are defined differently in derived data types.
- *Parameter Functions.* The set of functions that will be instantiated when a particular instance of the data type is used in a specification.

When data types are defined according to a hierarchical structure, the definition of a data type can be used for two purposes. First, a data type can be used as *complete data type.* This corresponds to the use of an instance of an object in object-oriented terminology. Before a data type can be used as complete data types, all incomplete functions have to be defined. Note the similarity with object-oriented languages like C++, in which the instantiation of a class that contains pure virtual methods is not permitted. Secondly, a data type can be used as *incomplete data type*. This corresponds to the use of a class as super class in the object-oriented world. When a data type is used as incomplete data type several incomplete functions may exist.

Example 8.3.9 The data type List will contain several *complete functions*. The functions `empty?` and `length` are typical examples of complete functions. When List is used as base data type for SortedList, the function `insert` needs to be defined as *incomplete function*. The use of List as *complete data type* corresponds to the use of a particular instance of the data type. For example, it could have been instantiated with naturals to obtain a list of natural numbers. The function `insert` will have been defined when the data type is used as complete data type. Using the data type as an *incomplete data type* corresponds to the use of List as base data type. For example, it can be used as base for the data type SortedList. When SortedList is used as *complete data type*, the `insert` operation has been defined such that the ordering of its elements will be preserved (Example 8.3.8).

To determine where a new element should be inserted in a sorted list, a function is required to determine the relative order of two elements. Since the relative order of two elements depends on the type of the elements (which is a *sort parameter* of SortedList), this function should be a parameter of SortedList as well. This is what we call a *parameter function*. □

In the next section we will develop a framework for the definition of data types that accounts for their use as complete or incomplete data types.

8.4 Deriving a New Module

Restricting or extending the functionality of a data type and changing the behavior of some of its functions, motivates the derivation of a new data type. These factors have been described in detail in the previous section. This section concentrates on the construction of new data types in ASF+SDF taking these factors into account. As explained before, the result of a renaming in ASF+SDF is stored in a separate module that cannot contain any new sort or production declarations. As a consequence, several modules are required in order to restrict and extend the derived signature or to adapt the behavior of derived functions. For the correct derivation of a new (parameterized) data type, we introduce therefore a *layered module structure* as depicted in Figure 8.1. The next paragraphs will concentrate on these three layers separately.

8.4.1 Import Layer

The first layer in Figure 8.1 is the *Import Layer*. This layer contains only renamings on the base data type. It contains all renamings required to obtain the desired imported signature for the new data type. The renamings serve two purposes. First, the renamings are used to modify the derived syntax using ordinary symbol and production renamings. For example, when the data types

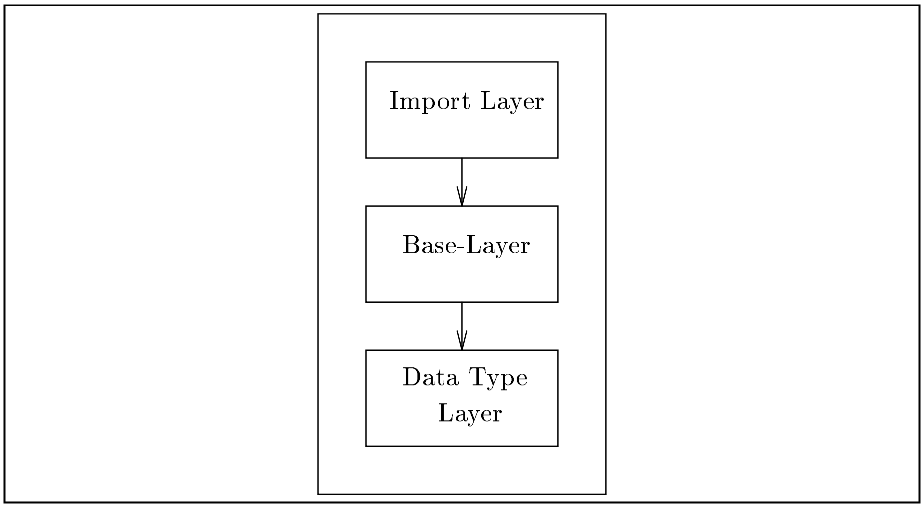


Figure 8.1: Layered Module Structure of a Data Type.

Queue and Stack are derived from List, this layer would most likely rename the operation `insert` to `enqueue` and `push` respectively. The second purpose of this layer is to restrict the functionality by removing functions from the visible signature. These are ordinary renamings too, but they should follow the convention described in §8.3.2.

We suggest to standardize the module name of the Import Layer by combining the name of the data type and the suffix "import" together. Hence, for the library implemented in the next chapter, we will store the Import Layer of a data type D in the module *D-import*.

8.4.2 Base Layer

The Base Layer of a data type contains extensions with respect to its base data type. This layer is used to introduce new operations by introducing new parameter functions or by extending the sets of complete and incomplete functions. This layer contains definitions for the new complete functions. To move a function from the set of incomplete functions to the set of complete functions its definition may be included in this layer as well. Since this layer contains only extensions and definitions, it is an ordinary module and not a renaming. This layer imports the module from the Import Layer as depicted in Figure 8.1. We introduce the convention to store the Base Layer of a data type D in the module *D-base*.

The Base Layer (together with the Import Layer) forms the *base* of a data type. That is, it defines the complete visible signature and it contains the implementation of all complete operations of the data type. This layer may still contain incomplete functions that have to be defined in the Data Type Layer (see the next section) or in a derived data type. The purpose of this layer corresponds therefore to the use of the data type as incomplete data type (§8.3.3). Thus, whenever a new data type D_n is derived from a data type D, the Base Layer of D will be used as base for D_n. The renamings defined in the Import Layer of D_n are thus defined on the base Layer of D.

8.4.3 Data Type Layer

The purpose of this layer is to complete the definition of a data type. In order to be able to instantiate a data type, this layer is required to define all remaining incomplete functions.

The only purpose of this layer is to define incomplete functions. Therefore, no new functions or sorts should be introduced. Furthermore, the Data Type Layer should contain only those equations that are required for the definition of incomplete functions. This layer imports the Base Layer as depicted in Figure 8.1. It remains empty when no incomplete functions have to be defined. We introduce the convention to store the Data Type Layer in the module D.

The renaming results in a module hierarchy rooted at D. It contains a set of parameter sorts and parameter functions and forms the definition of the parameterized data type D.

8.5 Representation Independence

The formalisms that we have studied (Larch (Guttag and Horning, 1993), PSF (Mauw and Veltink, 1990), ASF (Bergstra *et al.*, 1989), AFSL (Groenboom, 1997) and OPAL (Pepper, 1991)) all have a *fixed syntax*. As a result, the definition of a (parameterized) data type will "*standardize*" the syntax of that data type. In the PSF library (Van Wamel, 1993) for example, the representation of a queue is captured by the constructor function:

```
_ ~ _ : Q-ELEMENT # QUEUE -> QUEUE
```

Here, a queue is a sequence consisting of an element and a sequence separated by the symbol "~". In a formalism with fixed syntax the construction of a library is simplified because once a representation has been defined, only implementation issues remain.

In ASF+SDF however, we have *free* (or *user definable*) syntax. This free syntax results in the coexistence of many different representations of the same data type. Since in many cases the operations depend on the structure (or syntax) of a data type, many different implementations exist for different representations. Reuse of a data type is limited when its implementation depends on a particular representation. Avoiding *representation dependency* in a formalism with free syntax however, makes it difficult to develop a library of reusable modules. This difficulty has turned out to be one of the most important reasons for the lack of standardization and the absence of a library in the ASF+SDF formalism.

Definition 8.5.1 (Representation Dependence) In this text we will use the term *Representation Dependence* for functions that access the structure of a data type directly (i.e., they have precise knowledge of the syntax of the data type). We use the term *Representation Independence* for functions operating in a more generic way. Because they are not tied to a fixed syntax they can be used for several representations. □

Note that a data type that is defined representation independently promotes the reuse of its *operations* rather than its representation (i.e., its syntax), as is the case in a formalism with fixed syntax.

Example 8.5.2 (Representation Dependence) Lists are often represented in ASF+SDF using the list mechanism of the formalism. Thus, a typical representation would be a list of elements surrounded by brackets:

```
"[" Elem* "]" -> List
```

An advantage of this representation is that we can use *list matching* to find particular elements in the list. Implementing the function `lookup` for instance (that returns `true` when an element E is contained in the list and `false` otherwise), can be written in ASF+SDF very shortly using list matching as:

```
[01]          lookup( E, [E1* E E2*] ) = true
[default-02] lookup( E, [E1*] )        = false
```

Since this function accesses the list structure directly, it is representation dependent. The same function can therefore not be used for a recursive list representation using constructor functions:

```
cons( E1, cons( E2, empty-list ) )
```

□

Representation independence of a module can be obtained by using parameterized constructor and destructor functions (i.e., *access functions*). The representation dependent code in a module can be removed by using these access functions. The implementation can be bound to different representations by instantiating the access functions differently. The access functions have to be defined for each representation.

Definition 8.5.3 (Access Functions) For each representation of a data type a set of *access functions* $\mathcal{A}$ should be defined such that, by using only these functions, the data type can be constructed (i.e., elements can be inserted) and de-constructed (i.e., its elements can be retrieved).

Furthermore, we require the set $\mathcal{A}$ to be *minimal*, such that whenever a function f can be expressed using a subset of the operations from $\mathcal{A}$, then f does not belong to $\mathcal{A}$. □

Accessing the elements of a data type using these functions does not improve the performance of course. Moreover, the use of some language constructs (like list matching in the previous example) that improve the readability of a specification has to be avoided.

Example 8.5.4 (Representation Independence) In this example we will re-define the `lookup` function from Example 8.5.2 to make it representation independent.

First we define a set of generic access functions that enables us to access the elements of a list. The set consists of the constructor functions `empty-list` and `cons` and the destructor functions `first` and `rest`.

For the implementation of `lookup` we need the function `empty?` that tests whether a list is empty or not. The following definition of empty? is representation independent since it uses access functions only:

```
[01] empty?( L ) = true  when L  = empty-list
[02] empty?( L ) = false when L != empty-list
```

Below we define the function `lookup` representation independently:

```
[03] empty?( L ) = true
     ======================
     lookup( E, L ) = false

[04] empty?( L ) = false,
      first( L ) = E
     =====================
     lookup( E, L ) = true

[05] empty?( L ) = false,
     first( L ) != E
     ======================================
     lookup( E, L ) = lookup( E, rest( L ) )
```

Observe that in this example equivalence is used for testing equality. This can be seen as a representation dependency though, because depending on the elements, equality does not always imply equivalence. We will will ignore this in the example.

After the representation independent function `lookup` has been defined, we will now bind the definition to the list representations from Example 8.5.2. First we give the implementation of the access functions for the two list representations.

The following equations are used for the constructor based representation (note that the functions `cons` and `empty-list` are constructors for this representation and therefore no equations are defined for them):

```
[01] first( cons( E, L ) ) = E
[02]  rest( cons( E, L ) ) = L
```

For the representation based on the list construct of the ASF+SDF formalism we get the following equations:

```
[01]      empty-list = []
[02] cons( E, [E*] ) = [E E*]
[03] first( [E E*] ) = E
[04]  rest( [E E*] ) = [E*]
```

Finally, by instantiating, the module defining `lookup` can be bound to one of these representations. □

Consider two data types D_1 and D_2 that cannot be expressed using the same set of access functions. According to Definition 8.5.3 we should note that it is not possible to construct the data type D_2, or to retrieve its elements using the access functions defined for D_1. This resulted in the observation that the sets of access functions can be used to determine whether or not two data types can be included in the same hierarchy:

Observation 8.5.5 (Forest of Data Types) Two data types that cannot be expressed by the same set of access functions should be included in different hierarchies of data types. □

This observation introduces the notion of different hierarchies of data types (i.e., data types that can be expressed using the same set of access functions should be included in the same hierarchy). Examples of different hierarchies of data types include *Bag-Like* (including sets, lists and strings) and *tree-like* data types (including binary trees and terms). The names of these hierarchies were inspired by Didrich (1994) in which data types are classified as ProductLike, UnionLike, SeqLike, TreeLike, SetLike and MapLike data types. In the next chapter we will implement a *Bag-Like* hierarchy of data types. Observation 8.5.5 was used to select the data types that we have included in the hierarchy and to determine for which data types a new hierarchy should be constructed.

Chapter 9

Case Study: a Library for ASF+SDF

9.1 Introduction

In this chapter we will use the techniques from the previous chapters to develop a set of reusable data types for ASF+SDF. Together they form (part of) a library for ASF+SDF. This library consist of a very limited number of data types. Its main purpose is to experiment with renaming and with the ideas discussed in Chapter 8. Furthermore, the functionality of the data types is incomplete because many operations are missing. The operations that are implemented are used only to verify that the construction of a library is possibly. However, despite its limitations, the library is a first attempt for the development of a extensive library defining many data types that can be reused (by instantiation) in many different specifications.

The goal of the development is to maximize the reuse of modules within the library and to maximize its hierarchical structure. We are therefore not concerned with any performance issues. The library as far as we have implemented it, is limited to a single hierarchy of data types only (classified as Bag-Like according to §8.5). Data types that are contained in this hierarchy include: Bags, Maps, Strings and Sets. Their hierarchical structure is depicted in Figure 9.1. The definition of other hierarchies including data types like Trees and Terms, has still to be investigated.

From the figure we see that we use three data types that are not included in the hierarchy of Bag. The first two data types (Boolean and Integer) are included in the ATerm library (Van den Brand *et al.*, 1997c). Since these data types are already reused in many specifications at the moment, we will reuse that implementation rather than re-define them. For the Integer data type however, this forms a deviation from our design strategy. When maximizing the hierarchical structure of the library, the data type should be considered as a sequence of digits. Integers should therefore have been defined as instance

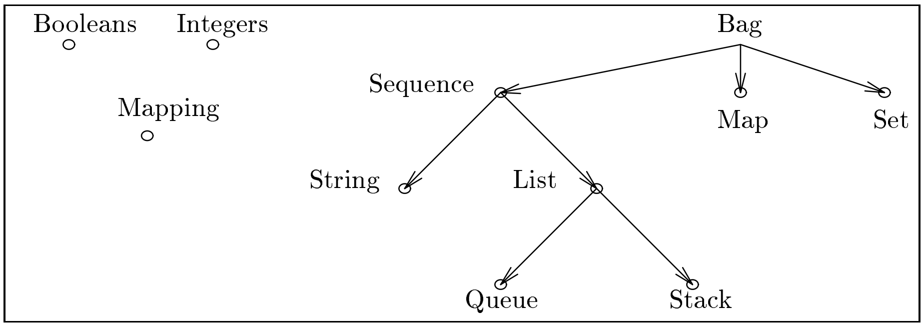


Figure 9.1: Overview of the hierarchical structure of the modules contained in the library that is developed in this chapter.

of the data type Sequence. The third data type (Mapping) is required for the instantiation of the data type Map defined in §9.4. The remaining data types in the figure are defined in the next sections.

9.2 Bag

In this section we will define the data type Bag that forms the base data type for the hierarchy. A Bag is a collection without internal structure in which elements may occur more then once (Uhl and Schmid, 1990). Another data type that is similar to Bag is Multi-Set.

Because Bag forms the base of the hierarchy and therefore is not derived from a data type itself, it is the only data type without an Import Layer. Moreover, because Bag is the base data type for this hierarchy of data types, it defines the set of access functions. We will therefore deviate slightly from the layered structure introduced in §8.4 for the definition of Bag.

According to Observation 8.5.5 all data types contained in a single hierarchy can be expressed by the same set of access functions. Hence, before defining the complete signature of Bag we first define a minimal set of access functions.

A Bag-Like data type is a collection of elements. Depending on the data type, its elements may or may not occur in a particular order. Internally, we will represent a Bag (and all its derived data types) as a sequence of elements. The elements thus occur internally in a fixed order. The set of access functions should therefore provide at least the following operations: 'empty' for the construction of a new (empty) data type, 'cons' to create a new data type from an existing one by putting a new element in front, 'first' to obtain the first element in the data type and 'rest' to obtain all elements except the one in front.

Obviously, this set of operations is sufficient to construct any Bag-Like data type and to retrieve elements from it. Furthermore, it is not difficult to see that none of these operations is superfluous (i.e., Definition 8.5.3 cannot be satisfied when one of these operations is missing). Hence, this set of access functions is minimal.

Let us now define the other operations that are provided by Bag. We define the function 'is-empty' to test for emptiness. It returns 'true' when the Bag is empty and 'false' otherwise. The function 'size' returns the number of elements that are included in the Bag, 'contains' returns 'true' when a particular element is included in the Bag and 'false' otherwise. Finally, we define the operation 'insert' for the insertion of an element in the Bag. This operation is defined as an incomplete function because the behavior of the insertion function will differ among different derived data types. Hence, the Base Layer of the data type Bag is defined as follows:

Module Bag-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`size, is-empty, contains`
incomplete functions:	`insert`

imports Layout Booleans Integers
exports
sorts Elem Bag
context-free syntax
"empty" → Bag

"cons" "(" Elem "," Bag ")" → Bag
"first" "(" Bag ")" → Elem
"rest" "(" Bag ")" → Bag

"is-empty" "(" Bag ")" → Bool
"size" "(" Bag ")" → Int
"contains" "(" Bag "," Elem ")" → Bool
"insert" "(" Bag "," Elem ")" → Bag
hiddens
variables
B → Bag
E → Elem
equations

[01] $$\frac{B \;=\; \text{empty}}{\text{is-empty}(B) \;=\; \text{true}}$$

[02] $$\frac{B \;\neq\; \text{empty}}{\text{is-empty}(B) \;=\; \text{false}}$$

[03] $$\frac{\text{is-empty}(B) \;=\; \text{true}}{\text{size}(B) \;=\; 0}$$

$$[04] \quad \frac{\text{is-empty}(B) \;=\; \text{false}}{\text{size}(B) \;=\; 1 + \text{size}(\text{rest}(B))}$$

$$[05] \quad \frac{\text{is-empty}(B) \;=\; \text{true}}{\text{contains}(B,\, E) \;=\; \text{false}}$$

$$[06] \quad \frac{\begin{array}{c}\text{is-empty}(B) \;=\; \text{false}, \\ \text{first}(B) \;=\; E\end{array}}{\text{contains}(B,\, E) \;=\; \text{true}}$$

$$[07] \quad \frac{\begin{array}{c}\text{is-empty}(B) \;=\; \text{false}, \\ \text{first}(B) \;\neq\; E\end{array}}{\text{contains}(B,\, E) \;=\; \text{contains}(\text{rest}(B),\, E)}$$

Because the Base Layer of Bag defines the function 'insert' as incomplete function, the Data Type Layer of Bag that defines all incomplete functions has to contain the implementation of this operation. Hence, the Data Type Layer of Bag becomes:

Module Bag

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`size, is-empty, contains, insert`
incomplete functions:	

imports Bag-base
hiddens
  variables
    $B \rightarrow$ Bag
    $E \rightarrow$ Elem
equations

[01] $\text{insert}(B,\, E) = \text{cons}(E,\, B)$

9.3 Set

Like a Bag, a Set is a collection without internal structure. However, in a Set each element occurs exactly once. Therefore, Bag will be used as base for Set but we define the incomplete function 'insert' to insert an element only when that elements is not already contained in the Set.

A Set does not limit the set of operations derived from Bag. The only renaming that is required in the Import Layer is therefore the renaming that renames the sort 'Bag' to 'Set':

Set-import.ren:

```
Bag-base [
   Bag => Set
]
```

The Base Layer of Set extends the derived signature with the following operations: The operation 'setminus' returns the difference of two Sets $\mathcal{A}$ and $\mathcal{B}$. (i.e., $\mathcal{A} \setminus \mathcal{B}$). 'setunion' returns the union of two Sets (i.e., $\mathcal{A} \cup \mathcal{B}$), 'setintersect' returns the intersection of two Sets (i.e., $\mathcal{A} \cap \mathcal{B}$). Furthermore, the Base Layer of Set implements the incomplete function 'insert'. This function will only insert an element when that element is not already contained in the Set.

Module Set-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`size, is-empty, contains, insert, setminus, setunion, setintersect`
incomplete functions:	

imports Set-import
exports
  context-free syntax
    "setminus" "(" Set "," Set ")" → Set
    "setunion" "(" Set "," Set ")" → Set
    "setintersect" "(" Set "," Set ")" → Set
hiddens
  variables
    E [*0-9*]∗ → Elem
    S [*0-9*]∗ → Set
equations

$$[01] \quad \frac{\text{contains}(S, E) \ = \ \text{true}}{\text{insert}(S, E) \ = \ S}$$

$$[02] \quad \frac{\text{contains}(S, E) \ = \ \text{false}}{\text{insert}(S, E) \ = \ \text{cons}(E, S)}$$

$$[03] \quad \frac{\text{is-empty}(S_1) \ = \ \text{true}}{\text{setminus}(S_1, S_2) \ = \ \text{empty}}$$

$$[04] \quad \frac{\begin{array}{rcl}\text{is-empty}(S_1) & = & \text{false},\\ \text{contains}(S_2, \text{first}(S_1)) & = & \text{true}\end{array}}{\text{setminus}(S_1, S_2) \ = \ \text{setminus}(\text{rest}(S_1), S_2)}$$

$$[05] \quad \frac{\begin{array}{rcl}\text{is-empty}(S_1) & = & \text{false},\\ \text{contains}(S_2, \text{first}(S_1)) & = & \text{false}\end{array}}{\text{setminus}(S_1, S_2) \ = \ \text{insert}(\text{setminus}(\text{rest}(S_1), S_2), \text{first}(S_1))}$$

$$[06] \quad \frac{\text{is-empty}(S_2) \ = \ \text{true}}{\text{setunion}(S_1, S_2) \ = \ S_1}$$

$$[07] \quad \frac{\begin{array}{rcl}\text{is-empty}(S_2) & = & \text{false},\\ \text{contains}(S_1, \text{first}(S_2)) & = & \text{true}\end{array}}{\text{setunion}(S_1, S_2) \ = \ \text{setunion}(S_1, \text{rest}(S_2))}$$

$$[08] \quad \frac{\begin{array}{rcl}\text{is-empty}(S_2) & = & \text{false},\\ \text{contains}(S_1, \text{first}(S_2)) & = & \text{false}\end{array}}{\text{setunion}(S_1, S_2) \ = \ \text{insert}(\text{setunion}(S_1, \text{rest}(S_2)), \text{first}(S_2))}$$

$$[09] \quad \text{setintersect}(S_1, S_2) = \text{setminus}(S_1, \text{setminus}(S_1, S_2))$$

Because the Base Layer does not contain any incomplete functions, the Data Type Layer of the data type Set remains empty:

Module Set

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`size, is-empty, contains, insert, setdiff, setunion, setintersect`
incomplete functions:	

imports Set-base

9.4 Map and Mapping

Maps are representations of finite functions from a "*key*" to an "*info*" type (Uhl and Schmid, 1990). Like Bags, Maps are collections without internal structure.

In order to structure the elements of the base data type Bag in a Key and Value, we use the data type Mapping to instantiate the parameter sort 'Elem'. This data type is used to represent mappings between Keys and Values as

Key/Value pairs. The data type contains only parameter sorts and functions. For simplicity we have deviated from the layered model by defining Mapping in a single module only:

Module Mapping

parameter sorts:	`Key, Value, Mapping`
parameter functions:	`key-of, value-of, mapping`
complete functions:	
incomplete functions:	

imports Layout
exports
  sorts Mapping Key Value
  context-free syntax
    "key-of" "(" Mapping ")" → Key
    "value-of" "(" Mapping ")" → Value
    "mapping" "(" Key "," Value ")" → Mapping

Next, we replace (by instantiation) the parameter sort 'Elem' from Bag by 'Mapping'. After this instantiation, the data type Map contains three parameter sorts ('Key', 'Value' and 'Mapping' that represents the mapping between them). For a Map we will define the behavior of the function 'contains' as follows:

$$contains(v, M) = \begin{cases} true & \text{whenever } \exists_k \langle k \rightarrow v \in M \rangle, \\ false & \text{otherwise.} \end{cases}$$

Observe, that (after renaming) the derived function declaration of 'contains' has 'Mapping' as its first argument sort. However, according to the definition of 'contains' above, the function should have 'Value' as its first argument sort. We can therefore not use the operation 'contains' from the base data type Bag. As a consequence, we will remove the function from the imported signature and give a new definition in the Base Layer of the data type.

To summarize, the Import Layer of Map contains the renaming of the sort Bag, the instantiation of the sort 'Elem' and the removal of the function 'contains' (that will be renamed according to Convention 8.3.7):

Map-import.ren:

```
Bag-base [
    Bag => Map
   Elem => Mapping <Mapping>

        "contains" "(" Bag "," Elem ")"    -> Bag =>
   "@Map-contains" "(" Map "," Mapping ")" -> Map
]
```

The Base Layer of the data type Map contains the implementation of the function 'contains' and a new declaration with correct argument types. Remember that the derived declaration has been removed and that 'contains' is therefore not *overloaded*. Finally, a new function 'map' is defined, that gives the value corresponding to a given key.

Module Map-base

parameter sorts:	`Key, Value, Mapping`
parameter functions:	`key-of, value-of, mapping, empty, cons, first, rest`
complete functions:	`size, is-empty, map, contains`
incomplete functions:	`insert`

imports Map-import
exports
  context-free syntax
    "contains" "(" Map "," Value ")" → Bool
    "map" "(" Map "," Key ")" → Value
hiddens
  variables
    M → Map
    K → Key
    V → Value
equations

$$[01]\quad \frac{\text{is-empty}(M) \;=\; \text{true}}{\text{contains}(M,\ V) \;=\; \text{false}}$$

$$[02]\quad \frac{\begin{array}{c}\text{is-empty}(M) \;=\; \text{false},\\ \text{value-of}(\text{first}(M)) \;=\; V\end{array}}{\text{contains}(M,\ V) \;=\; \text{true}}$$

$$[03]\quad \frac{\begin{array}{c}\text{is-empty}(M) \;=\; \text{false},\\ \text{value-of}(\text{first}(M)) \;\neq\; V\end{array}}{\text{contains}(M,\ V) \;=\; \text{contains}(\text{rest}(M),\ V)}$$

$$[04]\quad \frac{\begin{array}{c}\text{is-empty}(M) \;=\; \text{false},\\ \text{key-of}(\text{first}(M)) \;=\; K\end{array}}{\text{map}(M,\ K) \;=\; \text{value-of}(\text{first}(M))}$$

$$[05]\quad \frac{\begin{array}{c}\text{is-empty}(M) \;=\; \text{false},\\ \text{key-of}(\text{first}(M)) \;\neq\; K\end{array}}{\text{map}(M,\ K) \;=\; \text{map}(\text{rest}(M),\ K)}$$

Note that the function 'insert' remains an incomplete function, because several different types of mappings might require different behavior of the function.

In the Data Type Layer of Map we only have to implement the function 'insert'. This function adds a new Key/Value pair or changes the value of an existing pair:

Module Map

parameter sorts:	`Key, Value, Mapping`
parameter functions:	`key-of, value-of, mapping, empty, cons, first, rest`
complete functions:	`size, is-empty, map, insert, contains`
incomplete functions:	

imports Map-base
hiddens
variables
M → Map
V → Value
P → Mapping
equations

$$[01] \quad \frac{\text{is-empty}(M) \ = \ \text{true}}{\text{insert}(M,\, P) \ = \ \text{cons}(P,\, M)}$$

$$[02] \quad \frac{\begin{array}{rcl}\text{is-empty}(M) & = & \text{false},\\ \text{key-of}(P) & = & \text{key-of}(\text{first}(M))\end{array}}{\begin{array}{l}\text{insert}(M,\, P) \ = \\ \text{cons}(\text{mapping}(\text{key-of}(\text{first}(M)),\, \text{value-of}(P)),\, \text{rest}(M))\end{array}}$$

$$[03] \quad \frac{\begin{array}{rcl}\text{is-empty}(M) & = & \text{false},\\ \text{key-of}(P) & \neq & \text{key-of}(\text{first}(M))\end{array}}{\text{insert}(M,\, P) \ = \ \text{cons}(\text{first}(M),\, \text{insert}(\text{rest}(M),\, P))}$$

9.5 Sequence

The data type Sequence is a collection in which the elements are ordered *in sequence*. We can therefore associate with each element a unique index. New operations can be defined that use this index to access and modify individual elements.

Because the elements are ordered successively we associate a *length* with the data type. We will therefore rename the operation 'size' derived from the data type Bag to 'length'. Sequence does not limit the derived set of operations. The renamings that form the Import Layer of the data type are therefore defined as follows:

Sequence-import.ren:

```
Bag-base [
   Bag => Sequence

     "size" "(" Bag ")"       -> Int =>
   "length" "(" Sequence ")" -> Int
]
```

Several new operations can be defined that use the index of the elements. In this case study we will only introduce two new functions. The first function is 'concat' that returns the concatenation of two sequences, the second function reverses the order of the elements. These functions are defined in the Base Layer of Sequence:

Module Sequence-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`length, is-empty, contains, concat, reverse`
incomplete functions:	`insert`

imports Sequence-import
exports
  context-free syntax
    "concat" "(" Sequence "," Sequence ")" → Sequence
    "reverse" "(" Sequence ")" → Sequence
hiddens
  context-free syntax
    "concat1" "(" Sequence "," Sequence ")" → Sequence
    "reverse1" "(" Sequence "," Sequence ")" → Sequence
  variables
    S *[0-9]*$*$ → Sequence
    I → Int
    E → Elem
equations

[01] $\text{concat}(S_1, S_2) = \text{concat1}(\text{reverse}(S_1), S_2)$

[02] $\text{concat1}(S_1, S_2) = S_2$ **when** $\text{is-empty}(S_1) = \text{true}$

[03] $$\frac{\text{is-empty}(S_1) = \text{false}}{\text{concat1}(S_1, S_2) = \text{concat1}(\text{rest}(S_1), \text{cons}(\text{first}(S_1), S_2))}$$

[04] $\text{reverse}(S) = \text{reverse1}(S, \text{empty})$

[05] $$\frac{\text{is-empty}(S_1) \;=\; \text{true}}{\text{reverse1}(S_1,\, S_2) \;=\; S_2}$$

[06] $$\frac{\text{is-empty}(S_1) \;=\; \text{false}}{\begin{array}{l}\text{reverse1}(S_1,\, S_2) \;=\\ \text{reverse1}(\text{rest}(S_1),\, \text{cons}(\text{first}(S_1),\, S_2))\end{array}}$$

In order to complete the definition of the data type Sequence, the Data Type Layer contains the implementation of the 'insert' operation:

Module Sequence

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`insert, length, is-empty, contains, concat, reverse`
incomplete functions:	

imports Sequence-base
hiddens
  variables
    S → Sequence
    E → Elem
equations

[01] $\text{insert}(S, E) = \text{cons}(E, S)$

9.6 List

Like in a sequence, the elements of lists have a linear structure with a first and a last element. The signature of List is not restricted with respect to the derived signature of Sequence. Hence, the Import Layer of List contains only the renaming that renames 'Sequence' to 'List':

List-import.ren:

```
Sequence-base [
   Sequence => List
]
```

List extends the functionality of Sequence by the operation 'remove' for the removal of individual elements from a list. This operation will be defined as an incomplete function because the derived data types Stack and Queue require different implementations. As a consequence, the Base Layer of List contains only the declaration of the operation:

Module List-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`length, is-empty, contains, concat, reverse`
incomplete functions:	`insert, remove`

imports List-import
exports
  context-free syntax
    "remove" "(" List "," Elem ")" → List

For the completion of the definition of List, the Data Type Layer contains the definition of the operations 'insert' and 'remove'.

Module List

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`length, is-empty, contains, concat,` `reverse, insert, remove`
incomplete functions:	

imports List-base
hiddens
  variables
    *L [0-9]** → List
    E → Elem
    I → Int
equations

[01] $\text{insert}(L, E) = \text{cons}(E, L)$

[02] $$\frac{\text{is-empty}(L) \ = \ \text{true}}{\text{remove}(L, E) \ = \ \text{rest}(L)}$$

[03] $$\frac{\text{is-empty}(L) \ = \ \text{false}, \quad \text{first}(L) \ = \ E}{\text{remove}(L, E) \ = \ \text{rest}(L)}$$

$$[04]\quad \frac{\begin{array}{r}\text{is-empty}(L) \;=\; \text{false},\\ \text{first}(L) \;\neq\; E\end{array}}{\begin{array}{l}\text{remove}(L, E) \;=\\ \text{cons}(\text{first}(L), \text{remove}(\text{rest}(L), E))\end{array}}$$

9.7 Stack

The data type stack is a *last in, first out* (LiFo) data type. Basically a stack is a list with a restricted set of insert and remove operations (Uhl and Schmid, 1990). Hence, we derive Stack from List and remove the operations 'length', 'contains', 'reverse', 'remove' and 'concat' from the signature. Observe that it does not make sense for the representation independent definition of Stack to rename the operations 'insert', 'first' and 'rest' to 'push', 'top' and 'pop' respectively. Hence, we define Stack using the names of the derived operations:

Stack-import.ren:

```
List-base [
   List => Stack

           "length" "(" List ")" -> Int =>
    "@Stack-length" "(" Stack ")" -> Int

         "contains" "(" List "," Elem ")" -> Bool =>
  "@Stack-contains" "(" Stack "," Elem ")" -> Bool

           "remove" "(" List "," Elem ")"  -> List =>
    "@Stack-remove" "(" Stack "," Elem ")" -> Stack

           "concat" "(" List "," List ")"  -> List =>
    "@Stack-concat" "(" Stack ","Stack ")" -> Stack

          "reverse" "(" List  ")" -> List =>
   "@Stack-reverse" "(" Stack ")" -> Stack
]
```

We do not suppose that Stack will serve as base for another data type. The incomplete function 'insert' is therefore defined in the base Layer of Stack:

Module Stack-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`is-empty, insert`
incomplete functions:	

imports Stack-import
hiddens
  variables
    S *[0-9]*$*$ $\rightarrow$ Stack
    E $\rightarrow$ Elem
equations

[01] $\text{insert}(S, E) = \text{cons}(E, S)$

The Data Type Layer of Stack remains empty because all incomplete functions have already been defined:

Module Stack

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`is-empty, insert`
incomplete functions:	

imports Stack-base

9.8 Queue

The data type Queue is a *first in, first out* (FiFo) data type. Like stacks, also queues are lists with a restricted set of insert and remove operations. Like the data type Stack, we will derive Queue from List and remove the operations 'length', 'contains', 'reverse', 'remove' and 'concat' from the signature. Observe that we do not rename the names of derived operations:

Queue-import.ren:

```
List-base [
   List => Queue

           "length" "(" List ")" -> Int =>
    "@Queue-length" "(" Queue ")" -> Int

         "contains" "(" List "," Elem ")" -> Bool =>
  "@Queue-contains" "(" Queue "," Elem ")" -> Bool

           "remove" "(" List "," Elem  ")" -> List =>
    "@Queue-remove" "(" Queue "," Elem ")" -> Queue

           "concat" "(" List "," List  ")" -> List =>
```

```
       "@Queue-concat" "(" Queue "," Queue ")" -> Queue

             "reverse" "(" List  ")" -> List =>
      "@Queue-reverse" "(" Queue ")" -> Queue
   ]
```

In this case study we will define the incomplete operation 'insert' in the Base Layer of Queue because we do not consider Queue as base for another data type. This operations inserts an element at the back of the queue:

Module Queue-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`is-empty, insert`
incomplete functions:	

imports Queue-import
hiddens
  variables
    $Q \rightarrow$ Queue
    $E \rightarrow$ Elem
equations

$$[01] \quad \frac{\text{is-empty}(Q) \ = \ \text{true}}{\text{insert}(Q, E) \ = \ \text{cons}(E, Q)}$$

$$[02] \quad \frac{\text{is-empty}(Q) \ = \ \text{false}}{\text{insert}(Q, E) \ = \ \text{cons}(\text{first}(Q), \text{insert}(\text{rest}(Q), E))}$$

The Data Type Layer remains empty since no incomplete operations have to be defined:

Module Queue

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`is-empty, insert`
incomplete functions:	

imports Queue-base

9.9 String

As another example of a data type derived from Sequence, we will define the data type String. Since String is parameterized, it can be instantiated to construct for instance a string of characters or a string of bits. The Import Layer of String removes the operation 'insert' from the imported signature.

String-import.ren:

```
Sequence-base [
   Sequence => String

           "insert" "(" Sequence "," Elem ")" -> Sequence =>
   "@String-insert" "(" String "," Elem ")" -> String
]
```

Although many operations can be defined for strings, we will keep things simple and define only one new operation. This operation 'sub-string' verifies whether a substring is included in a string. This operation is declared and defined in the Base Layer of String:

Module String-base

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`length, is-empty, contains, concat,` `reverse, sub-string`
incomplete functions:	

imports String-import
exports
  context-free syntax
    "sub-string" "(" String "," String ")" $\rightarrow$ Bool
hiddens
  context-free syntax
    "sub-string1" "(" String "," String ")" $\rightarrow$ Bool
  variables
    S [*0-9*]$*$ $\rightarrow$ String
equations

$$[01]\quad \frac{\begin{array}{rcl}\text{is-empty}(S_1) & = & \text{false},\\ \text{sub-string1}(S_1, S_2) & = & \text{false}\end{array}}{\text{sub-string}(S_1, S_2) \;=\; \text{sub-string}(\text{rest}(S_1), S_2)}$$

$$[02]\quad \text{sub-string}(S_1, S_2) = \text{sub-string1}(S_1, S_2) \quad \textbf{otherwise}$$

$$[03]\quad \frac{\text{is-empty}(S_2) \;=\; \text{true}}{\text{sub-string1}(S_1, S_2) \;=\; \text{true}}$$

$$[04]\quad \frac{\begin{array}{rcl}\text{is-empty}(S_2) & = & \text{false},\\ \text{is-empty}(S_1) & = & \text{true}\end{array}}{\text{sub-string1}(S_1,\, S_2)\ =\ \text{false}}$$

$$[05]\quad \frac{\begin{array}{rcl}\text{is-empty}(S_2) & = & \text{false},\\ \text{is-empty}(S_1) & = & \text{false},\\ \text{first}(S_1) & = & \text{first}(S_2)\end{array}}{\text{sub-string1}(S_1,\, S_2)\ =\ \text{sub-string1}(\text{rest}(S_1),\, \text{rest}(S_2))}$$

$$[06]\quad \frac{\begin{array}{rcl}\text{is-empty}(S_2) & = & \text{false},\\ \text{is-empty}(S_1) & = & \text{false},\\ \text{first}(S_1) & \neq & \text{first}(S_2)\end{array}}{\text{sub-string1}(S_1,\, S_2)\ =\ \text{false}}$$

String does not introduce any incomplete functions, hence the Data Type Layer remains empty:

Module String

parameter sorts:	`Elem`
parameter functions:	`empty, cons, first, rest`
complete functions:	`length, is-empty, contains, concat,` `reverse, sub-string`
incomplete functions:	

imports String-base

9.10 Example

In this section we will develop a specification that includes two different instances of the data type Queue. The example demonstrates the use of the renaming tool, the mapping of representation independent data types to a particular representation and the instantiation of parameterized data types.

The data type Queue that we will use in the specification is a representation independent data type. We will therefore first define a representation of a Queue. Thereafter we will instantiate the access functions of Queue to bind the implementation of Queue to this particular representation.

In this example we will represent a Queue as a sequence of elements surrounded by '<' and '>'. Furthermore, we will use the operations 'enqueue', 'dequeue' and 'serve' to insert, remove and retrieve an element respectively.

To bind the implementation of the data type Queue to this representation, the access functions 'empty', 'cons', 'first' and 'rest' have to be defined. Observe the similarity between de functions 'first' and 'serve' (both returning the first element) and the functions 'rest' and 'dequeue' (that return all elements except the one in front). We will therefore implement the operations 'serve' and 'dequeue' and rename the operations 'first' and 'rest' in the implementation of Queue. The remaining access functions do not correspond directly to operations on queues but are required for the implementation of Queue. We will define the representation of our queue and its access function in a separate module Queue-repr:

Module Queue-repr

imports Layout
exports
  sorts Queue Elem
  context-free syntax
    "<" Elem$*$ ">" $\rightarrow$ Queue

    "empty" $\rightarrow$ Queue
    "cons" "(" Elem "," Queue ")" $\rightarrow$ Queue
    "serve" "(" Queue ")" $\rightarrow$ Elem
    "dequeue" "(" Queue ")" $\rightarrow$ Queue
hiddens
  variables
    "*Elem*$*$" $\rightarrow$ Elem$*$
    Elem $\rightarrow$ Elem
equations

[01] empty $= < \ >$
[02] cons($Elem$, $< Elem^* >$) $= < Elem\ Elem^* >$
[03] serve($< Elem\ Elem^* >$) $= Elem$
[04] dequeue($< Elem\ Elem^* >$) $= < Elem^* >$

After the representation has been defined we can instantiate the representation independent data type Queue to bind its implementation to our representation.

The following renaming suffices (observe the renamings of the access functions 'first' and 'rest' and the renaming of the function 'insert' to 'push':

MyQueue.ren:

```
Queue [
   Queue => Queue <Queue-repr>
    Elem => Elem  <Queue-repr>

   "empty" -> Queue =>
   "empty" -> Queue <Queue-repr>
```

```
        "cons" "(" Elem "," Queue ")" -> Queue =>
        "cons" "(" Elem "," Queue ")" -> Queue <Queue-repr>

       "first" "(" Queue ")" -> Elem =>
       "serve" "(" Queue ")" -> Elem <Queue-repr>

        "rest" "(" Queue ")" -> Queue =>
     "dequeue" "(" Queue ")" -> Queue <Queue-repr>

      "insert" "(" Queue "," Elem ")" -> Queue =>
     "enqueue" "(" Queue "," Elem ")" -> Queue
  ]
```

When this renaming is defined in the renaming module `MyQueue.ren` then the module MyQueue is created after the renaming has been applied. This module will contain the parameterized data type Queue using the representation as defined in the module Queue-repr.

The representation dependent data type Queue defined in MyQueue will be used in our specification for the construction of IntQueue and BoolQueue containing integer and boolean values respectively. IntQueue is constructed by the following instantiation of MyQueue defined in `IntQueue.ren`:

IntQueue.ren:

```
  MyQueue [
     Queue => IntQueue
      Elem => Int <Integers>
     "empty" -> Queue => "empty-intqueue" -> IntQueue
  ]
```

For the construction of BoolQueue we use a similar instantiation which we define in `BoolQueue.ren`

BoolQueue.ren:

```
  MyQueue [
     Queue => BoolQueue
      Elem => Bool <Booleans>
     "empty" -> Queue => "empty-boolqueue" -> BoolQueue
  ]
```

Observe the renamings of the constant 'empty' and the sort 'Queue'. These renamings are required to prevent ambiguities when both data types are included in a single specification.

The use of both data types in a single specification is demonstrated below. The corresponding import graph is depicted in Figure 9.2. In the figure we see the several instances of modules and their module identifiers. Furthermore, we see that modules are reused whenever possible.

Module Example

imports IntQueue BoolQueue
exports
  context-free syntax
    "use-queue" "(" IntQueue ")" $\rightarrow$ Int
    "use-queue" "(" BoolQueue ")" $\rightarrow$ Bool
hiddens
  variables
    IQ $\rightarrow$ IntQueue
    BQ $\rightarrow$ BoolQueue
equations

[01] use-queue(IQ) = serve(IQ)
[02] use-queue(BQ) = serve(BQ)

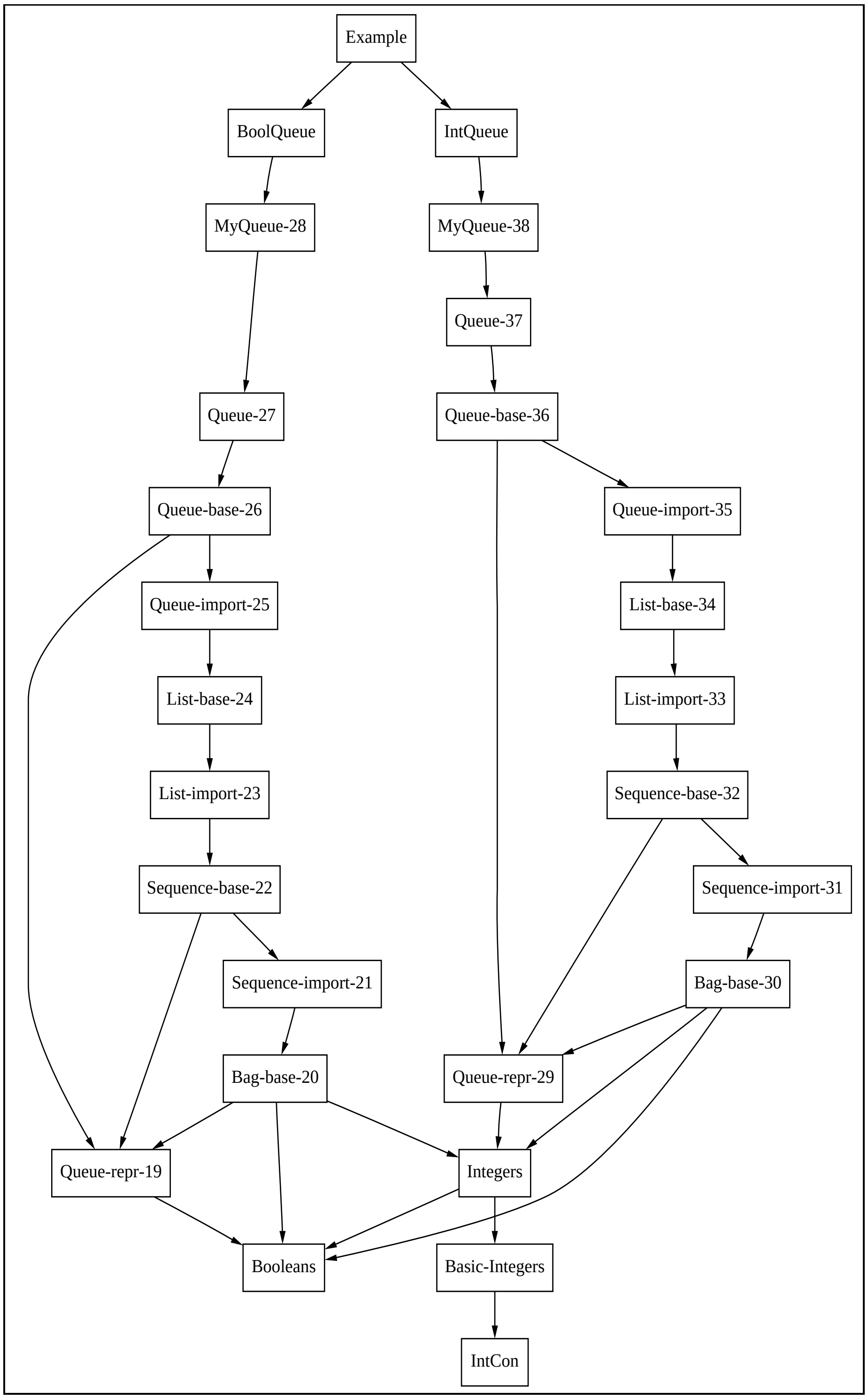


Figure 9.2: Import Graph for the specification defined in §9.10.

Chapter 10

Conclusions

In the previous chapters we have developed a framework for the development of hierarchical parameterized data types in ASF+SDF. The framework resulted from the informal ideas of structuring data types according to a set of properties associated with the data type. The framework was used in Chapter 9 for the development of a very limited hierarchy of data types. The binding of a data type to a particular representation and its instantiation in §9.10 demonstrated the use of the library in practice.

Together these chapters answered the question whether it is possible to develop a library of reusable components for ASF+SDF. However, the framework and the library have several drawbacks. First of all, we emphasize that the ideas of structuring data types are based on rather informal notions. They gave some intuition for the structuring of data types but they cannot be used in general.

Secondly, we mention that a hiding mechanism for operations is required for the hierarchical structuring of data types, but the convention introduced in §8.3.2 is not satisfactionary since it renames a function rather than removing it from the visible signature.

Finally, we mention the great number of modules involved. Due to the layered module structure introduced in §8.4, each new data type requires at least three new modules. This results in much overhead for the development of a data type and its instantiation (since a renaming has to be applied to all modules). Moreover, the import graph of a specification becomes large and difficult to understand (see for example Figure 9.2).

The previous chapters concentrated on parameterization and renaming as only mechanism for reuse. Since ASF+SDF has good support for *literate programming*, we should also mention documentation as a good technique to promote reuse. Using the Meta-Environment, documentation can be created automatically from the specification by translating the specification to several formats. These translations include: the translation to LaTeX, by the tool ToLaTeX (Visser, 1994) and the translation to HTML (Van der Graaf, 1997) using the Box formalism (Van den Brand and Visser, 1996).

10.1 Future Work

Because the library of Chapter 9 is very small we are interested in the further development of the library. Not only by developing new operations and new BagLike data types, but also by developing new hierarchies (a TreeLike hierarchy for instance).

The library developed in the previous chapter is based on incomplete equations. We have only defined equations for the *good* case. No equations have been defined for exceptions like popping an element from an empty stack. It would therefore be interesting to investigate the development of a library of complete equations using an exception mechanism. For example, we could consider the specification of data types based on four valued logic (Bergstra *et al.*, 1995) in accordance with Bergstra and Sellink (1996).

Another topic that needs further investigation is the introduction of higher-order functions in the library. Using parameterized data types we believe that higher-order functions, based on the framework as defined for ATerms (Van den Brand *et al.*, 1997c), is possible.

Finally, we consider the informal notions of the structuring of data types unsatisfactory. Although these ideas helped with the structuring of data types, they are not sufficient for use in general. We think that some formal notion for the structuring of components is required because structuring promotes the reuse of components as was demonstrated in the previous chapters.

Part III

Appendices

Appendix A

Specification of Single Module Renaming

In this appendix the complete renaming specification is included.

A.1 Sdf-syntax

Part of the syntax definition of SDF in SDF. This syntax definition is based on the definition in Heering *et al.* (1989). We have prefixed each sort declaration in order to prevent name clashes with declarations in other parts of the renaming specification

exports

sorts Sdf-Id Sdf-Iterator Sdf-EscChar Sdf-C-Char Sdf-CharRange Sdf-CharClass Sdf-L-Char Sdf-Literal Sdf-CfFunction Sdf-CfElem Sdf-Attributes Sdf-Q-Literal Sdf-U-Literal Sdf-SortId Sdf-LexElem Sdf-LexicalFunction Sdf-SpecialLexId Sdf-BasicLexElem

lexical syntax

[A-Z]	→ Sdf-Id
[A-Z][A-Za-z0-9\-_]∗[A-Za-z0-9]	→ Sdf-Id
"\"∼□	→ Sdf-EscChar
"\"[01][0-7][0-7]	→ Sdf-EscChar
∼[\000-\037\-\[\]\␣]	→ Sdf-C-Char
Sdf-EscChar	→ Sdf-C-Char
Sdf-C-Char	→ Sdf-CharRange
Sdf-C-Char "−"Sdf-C-Char	→ Sdf-CharRange
"["Sdf-CharRange∗"]"	→ Sdf-CharClass
∼[\000-\037"\␣]	→ Sdf-L-Char
Sdf-EscChar	→ Sdf-L-Char
"\""Sdf-L-Char∗"\""	→ Sdf-Q-Literal
[a-z]	→ Sdf-U-Literal

[a-z][A-Za-z0-9\-_]*[A-Za-z0-9]	→ Sdf-U-Literal
[␣\t\n]	→ LAYOUT
"%%"~[\n]*"\n"	→ LAYOUT
"%"~[\n%]+"%"	→ LAYOUT
Sdf-Id	→ Sdf-SortId

context-free syntax

Sdf-U-Literal	→ Sdf-Literal
Sdf-Q-Literal	→ Sdf-Literal
"+"	→ Sdf-Iterator
"*"	→ Sdf-Iterator
Sdf-LexElem+ "→" Sdf-SortId	→ Sdf-LexicalFunction
Sdf-LexElem+ "→" Sdf-SpecialLexId	→ Sdf-LexicalFunction
"LAYOUT"	→ Sdf-SpecialLexId
"IGNORE"	→ Sdf-SpecialLexId
"REJECT"	→ Sdf-SpecialLexId
Sdf-SortId	→ Sdf-BasicLexElem
Sdf-Literal	→ Sdf-BasicLexElem
Sdf-CharClass	→ Sdf-BasicLexElem
"~" Sdf-CharClass	→ Sdf-BasicLexElem
Sdf-BasicLexElem Sdf-Iterator	→ Sdf-LexElem
Sdf-BasicLexElem	→ Sdf-LexElem
Sdf-CfElem* "→" Sdf-SortId Sdf-Attributes	→ Sdf-CfFunction
Sdf-Literal "(" {Sdf-CfElem ","}* ")" "→"	
Sdf-SortId Sdf-Attributes	→ Sdf-CfFunction
Sdf-SortId	→ Sdf-CfElem
Sdf-Literal	→ Sdf-CfElem
Sdf-SortId Sdf-Iterator	→ Sdf-CfElem
"{" Sdf-SortId Sdf-Literal "}" Sdf-Iterator	→ Sdf-CfElem
	→ Sdf-Attributes
"{" {Sdf-Literal ","}+ "}"	→ Sdf-Attributes

A.2 Sdf-renamings

imports Sdf-syntax[A.1]

exports

sorts Import Renaming Renamings Symbol Production

context-free syntax

Sdf-SortId	→ Symbol
Sdf-SortId Sdf-Iterator	→ Symbol
"{" Sdf-SortId Sdf-Literal "}" Sdf-Iterator	→ Symbol
Sdf-LexicalFunction	→ Production
Sdf-CfFunction	→ Production

Sdf-Id Renamings → Import
“[” Renaming∗ “]” → Renamings

Symbol “⇒” Symbol → Renaming
Symbol “⇒” Symbol “<” Sdf-Id “>” → Renaming

Production “⇒” Production → Renaming
Production “⇒” Production “<” Sdf-Id “>” → Renaming

A.3 Module-table

imports Sdf-syntax[A.1]
exports
sorts ModuleTable ModuleEntry ModuleExt
lexical syntax
[0-9]+ → ModuleExt
context-free syntax
Sdf-Id “,” Sdf-Id “,” ModuleExt → ModuleEntry
ModuleEntry∗ → ModuleTable
variables
MT [*0-9′*]∗ → ModuleTable

A.4 Sdf2asfix

imports Sdf-syntax[A.1] AsFixParseTrees
exports
context-free syntax
asfix(Sdf-Id) → ATerm
asfix(Sdf-LexicalFunction) → ATerm
asfix(Sdf-CfElem) → ATerm
asfix(Sdf-CfFunction) → ATerm
hiddens
context-free syntax
asfixlex(Sdf-LexElem) → ATerm
asfix(Sdf-SpecialLexId) → ATerm
asfix-elems(Sdf-LexElem+) → ATermList
asfix(Sdf-Iterator) → ATerm
asfix(Sdf-EscChar) → ATerm
asfix(Sdf-CharClass) → ATerm
asfix(Sdf-Attributes) → ATerm
asfix-attrs({Sdf-Literal “,”}+) → ATermList
asfix-syms(Sdf-CfElem∗) → ATermList
asfix-syms-sep({Sdf-CfElem “,”}∗) → ATermList
variables
“*Char*∗”[*0-9*]∗ → CHAR∗

Char [0-9]∗	→	CHAR
Id	→	Sdf-Id
SortId	→	Sdf-SortId
Attributes	→	Sdf-Attributes
Literals	→	{Sdf-Literal “,”}+
Literal	→	Sdf-Literal
CfElem [0-9]∗	→	Sdf-CfElem
“*CfElem∗*”	→	Sdf-CfElem∗
CfElems	→	{Sdf-CfElem “,”}∗
Iterator	→	Sdf-Iterator
“*LexElem+*”	→	Sdf-LexElem+
“*LexElem∗*”	→	Sdf-LexElem∗
LexElem	→	Sdf-LexElem
BasicLexElem	→	Sdf-BasicLexElem
SpecialLexId	→	Sdf-SpecialLexId
CharClass	→	Sdf-CharClass

equations

[01] asfix(sdf-id($Char^*$))
= init(id)[chars := literal(`"""` $Char^*$ `"""`)]

[02] asfix(sdf-sortid($Char^*$))
= init(sort)[chars := literal(`"""` $Char^*$ `"""`)]

[03] asfix(+)
= init(l)[chars := literal(`""" "+" """`)]

[04] asfix(∗)
= init(l)[chars := literal(`""" "*" """`)]

[05] asfix(sdf-escchar($Char^*$))
= init(l)[chars := literal(`"""` $Char^*$ `"""`)]

[06] asfix(sdf-charclass($Char^*$))
= init(char-class)[chars := literal(`"""` $Char^*$ `"""`)]

[07] asfix(sdf-q-literal($Char^*$))
= init(ql)[chars := literal($Char^*$)]

[08] asfix(sdf-u-literal($Char^*$))
= init(l)[chars := literal($Char^*$)]

Conversion of lexical function to AsFix: First, its lexical elements and target sort are converted to AsFix. Next, a new AsFix production is constructed . Note that a lexical function cannot contain attributes, hence we set the attributes of the production to "no-attrs".

$$[09]\quad \frac{\begin{aligned}\text{asfix-elems}(\mathit{LexElem}^+) &= \mathit{Tl},\\ \text{asfix}(\mathit{SortId}) &= T_2\end{aligned}}{\begin{array}{l}\text{asfix}(\mathit{LexElem}^+ \to \mathit{SortId}) = \\ \text{init(prod)}[\text{syms} := \mathit{Tl}]\\ [\text{sym} := T_2]\\ [\text{attrs} := \text{no-attrs}]\\ [> \text{syms} := \text{w}(\texttt{"␣"})]\\ [< \text{sym} := \text{w}(\texttt{"␣"})]\end{array}}$$

Again, a conversion of a lexical function to AsFix. The only difference with the previous equation is the target type of the function which will be either "LAYOUT", "IGNORE", or "REJECT".

$$[10]\quad \frac{\begin{aligned}\text{asfix-elems}(\mathit{LexElem}^+) &= \mathit{Tl},\\ \text{asfix}(\mathit{SpecialLexId}) &= T_2\end{aligned}}{\begin{array}{l}\text{asfix}(\mathit{LexElem}^+ \to \mathit{SpecialLexId}) = \\ \text{init(prod)}[\text{syms} := \mathit{Tl}]\\ [\text{sym} := T_2]\\ [\text{attrs} := \text{no-attrs}]\\ [> \text{syms} := \text{w}(\texttt{"␣"})]\\ [< \text{sym} := \text{w}(\texttt{"␣"})]\end{array}}$$

Conversion of Sdf-LexElem+ to AtermList. The elements in the list are separated by white spaces.

$$[11]\quad \text{asfix-elems}(\mathit{LexElem}) = [\text{asfixlex}(\mathit{LexElem})]$$

$$[12]\quad \frac{\begin{aligned}\text{asfixlex}(\mathit{LexElem}) &= T_1,\\ \text{asfix-elems}(\mathit{LexElem}^+) &= \mathit{Tl}_1\end{aligned}}{\text{asfix-elems}(\mathit{LexElem}\ \mathit{LexElem}^+) = T_1 : \text{w}(\texttt{"␣"}) : \mathit{Tl}_1}$$

$$\begin{array}{ll}[13] & \text{asfix(LAYOUT)}\\ & = \text{init(sort)}[\text{chars} := \text{literal}(\texttt{"LAYOUT"})]\end{array}$$

$$\begin{array}{ll}[14] & \text{asfix(IGNORE)}\\ & = \text{init(sort)}[\text{chars} := \text{literal}(\texttt{"IGNORE"})]\end{array}$$

$$\begin{array}{ll}[15] & \text{asfix(REJECT)}\\ & = \text{init(sort)}[\text{chars} := \text{literal}(\texttt{"REJECT"})]\end{array}$$

[16] $\text{asfixlex}(\mathit{SortId}) = \text{asfix}(\mathit{SortId})$

[17] $\text{asfixlex}(\mathit{Literal}) = \text{asfix}(\mathit{Literal})$

[18] $\text{asfixlex}(\mathit{CharClass}) = \text{asfix}(\mathit{CharClass})$

[19] $$\frac{\text{asfix}(\mathit{CharClass}) \;=\; T}{\text{asfixlex}(\sim \mathit{CharClass}) \;=\; \text{init}(\text{neg})[\text{char-class} := T]}$$

[20] $$\frac{\begin{array}{rcl}\text{asfixlex}(\mathit{BasicLexElem}) &=& T_1,\\ \text{asfix}(\mathit{Iterator}) &=& T_2\end{array}}{\begin{array}{l}\text{asfixlex}(\mathit{BasicLexElem}\ \mathit{Iterator}) \;=\\ \text{init}(\text{iter})[\text{sym} := T_1][\text{iter} := T_2]\end{array}}$$

Conversion of context free function to AsFix.

[21] $\text{asfix}(\mathit{CfElem}^* \to \mathit{SortId}\ \mathit{Attributes})$
$= \text{init}(\text{prod})[\text{syms} := \text{asfix-syms}(\mathit{CfElem}^*)]$
$[\text{sym} := \text{asfix}(\mathit{SortId})]$
$[\text{attrs} := \text{asfix}(\mathit{Attributes})]$

Conversion of context free function to AsFix.

[22] $$\frac{\begin{array}{rcl}\text{asfix-syms-sep}(\mathit{CfElems}) &=& \mathit{Tl},\\ \text{asfix}(\mathit{Literal}) &=& T,\\ [T, \text{w}(\texttt{"␣"}), \text{l}(\texttt{"("}), \text{w}(\texttt{"␣"})] &=& \mathit{Tl}'\end{array}}{\begin{array}{l}\text{asfix}(\mathit{Literal}(\mathit{CfElems}) \to \mathit{SortId}\ \mathit{Attributes}) \;=\\ \text{init}(\text{prod})[\text{syms} := \mathit{Tl}' +\!\!+\ \mathit{Tl} +\!\!+ [\text{w}(\texttt{"␣"}), \text{l}(\texttt{")"})]]\\ [\text{sym} := \text{asfix}(\mathit{SortId})]\\ [\text{attrs} := \text{asfix}(\mathit{Attributes})]\end{array}}$$

[23] $\text{asfix}() = \text{no-attrs}$

[24] $\text{asfix}(\{\mathit{Literals}\})$
$= \text{init}(\text{attrs})[\text{attrs} := \text{asfix-attrs}(\mathit{Literals})]$

[25] $\text{asfix}(\mathit{SortId}\ \mathit{Iterator})$
$= \text{init}(\text{iter})[\text{sym} := \text{asfix}(\mathit{SortId})]$
$[\text{iter} := \text{asfix}(\mathit{Iterator})]$

[26] $\text{asfix}(\{\mathit{SortId}\ \mathit{Literal}\}\ \mathit{Iterator})$

$=$ init(iter-sep)[sym := asfix(*SortId*)]
[sep := asfix(*Literal*)]
[iter := asfix(*Iterator*)]

[27] asfix-attrs(*Literal*) = [asfix(*Literal*)]

[28] asfix-attrs(*Literal*, *Literals*)
$=$ asfix(*Literal*) : asfix-attrs(*Literals*)

[29] asfix-syms() = [ ]

[30] asfix-syms(*CfElem* $CfElem^*$)
$=$ asfix(*CfElem*) : asfix-syms($CfElem^*$)

[31] asfix-syms-sep() = [ ]

[32] asfix-syms-sep(*CfElem*) = [asfix(*CfElem*)]

[33] asfix-syms-sep($CfElem_1$, $CfElem_2$, *CfElems*)
$=$ asfix($CfElem_1$) : w("␣") : l(",") : w("␣")
: asfix-syms-sep($CfElem_2$, *CfElems*)

A.5 Renamings2asfix

imports Sdf-renamings$^{A.2}$ Sdf2asfix$^{A.4}$
exports
context-free syntax
"asfix" "(" Import ")" → ATermList
hiddens
context-free syntax
"asfix" "(" Renamings ")" → ATermList
"asfix" "(" Renaming ")" → ATerm
"asfix-sym" "(" Symbol ")" → ATerm
"asfix-prod" "(" Production ")" → ATerm
variables
Renamings → Renamings
"*Renaming*∗" → Renaming∗
Renaming → Renaming
Id → Sdf-Id
Symbol [′]∗ → Symbol

$Production$ $[']*$ $\to$ Production

$SortId$ $\to$ Sdf-SortId
$Iterator$ $\to$ Sdf-Iterator
$Literal$ $\to$ Sdf-Literal
$CfFunction$ $\to$ Sdf-CfFunction
$LexFunction$ $\to$ Sdf-LexicalFunction

equations

[01] asfix(Id $Renamings$) = asfix($Renamings$)

[02] asfix([]) = [ ]

[03] asfix([$Renaming$ $Renaming^*$])
= asfix($Renaming$) : asfix([$Renaming^*$])

[04] asfix($Symbol \Rightarrow Symbol'$)
= [asfix-sym($Symbol$), asfix-sym($Symbol'$)]

[05] asfix($Symbol \Rightarrow Symbol' < Id >$)
= [asfix-sym($Symbol$), asfix-sym($Symbol'$)] {asfix(Id)}

[06] asfix($Production \Rightarrow Production'$)
= [asfix-prod($Production$), asfix-prod($Production'$)]

[07] asfix($Production \Rightarrow Production' < Id >$)
= [asfix-prod($Production$), asfix-prod($Production'$)] {asfix(Id)}

[08] asfix-sym($SortId$) = asfix($SortId$)

[09] asfix-sym($SortId$ $Iterator$) = asfix($SortId$ $Iterator$)

[10] asfix-sym({$SortId$ $Literal$} $Iterator$)
= asfix({$SortId$ $Literal$} $Iterator$)

[11] asfix-prod($CfFunction$) = asfix($CfFunction$)

[12] asfix-prod($LexFunction$) = asfix($LexFunction$)

A.6 ModuleTable2AsFix

imports Module-table[A.3] AsFixParseTrees Sdf-renamings[A.2] Sdf2asfix[A.4]
exports
context-free syntax
"asfix" "(" ModuleTable ")" → ATermList
"update-imports" "(" ATermList "," ATermList ")" → ATermList
hiddens
context-free syntax
"asfix1" "(" ModuleEntry ")" → ATermList
"asfix" "(" ModuleExt ")" → ATerm

"make-new-name" "(" ATerm "," ATerm ")" → ATerm
"basename" "(" ATerm ")" → ATerm
"basename′" "(" Literal ")" → Literal
"is-nat-con" "(" NatCon ")" → Bool
"is-nat-digit" "(" NatCon ")" → Bool
variables
Id [0-9′]∗ → Sdf-Id
ME [0-9′]∗ → ModuleEntry
"*ME*∗"[0-9′]∗ → ModuleEntry∗
MExt [0-9′]∗ → ModuleExt
"*Char*∗"[0-9′]∗ → CHAR∗
Char [0-9]∗ → CHAR
equations
Convert module table to AsFix, i.e. create an AsFix list and for all elements $ModuleName$, *Renaming*, Id in the table, insert the entry $[ModuleName, Id]$ in the list.

[01] asfix() = [ ]

[02] asfix(*ME* *ME**) = asfix1(*ME*) : asfix(*ME**)

[03] asfix1(*Id*$_1$, *Id*$_2$, *MExt*)
= [asfix(*Id*$_1$), asfix(*MExt*)]

[04] asfix(moduleext(*Char**))
= init(id)[chars := literal(""" *Char** """)]

ModuleName generates a valid module name for a module, using information in the module table Tl. Suppose $M = basename(T)$, then ModuleName returns T when the module table does not contain an entry for module M. If the table does contain an entry $[M, Id]$, then *M-Id* is returned. The function 'basename' is defined below.

No entry for module T exists in the module table Tl.

$$[05] \quad \frac{\begin{array}{rcl} \text{filter}(\texttt{"matchingModuleName"}(T),\ Tl) & = & Tl_1, \\ \text{empty}(Tl_1) & = & \text{true} \end{array}}{\text{apply}(\texttt{"moduleName"},\ Tl,\ T)\ =\ T}$$

The module table Tl contains an entry $[T_1, T_2]$ for module T, Create a new module name T_1-T_2 for it.

$$[06] \quad \frac{\text{filter}(\texttt{"matchingModuleName"}(T),\ Tl)\ =\ [[T_1,\ T_2]]}{\text{apply}(\texttt{"moduleName"},\ Tl,\ T)\ =\ \text{make-new-name}(T_1,\ T_2)}$$

Predicate that determines whether module name T_1 matches T_2. T_1 and T_2 match when $basename(T_1) \equiv T_2$.

$$[07] \quad \frac{\text{basename}(T_1)\ =\ T_1'}{\text{holds}(\texttt{"matchingModuleName"},\ T_1,\ [T_2,\ T_3])\ =\ T_1' \equiv T_2}$$

Remove possible id from T, i.e. $basename(M) = M$, $basename(M - id) = M$.

$$[08] \quad \frac{\begin{array}{rcl} T[\text{chars}] & = & L, \\ \text{basename}'(L) & = & L' \end{array}}{\text{basename}(T)\ =\ \text{init}(\text{id})[\text{chars} := L']}$$

To remove an id from a module name, we must assure that we remove only the rightmost suffix `-id`. The function 'basename'' should be called recursively on the suffix to find a "minimal" match in strings containing several hyphens. Moreover, we have to assure that the suffix is a natural number and not an ordinary string. Therefore, we use an unusual approach by converting $Char_2^*$ to `NatCon` (using its lexical constructor function 'natcon'), and by calling the predicate 'is-nat-con' to determine whether all its characters are digits.

$$[09] \quad \frac{\begin{array}{l} \text{basename}'(\text{literal}(\texttt{"""}\ Char_2^*\ \texttt{"""}))\ =\ \text{literal}(\texttt{"""}\ Char_3^*\ \texttt{"""}), \\ \text{if is-nat-con}(\text{natcon}(Char_2^*)) \\ \text{then literal}(\texttt{"""}\ Char_1^*\ \texttt{"""}) \\ \text{else literal}(\texttt{"""}\ Char_1^*\ \texttt{"-"}\ Char_3^*\ \texttt{"""})\ \text{fi}\ =\ L \end{array}}{\text{basename}'(\text{literal}(\texttt{"""}\ Char_1^*\ \texttt{"-"}\ Char_2^*\ \texttt{"""}))\ =\ L}$$

[10] $\text{basename}'(L) = L$ **otherwise**

Construct a new module name by appending the id T_2 to T_1.

$$[11] \quad \frac{\begin{array}{rcl} T_1[\text{chars}] & = & \text{literal}(\texttt{"""}\ Char_1^*\ \texttt{"""}), \\ T_2[\text{chars}] & = & \text{literal}(\texttt{"""}\ Char_2^*\ \texttt{"""}) \end{array}}{\begin{array}{l} \text{make-new-name}(T_1,\ T_2)\ = \\ \text{init}(\text{id})[\text{chars} := \text{literal}(\texttt{"""}\ Char_1^*\ \texttt{"-"}\ Char_2^*\ \texttt{"""})] \end{array}}$$

Renaming the names of imported modules according to information in the module table.

[12] update-imports(Tl_1, Tl_2) = map(`"moduleName"`(Tl_2), Tl_1)

[13] update-imports(Tl_1, Tl_2) = Tl_2 **otherwise**

is-nat-con: predicate that determines whether all characters of a string are digits.

[14] is-nat-con(natcon($Char$)) = is-nat-digit(natcon($Char$))

[15] is-nat-con(natcon($Char_1$ $Char_2$ $Char^*$))
= is-nat-con(natcon($Char_1$)) $\wedge$ is-nat-con(natcon($Char_2$ $Char^*$))

[16] is-nat-digit(0) = true
[17] is-nat-digit(1) = true
[18] is-nat-digit(2) = true
[19] is-nat-digit(3) = true
[20] is-nat-digit(4) = true
[21] is-nat-digit(5) = true
[22] is-nat-digit(6) = true
[23] is-nat-digit(7) = true
[24] is-nat-digit(8) = true
[25] is-nat-digit(9) = true
[26] is-nat-digit(n) = false **otherwise**

A.7 Ren

imports Ren-symbols $^{A.14}$ Ren-prod $^{A.10}$ Ren-productions $^{A.12}$ Imports $^{A.15}$
Ren-sorts $^{A.13}$ Ren-prod-skel $^{A.11}$ Ren-appl $^{A.9}$ Renamings2asfix $^{A.5}$
exports
context-free syntax
"Ren" "(" Import "," ATerm "," ModuleTable ")" $\rightarrow$ ATerm
hiddens
context-free syntax
"Ren-productions" "(" ATerm "," ATermList ")" $\rightarrow$ ATerm
variables
I $\rightarrow$ Import
M *[0-9']** $\rightarrow$ ATerm
equations
Main renaming function.
First, we convert the renaming to AsFix and obtain the list of abbreviations

from the AsFix module. Then, we apply the renaming to the module and update the imported modules according to the origin specifications. As a final step, we remove empty grammars. Grammars may have become empty due to the removal of a sort or production declaration corresponding to an origin specification.

$$
[01] \quad \frac{\begin{array}{rcl} \text{asfix}(I) & = & Tl, \\ M[\text{abbrevs}] & = & Abbrevs, \\ \text{rename-aterm}(Tl, Abbrevs, M) & = & M_1, \\ \text{imports}(M_1, MT) & = & M_2, \\ \text{remove-empty-grammars}(M_2) & = & M_3 \end{array}}{\text{Ren}(I, M, MT) \ = \ M_3}
$$

A.8 Ren-ATerm

imports ATerm-Trees-Ann$^{A.18}$ Lookup$^{A.16}$ Misc$^{A.17}$
exports
context-free syntax
"rename" "(" ATermList "," ATerm "," ATerm ")" → ATerm
"rename-aterm" "(" ATermList "," ATerm "," ATerm ")" → ATerm
"should-rename" "(" ATermList "," ATerm "," ATerm ")" → Bool
equations
Renaming of a term T according to a list of renamings Tl. We use the annotated function '`treplace-ann`' to traverse the tree and to replace terms that match a renaming. '`treplace-ann`' returns the renamed ATerm and, as annotation, a set of module names (corresponding to matching renamings with origin specifications). From this list a new import section is created that is appended to the module

[01] rename-aterm(Tl, $Abbrevs$, T)
= treplace-ann(`"should-rename"`(Tl, $Abbrevs$),
`"rename"`(Tl, $Abbrevs$), T)

[02] apply(`"rename-aterm"`, Tl, $Abbrevs$, T)
= rename-aterm(Tl, $Abbrevs$, T)

[03] apply(`"rename"`, Tl, $Abbrevs$, T) = rename(Tl, $Abbrevs$, T)

[04] holds(`"should-rename"`, Tl, T_2, T_3) = should-rename(Tl, T_2, T_3)

For each kind of term that can be renamed (i.e., sorts, productions etc.), the predicate '`should-rename`' should be defined, that returns '`true`' to indicate that a matching renaming has been found for a particular term. The default case, in which case no renaming should applied to a term, is handled here.

[05] should-rename(Tl, T_2, T_3) = false **otherwise**

A.9 Ren-appl

imports Ren-ATerm$^{A.8}$
hiddens
  context-free syntax
    "subst" "(" ATerm "," ATerm ")" → ATermList
  variables
    P *[0-9]∗* → ATerm
    Syms *[0-9]∗* → ATermList
    Sym *[0-9]∗* → ATerm
    Attrs *[0-9]∗* → ATerm
equations

Renaming of a term with function symbol 'appl'. Return '`true`' when a matching renaming exists for T in Tl. To determine whether a matching renaming exists for T, the production declaration of T is matched against the renamings in Tl. The declaration is found in the table of abbreviations.

[01]
$$\frac{\begin{array}{rcl}\text{fun}(T) & = & \text{appl},\\ \text{abbreviation}(T[\text{prod}],\ \mathit{Abbrevs}) & = & T',\\ \text{exists-matching-renaming}(T',\ Tl) & = & \text{true}\end{array}}{\text{should-rename}(Tl,\ \mathit{Abbrevs},\ T)\ =\ \text{true}}$$

Renaming of application T. The matching renaming $[T1, T2]$ is retrieved from Tl. The existence of such renaming is assured since '`should-rename`' has yielded '`true`' before. T is replaced by $T2$, and all formal parameters are substituted by their corresponding terms from T. Finally, the renaming is applied recursively to all function arguments.

[02]
$$\frac{\begin{array}{rcl}\text{fun}(T) & = & \text{appl},\\ T' & = & \text{abbreviation}(T[\text{prod}],\ \mathit{Abbrevs}),\\ [T_1,\ T_2] & = & \text{term}(\text{lookup}(T',\ Tl)),\\ Tl' & = & \text{subst}(T_2[\text{syms}],\ T[\text{args}]),\\ T'' & = & T[\text{args} := Tl']\end{array}}{\begin{array}{l}\text{rename}(Tl,\ \mathit{Abbrevs},\ T)\ =\\ \text{map-sons-ann}(\texttt{"rename-aterm"}(Tl,\ \mathit{Abbrevs}),\ T'')\end{array}}$$

Substitute formal parameters of Tl_1 by their corresponding actual parameters of Tl_2.

[03]
$$\frac{\text{empty}(Tl_1)\ =\ \text{true}}{\text{subst}(Tl_1,\ Tl_2)\ =\ Tl_1}$$

Quoted literal

$$[04]\quad \frac{\begin{array}{rcl}\text{empty}(Tl_1) & = & \text{false},\\ \text{first}(Tl_1) & = & \text{ql}(L)\end{array}}{\text{subst}(Tl_1,\ Tl_2)\ =\ \text{l}(L) : \text{subst}(\text{rest}(Tl_1),\ Tl_2)}$$

Instantiation of a sort by a lexical symbol, variable or application. Observe that we forget about the symbol information in the *lex* and *var* constructs.

$$[05]\quad \frac{\begin{array}{rcl}\text{empty}(Tl_1) & = & \text{false},\\ \text{fun}(\text{first}(Tl_1)) & = & \text{sort},\\ \begin{array}{l}\text{fun}(\text{first}(Tl_2)) \equiv \text{var}\\ \vee\ \text{fun}(\text{first}(Tl_2)) \equiv \text{lex}\\ \vee\ \text{fun}(\text{first}(Tl_2)) \equiv \text{appl}\end{array} & = & \text{true}\end{array}}{\begin{array}{l}\text{subst}(Tl_1,\ Tl_2)\ =\\ \text{first}(Tl_2) : \text{subst}(\text{rest}(Tl_1), \text{rest}(Tl_2))\end{array}}$$

Instantiation of a list.

$$[06]\quad \frac{\begin{array}{rcl}\text{empty}(Tl_1) & = & \text{false},\\ \begin{array}{l}\text{fun}(\text{first}(Tl_1)) \equiv \text{iter}\\ \vee\ \text{fun}(\text{first}(Tl_1)) \equiv \text{iter-sep}\end{array} & = & \text{true},\\ \text{fun}(\text{first}(Tl_2)) & = & \text{list}\end{array}}{\begin{array}{l}\text{subst}(Tl_1,\ Tl_2)\ =\\ \text{first}(Tl_2)[\text{sym} := \text{first}(Tl_2)] : \text{subst}(\text{rest}(Tl_1), \text{rest}(Tl_2))\end{array}}$$

Skip any white space or literals from Tl_2 (no quoted-literals possible).

$$[07]\quad \frac{\begin{array}{rcl}\text{empty}(Tl_1) & = & \text{false},\\ \begin{array}{l}\text{fun}(\text{first}(Tl_2)) \equiv \text{w}\\ \vee\ \text{fun}(\text{first}(Tl_2)) \equiv \text{l}\end{array} & = & \text{true}\end{array}}{\text{subst}(Tl_1,\ Tl_2)\ =\ \text{subst}(Tl_1, \text{rest}(Tl_2))}$$

[08] $\text{subst}(Tl_1,\ Tl_2) = \text{first}(Tl_1) : \text{subst}(\text{rest}(Tl_1),\ Tl_2)$ **otherwise**

A.10 Ren-prod

imports Ren-ATerm$^{A.8}$
hiddens
  variables
    Syms [*0-9*]$*$ $\rightarrow$ ATermList
    Sym [*0-9*]$*$ $\rightarrow$ ATerm
    Attrs [*0-9*]$*$ $\rightarrow$ ATerm
equations

Renaming of a term T with function symbol 'prod'. Return '`true`' when a matching renaming exists for T in Tl.

$$[01] \quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{prod}, \\ \text{exists-matching-renaming}(T,\ Tl) & = & \text{true} \end{array}}{\text{should-rename}(Tl,\ Abbrevs,\ T) \ = \ \text{true}}$$

Production renaming according to renaming P1 ⇒ P2.
Renaming is applied when a matching renaming for T can be found, i.e., there exists a renaming $[T_1, T_2]$ such that:

- Symbols of both productions T and T_1 match (discarding any white space),
- Attributes of T and T_1 match.

The attributes of the renamed term are constructed as follows:

- When no attributes are specified in $T1$ and $T2$, then the attributes of T are used,
- else, the attributes of $T2$ are used. (The attributes of T and $T1$ should match exactly in this case but this has been checked already in '`lookup`').

$$[02] \quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{prod}, \\ \text{lookup}(T,\ Tl) & = & T_1, \\ \text{term}(T_1) & = & [T_2,\ T_3], \\ \begin{array}{l} \text{if } T_2[\text{attrs}] \equiv \text{no-attrs} \\ \quad \wedge\ T_3[\text{attrs}] \equiv \text{no-attrs} \\ \text{then } T[\text{attrs}] \\ \text{else } T_3[\text{attrs}] \text{ fi} \end{array} & = & Attrs, \\ T_3[\text{syms}] & = & Syms, \\ T_3[\text{sym}] & = & Sym, \\ \begin{array}{l} T[\text{syms} := Syms] \\ [\text{sym} := Sym] \\ [\text{attrs} := Attrs] \end{array} & = & T', \\ \text{fun}(T') & = & \text{prod} \end{array}}{\text{rename}(Tl,\ Abbrevs,\ T) \ = \ \text{set-ann}(T',\ \text{ann}(T_1))}$$

A.11 Ren-prod-skel

imports Ren-ATerm$^{A.8}$
hiddens
  variables
    Syms [*0-9*]$*$ → ATermList
    Sym [*0-9*]$*$ → ATerm

equations

Renaming of a term T with function symbol 'prod-skel'. Return 'true' when a matching renaming exists for T in Tl. To determine whether a matching renaming exists for T, the production declaration of T is matched against the renamings in Tl. The declaration is found in the table of abbreviations.

$$[01]\quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{prod-skel}, \\ \text{abbreviation}(T[\text{prod}],\ \mathit{Abbrevs}) & = & T', \\ \text{exists-matching-renaming}(T',\ Tl) & = & \text{true} \end{array}}{\text{should-rename}(Tl,\ \mathit{Abbrevs},\ T)\ =\ \text{true}}$$

Renaming of a term with function symbol 'prod-skel'.

$$[02]\quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{prod-skel}, \\ T' & = & \text{abbreviation}(T[\text{prod}],\ \mathit{Abbrevs}), \\ [T_1,\ T_2] & = & \text{term}(\text{lookup}(T',\ Tl)), \\ T_2[\text{syms}] & = & \mathit{Syms}, \\ T_2[\text{sym}] & = & \mathit{Sym}, \\ T_3 & = & \mathit{Syms} +\!\!+ [\text{w}(\texttt{"␣"}), \text{l}(\texttt{"->"}), \text{w}(\texttt{"␣"}), \mathit{Sym}, \text{w}(\texttt{"␣"})], \\ T[\text{syms} := T_3] & = & T'' \end{array}}{\text{rename}(Tl,\ \mathit{Abbrevs},\ T)\ =\ T''}$$

A.12 Ren-productions

imports Ren-ATerm$^{A.8}$ Ren-prod$^{A.10}$
hiddens
context-free syntax
"rename-prod" "(" ATermList "," ATerm "," ATerm ")" → ATerm
equations

The renaming of productions in a lexical syntax or context-free syntax section may yield an empty section as a result of the removal of production declarations according to an origin specification. Because an empty section is not allowed in ASF+SDF, we have to remove them. Special care has to be taken when considering an origin specification, since productions occur in the lexical syntax and context-free syntax parts as well as in the abbreviations section. Because productions should only removed from the lexical syntax and context-free syntax sections, we use different functions for renaming of productions in lexical syntax, context-free syntax and abbreviations sections. Here we enter a lexical syntax or context free syntax section.

$$[01]\quad \frac{\begin{array}{l} \text{fun}(T) \equiv \text{lexical-syntax} \\ \vee\ \text{fun}(T) \equiv \text{context-free-syntax} \end{array} = \text{true}}{\text{should-rename}(Tl,\ \mathit{Abbrevs},\ T)\ =\ \text{true}}$$

For productions in the lexical syntax and context-free syntax sections we use the function '`rename-prod`' (that removes productions when an origin specification has been specified). When the section becomes empty, it will be removed.

$$\text{[02]}\quad \frac{\begin{array}{rcl} \text{fun}(T) \equiv \text{lexical-syntax} & & \\ \vee\ \text{fun}(T) \equiv \text{context-free-syntax} & = & \text{true}, \\ \text{map-sons-ann}(\texttt{"rename-prod"}(Tl, \mathit{Abbrevs}), T) & = & T', \\ \text{term}(T'[\text{productions}]) & = & Tl_1 \end{array}}{\begin{array}{l} \text{rename}(Tl, \mathit{Abbrevs}, T) = \\ \text{if is-empty}(Tl_1) \\ \text{then set-ann}(\text{w}(\texttt{""}), \text{ann}(T')) \\ \text{else } T' \text{ fi} \end{array}}$$

[03] $\text{apply}(\texttt{"rename-prod"}, Tl, \mathit{Abbrevs}, T)$
$= \text{rename-prod}(Tl, \mathit{Abbrevs}, T)$

[04] $\text{rename-prod}(Tl, \mathit{Abbrevs}, T)$
$= \text{treplace-ann}(\texttt{"should-rename"}(Tl, \mathit{Abbrevs}),$
$\texttt{"rename-p"}(Tl, \mathit{Abbrevs}), T)$

Renaming of a production in a lexical syntax or context-free syntax section. When an origin specification has been specified, the production is removed and the module name of the origin specification is returned as annotation.

$$\text{[05]}\quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{prod}, \\ \text{rename}(Tl, \mathit{Abbrevs}, T) & = & T' \end{array}}{\begin{array}{l} \text{apply}(\texttt{"rename-p"}, Tl, \mathit{Abbrevs}, T) = \\ \text{if empty}(\text{ann}(T')) \\ \text{then } T' \\ \text{else set-ann}(\text{w}(\texttt{""}), \text{ann}(T')) \text{ fi} \end{array}}$$

$$\text{[06]}\quad \frac{\text{fun}(T) \neq \text{prod}}{\text{apply}(\texttt{"rename-p"}, Tl, \mathit{Abbrevs}, T) = \text{rename}(Tl, \mathit{Abbrevs}, T)}$$

A.13 Ren-sorts

imports Ren-ATerm$^{A.8}$
hiddens
context-free syntax
"rename-sort" "(" ATermList "," ATerm "," ATerm ")" → ATerm
equations

Matching sort declarations should be removed from the module when the renaming contains an origin specification. Since a matching term does not tell us whether we are dealing with a sort declaration or not (the term may be the type definition of a variable, for example), we have to handle the renamings of sort declarations and ordinary symbol renamings separately. We use '`rename-sort`' to rename sort declarations.

Entering a sorts grammar. We will handle the renaming of sort declarations here differently.

$$[01] \quad \frac{\text{fun}(T) \;=\; \text{sorts}}{\text{should-rename}(Tl,\ Abbrevs,\ T) \;=\; \text{true}}$$

We use '`rename-sort`' to rename the sort declarations. When the list becomes empty due to the removal of declarations according to origin specifications, we remove the import section.

$$[02] \quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{sorts}, \\ \text{map-sons-ann}(\texttt{"rename-sort"}(Tl,\ Abbrevs),\ T) & = & T', \\ \text{term}(T'[\text{sorts}]) & = & Tl_1 \end{array}}{\begin{array}{l} \text{rename}(Tl,\ Abbrevs,\ T) \;= \\ \text{if is-empty}(Tl_1) \\ \text{then set-ann}(\text{w}(\texttt{""}),\ \text{ann}(T')) \\ \text{else } T' \text{ fi} \end{array}}$$

$$[03] \quad \begin{array}{l} \text{apply}(\texttt{"rename-sort"},\ Tl,\ Abbrevs,\ T) \\ = \text{rename-sort}(Tl,\ Abbrevs,\ T) \end{array}$$

$$[04] \quad \begin{array}{l} \text{rename-sort}(Tl,\ Abbrevs,\ T) \\ = \text{treplace-ann}(\texttt{"should-rename"}(Tl,\ Abbrevs), \\ \qquad\qquad\qquad \texttt{"rename-s"}(Tl,\ Abbrevs),\ T) \end{array}$$

Renaming of a sort declaration without origin specification.

$$[05] \quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{sort}, \\ \text{lookup}(T,\ Tl) & = & [T_1,\ T_2] \end{array}}{\text{apply}(\texttt{"rename-s"},\ Tl,\ Abbrevs,\ T) \;=\; T_2}$$

Renaming of a sort declaration with origin specification. We remove the sort (by returning a term containing white space) and return the module name of the origin specification as annotation.

$$[06] \quad \frac{\begin{array}{rcl} \text{fun}(T) & = & \text{sort}, \\ \text{lookup}(T,\ Tl) & = & [T_1,\ T_2]\ \{T'\} \end{array}}{\text{apply}(\texttt{"rename-s"},\ Tl,\ Abbrevs,\ T) \;=\; \text{w}(\texttt{""})\ \{T'\}}$$

A.14 Ren-symbols

imports Ren-ATerm$^{A.8}$
hiddens
context-free syntax
"replace-list" "(" ATerm "," ATerm ")" → ATerm
equations

The Symbol T should be renamed when a matching renaming is found in Tl.

$$[01]\quad \frac{\begin{array}{rcl}\begin{array}{l}\text{fun}(T) \equiv \text{sort}\\ \vee\ \text{fun}(T) \equiv \text{list}\\ \vee\ \text{fun}(T) \equiv \text{iter}\\ \vee\ \text{fun}(T) \equiv \text{iter-sep}\end{array} & = & \text{true},\\ \text{exists-matching-renaming}(T,\ Tl) & = & \text{true}\end{array}}{\text{should-rename}(Tl,\ Abbrevs,\ T)\ =\ \text{true}}$$

Renaming of symbols. Here, the renaming of `sort`, `iter` and `iter-sep` is handled.

$$[02]\quad \frac{\begin{array}{rcl}\begin{array}{l}\text{fun}(T) \equiv \text{sort}\\ \vee\ \text{fun}(T) \equiv \text{iter}\\ \vee\ \text{fun}(T) \equiv \text{iter-sep}\end{array} & = & \text{true},\\ \text{term}(\text{lookup}(T,\ Tl)) & = & [T_1,\ T_2]\end{array}}{\text{rename}(Tl,\ Abbrevs,\ T)\ =\ T_2}$$

Renaming of symbols. Here, list renaming is handled. This is a special case of symbol renaming since we have to change not only the type of the list (i.e., its symbol) but changing the separators in the list is required too. Moreover, the renaming should be applied recursively to all elements of the list.

$$[03]\quad \frac{\begin{array}{rcl}\text{fun}(T) & = & \text{list},\\ \text{term}(\text{lookup}(T,\ Tl)) & = & [T_1,\ T_2],\\ \text{replace-list}(T,\ T_2) & = & T_3,\\ T_3[\text{elems}] & = & Tl_3,\\ \text{map-sons}(\texttt{"rename-aterm"}(Tl,\ Abbrevs),\ Tl_3) & = & Tl_3'\end{array}}{\text{rename}(Tl,\ Abbrevs,\ T)\ =\ T_3[\text{elems} := Tl_3']}$$

List without separator: remove all separators that might still occur in the list. Separate the items by w(" ").

$$[04]\quad \frac{\begin{array}{rcl}\text{fun}(T_2) & = & \text{iter},\\ T_1[\text{elems}] & = & Tl\end{array}}{\begin{array}{l}\text{replace-list}(T_1,\ T_2)\ =\\ T_1[\text{sym} := T_2]\\ [\text{elems} := \text{foldl}(\texttt{"new-sep"}([\text{w}(\texttt{"␣"})]),\ [\ ],\ Tl)]\end{array}}$$

List with separators: remove all "old" separators and insert $sep(L)$ as new separator, put $w(" ")$ between the elements and the separators.

$$
\text{[05]} \quad \frac{\begin{array}{rcl} \text{fun}(T_2) & = & \text{iter-sep}, \\ T_2[\text{sep}] & = & \text{ql}(L), \\ T_1[\text{elems}] & = & Tl \end{array}}{\begin{array}{l} \text{replace-list}(T_1,\ T_2)\ = \\ T_1[\text{sym} := T_2] \\ [\text{elems} := \text{foldl}(\texttt{"new-sep"}([\text{w}(\texttt{"␣"}),\ \text{sep}(L),\ \text{w}(\texttt{"␣"})]),\ [\ ],\ Tl)] \end{array}}
$$

Construct a list from T and separate its elements by Tl_0.

$$
\text{[06]} \quad \frac{\begin{array}{l} \text{fun}(T) \neq \text{w}, \\ \text{fun}(T) \neq \text{sep} \end{array}}{\begin{array}{l} \text{apply}(\texttt{"new-sep"},\ Tl,\ Tl_0,\ T)\ = \\ \text{if empty}(Tl_0) \\ \text{then } [T] \\ \text{else } Tl_0 +\!\!+\ Tl +\!\!+ [T] \text{ fi} \end{array}}
$$

Construct a list from T discarding separators and white space.

$$
\text{[07]} \quad \frac{\begin{array}{l} \text{fun}(T) \equiv \text{w} \\ \vee\ \text{fun}(T) \equiv \text{sep} \end{array} = \text{true}}{\text{apply}(\texttt{"new-sep"},\ Tl,\ Tl_0,\ T)\ =\ Tl_0}
$$

A.15 Imports

imports Misc$^{A.17}$ ModuleTable2AsFix$^{A.6}$

exports

context-free syntax

"imports" "(" ATerm "," ModuleTable ")" $\rightarrow$ ATerm

equations

The module T has a list of originating modules annotated (these module names are collected during the renaming of the module and correspond to renamings with origin specifications. We create a new import section containing the module names from the list. Moreover, we update (either by changing or appending) the id's of the imported modules according to the table of module names MT.

$$
\text{[01]}\quad
\frac{
\begin{array}{rcl}
\text{asfix}(MT) & = & Tl_1, \\
T[\text{sections}] & = & Tl_2, \\
\text{map}(\texttt{"update-imports"}(Tl_1),\ Tl_2) & = & Tl_3, \\
\text{foldl}(\texttt{"unify"},\ [\,],\ \text{ann}(T)) & = & Tl_4, \\
\text{update-imports}(Tl_4,\ Tl_1) & = & Tl_4', \\
\text{ins-ws}(Tl_4') & = & Tl_4'', \\
\text{init(imports)}[\text{ids} := Tl_4''\,]\ [<\text{ids} := \text{w}(\texttt{"␣"})] & = & T_2
\end{array}
}{
\begin{array}{l}
\text{imports}(T,\ MT)\ = \\
\text{if empty}(Tl_4) \\
\text{then term}(T)[\text{sections} := Tl_3] \\
\text{else term}(T)[\text{sections} := T_2 : \text{w}(\texttt{"\textbackslash n"}) : Tl_3]\ \text{fi}
\end{array}
}
$$

Update module names in an imports section according to the information in the module table Tl.

$$
\text{[02]}\quad
\frac{
\begin{array}{rcl}
\text{fun}(T) & = & \text{imports}, \\
T[\text{ids}] & = & Tl_1, \\
\text{update-imports}(Tl_1,\ Tl) & = & Tl_1'
\end{array}
}{
\text{apply}(\texttt{"update-imports"},\ Tl,\ T)\ =\ T[\text{ids} := Tl_1']
}
$$

No update is required for sections other than imports sections.

[03] $\text{apply}(\texttt{"update-imports"},\ Tl,\ T) = T \quad \textbf{when}\ \ \text{fun}(T) \neq \text{imports}$

Append the element T to Tl when Tl does not already contain the element T.

[04] $\text{apply}(\texttt{"unify"},\ Tl,\ T)$
$= \text{if contains}(Tl,\ T)$
$\quad \text{then } Tl$
$\quad \text{else } T : Tl \text{ fi}$

A.16 Lookup

imports AsFixParseTrees
exports
  context-free syntax
    "lookup" "(" ATerm "," ATermList ")" → ATerm
    "matching-renaming" "(" ATerm "," ATerm ")" → Bool
    "exists-matching-renaming" "(" ATerm "," ATermList ")" → Bool
hiddens
  context-free syntax
    "match-attrs" "(" ATerm "," ATerm "," ATerm ")" → Bool
    "match" "(" ATerm "," ATerm ")" → Bool

equations

Look for a renaming in Tl that matches T. When more matching renamings exists, we will use the first renaming.

$$[01] \quad \frac{\begin{array}{rcl}\text{filter}(\texttt{"matching-renaming"}(T),\ Tl) & = & Tl', \\ \text{empty}(Tl') & = & \text{false}\end{array}}{\text{lookup}(T,\ Tl)\ =\ \text{first}(Tl')}$$

[02] $\text{lookup}(T,\ Tl) = [\,]$ **otherwise**

Check whether T matches $T1$. since T is a production the attributes should also match (see match-attrs below).

$$[03] \quad \frac{\begin{array}{rcl}\text{fun}(T) & = & \text{prod}, \\ \text{term}(T') & = & [T_1,\ T_2], \\ \text{fun}(T_1) & = & \text{prod}, \\ \text{match-attrs}(T[\text{attrs}],\ T_1[\text{attrs}],\ T_2[\text{attrs}]) & = & \text{true}\end{array}}{\text{matching-renaming}(T,\ T')\ =\ \text{match}(T,\ T_1)}$$

$$[04] \quad \frac{\begin{array}{rcl}\text{fun}(T) & \neq & \text{prod}, \\ \text{fun}(T) & \neq & \text{iter-sep}, \\ \text{fun}(T) & \neq & \text{iter}, \\ \text{fun}(T) & \neq & \text{list}, \\ \text{term}(T') & = & [T_1,\ T_2]\end{array}}{\text{matching-renaming}(T,\ T')\ =\ \text{match}(T,\ T_1)}$$

In case of a list, the iterators of T and $T2$ should equal This is to assure that a list $A*$ cannot be renamed to $A+$.

$$[05] \quad \frac{\begin{array}{rcl}\begin{array}{l}\text{fun}(T) \equiv \text{iter} \\ \vee\ \text{fun}(T) \equiv \text{iter-sep}\end{array} & = & \text{true}, \\ \text{term}(T') & = & [T_1,\ T_2], \\ T_1[\text{iter}] & = & T_2[\text{iter}]\end{array}}{\text{matching-renaming}(T,\ T')\ =\ \text{match}(T,\ T_1)}$$

T' is a matching renaming for T whenever T' is a matching renaming for the symbol of T (i.e., $T[sym]$).

$$[06] \quad \frac{\text{fun}(T)\ =\ \text{list}}{\text{matching-renaming}(T,\ T')\ =\ \text{matching-renaming}(T[\text{sym}],\ T')}$$

[06] $\text{matching-renaming}(T,\ T') = \text{false}$ **otherwise**

Predicate that evaluates to '`true`' when a matching renaming exists for T.

[07] $\text{holds}(\texttt{"matching-renaming"},\ T_1,\ T_2) = \text{matching-renaming}(T_1,\ T_2)$

[08] exists-matching-renaming(T, Tl)
$= $ for-some("`matching-renaming`"(T), Tl)

Predicate that determines whether or not the attributes according to production a renaming $P \Rightarrow P'$ match the attributes of a production. A match is found when:

- No attributes are specified in the renaming i.e., both attributes of P and P' are `no-attrs`.
- Attributes are specified and the list of attributes of P match the list of attributes of the production (i.e., both lists contain the same elements discarding any white space).

Case 1: no attributes are specified.

[09] match-attrs(T, no-attrs, no-attrs) = true

[10] match-attrs(no-attrs, no-attrs, T) = true

Case 2: attributes are specified and the attributes of P match the attributes the of production P' (both lists of arguments should match).

[11]
$$\frac{\begin{array}{c} T_1 \neq \text{no-attrs}, \\ T_2 \neq \text{no-attrs} \end{array}}{\text{match-attrs}(T_1,\ T_2,\ T_3)\ =\ \text{match}(T_1[\text{attrs}],\ T_2[\text{attrs}])}$$

[12] match-attrs(T_1, T_2, T_3) = false **otherwise**

Match two symbols (either `iter` or `iter-sep`). All elements should equal discarding any white space.

[13]
$$\frac{\begin{array}{rcl} \text{fun}(T_1) & = & T, \\ \text{fun}(T_2) & = & T, \\ \begin{array}{l} T \equiv \text{iter} \\ \vee\ T \equiv \text{iter-sep} \end{array} & = & \text{true}, \\ T_1[\text{sym}] & = & T_2[\text{sym}], \\ T_1[\text{sep}] & = & T_2[\text{sep}], \\ T_1[\text{iter}] & = & T_2[\text{iter}] \end{array}}{\text{match}(T_1,\ T_2)\ =\ \text{true}}$$

List renaming (a special form of symbol renaming) should also rename the elements of the list in equations. As a consequence, a match is found when a corresponding list is found.

[14]
$$\frac{\begin{array}{rcl} \begin{array}{l} \text{fun}(T_2) \equiv \text{iter} \\ \vee\ \text{fun}(T_2) \equiv \text{iter-sep} \end{array} & = & \text{true}, \\ \text{fun}(T_1) & = & \text{list} \end{array}}{\text{match}(T_1,\ T_2)\ =\ \text{match}(T_2,\ T_1[\text{sym}])}$$

Match of two productions when their symbols match, discarding any white space.

$$[15] \quad \frac{\begin{array}{rcl} \text{fun}(T_1) & = & \text{prod}, \\ \text{fun}(T_2) & = & \text{prod}, \\ \text{match}(T_1[\text{sym}],\ T_2[\text{sym}]) & = & \text{true}, \\ \text{match}(T_1[\text{syms}],\ T_2[\text{syms}]) & = & \text{true} \end{array}}{\text{match}(T_1,\ T_2)\ =\ \text{true}}$$

Discard white space in Tl_1.

$$[17] \quad \frac{\begin{array}{rcl} \text{empty}(Tl_1) & = & \text{false}, \\ \text{fun}(\text{first}(Tl_1)) & = & \text{w} \end{array}}{\text{match}(Tl_1,\ Tl_2)\ =\ \text{match}(\text{rest}(Tl_1),\ Tl_2)}$$

Discard white space in Tl_2.

$$[18] \quad \frac{\begin{array}{rcl} \text{empty}(Tl_2) & = & \text{false}, \\ \text{fun}(\text{first}(Tl_2)) & = & \text{w} \end{array}}{\text{match}(Tl_1,\ Tl_2)\ =\ \text{match}(Tl_1, \text{rest}(Tl_2))}$$

Two empty lists do match.

$$[16] \quad \frac{\begin{array}{rcl} \text{empty}(Tl_1) & = & \text{true}, \\ \text{empty}(Tl_2) & = & \text{true} \end{array}}{\text{match}(Tl_1,\ Tl_2)\ =\ \text{true}}$$

Two lists match when their first elements match and the remaining elements match, discarding white space.

$$[19] \quad \frac{\begin{array}{rcl} \text{empty}(Tl_1) & = & \text{false}, \\ \text{empty}(Tl_2) & = & \text{false}, \\ \text{fun}(\text{first}(Tl_1)) & \neq & \text{w}, \\ \text{fun}(\text{first}(Tl_2)) & \neq & \text{w}, \\ \text{match}(\text{first}(Tl_1), \text{first}(Tl_2)) & = & \text{true} \end{array}}{\text{match}(Tl_1,\ Tl_2)\ =\ \text{match}(\text{rest}(Tl_1), \text{rest}(Tl_2))}$$

Two equivalent terms do match.

$$[20] \quad \text{match}(T,\ T) = \text{true}$$

Default case: no match.

$$[21] \quad \text{match}(T_1,\ T_2) = \text{false} \quad \textbf{otherwise}$$

A.17 Misc

imports AsFixParseTrees Annotations[A.19]
exports
context-free syntax

"abbreviation" "(" ATerm "," ATerm ")"	$\to$ ATerm
"is-empty" "(" ATermList ")"	$\to$ Bool
"ins-ws" "(" ATermList ")"	$\to$ ATermList
"remove-empty-grammars" "(" ATerm ")"	$\to$ ATerm
"contains" "(" ATermList "," ATerm ")"	$\to$ Bool

variables
$Abbrevs \to$ ATerm
equations

Find production that belongs to an abbreviation.

$$[01]\quad \frac{\text{fun}(Abbrevs) \;=\; \text{abbreviations}}{\text{abbreviation}(T, Abbrevs) \;=\; \text{abbreviation}(T, Abbrevs[\text{pairs}])}$$

$$[02]\quad \frac{\text{empty}(Tl) \;=\; \text{true}}{\text{abbreviation}(T, Tl) \;=\; [\,]}$$

$$[03]\quad \frac{\begin{array}{r}\text{empty}(Tl) \;=\; \text{false},\\ \text{first}(Tl) \;=\; [T, T_1]\end{array}}{\text{abbreviation}(T, Tl) \;=\; T_1}$$

$$[04]\quad \frac{\begin{array}{r}\text{empty}(Tl) \;=\; \text{false},\\ \text{first}(Tl) \;=\; [T_1, T_2],\\ T \;\neq\; T_1\end{array}}{\text{abbreviation}(T, Tl) \;=\; \text{abbreviation}(T, \text{rest}(Tl))}$$

Test for emptiness of list Tl discarding any white space.

[05] is-empty(Tl) = for-all(`"f-is-empty"`, Tl)
[06] holds(`"f-is-empty"`, T) = false **when** fun(T) $\neq$ w
[07] holds(`"f-is-empty"`, T) = true **when** fun(T) = w

Insert w(" ") between all elements in list.

[08] ins-ws([T, Ts]) = T : w(`"␣"`) : ins-ws([Ts])
[09] ins-ws([T]) = [T]
[10] ins-ws([ ]) = [ ]

Remove empty grammars from a module. These grammars can become empty due to the deletion of a sort or production declaration according to an origin specification. Observe that an import section cannot become empty due to an origin specification and is left unchanged here.

[11] remove-empty-grammars(T)
$= T$[sections $*=$ `"remove-empty-grammars"`]

$$\text{[12]}\quad \frac{\begin{array}{rcl} \text{fun}(T) \equiv \text{exports} \\ \vee\ \text{fun}(T) \equiv \text{hiddens} & = & \text{true}, \\ \text{term}(T[\text{grammars}]) & = & Tl, \\ \text{is-empty}(Tl) & = & \text{true} \end{array}}{\text{apply}(\texttt{"remove-empty-grammars"},\ T) \ = \ \text{w}(\texttt{""})}$$

[13] apply(`"remove-empty-grammars"`, T) $= T$ **otherwise**

Test whether T is member of a list

[14] contains([ ], T) = false
[15] contains([T_1], T_2) $= T_1 \equiv T_2$

[16] contains([T_1, Ts], T_2)
$=$ if $T_1 \equiv T_2$ then true else contains([Ts], T_2) fi

A.18 ATerm-Trees-Ann

imports Annotations$^{A.19}$
exports
context-free syntax

"map-sons-ann" "(" ATerm "," ATerm ")"	→ ATerm
"put-sons-ann" "(" ATerm "," ATermList ")"	→ ATerm
"treplace-ann" "(" ATerm "," ATerm "," ATerm ")"	→ ATerm

equations

This functions has the same behaviour as "map-sons" (Van den Brand *et al.*, 1997c), but it collects (and removes) the annotation from its subterms ("sons(T)") and adds them to T

$$ann(T) = \cup_{i \in sons(T)} ann(i)$$

[01] map-sons-ann(Fun, T) = put-sons-ann(T, map(Fun, sons(T)))

Collect (and remove) annotations from Tl and add them to T.

$$\text{[02]}\quad \frac{\text{foldl}(\texttt{"select-annotations"}, [\,], Tl) \ = \ [\,]}{\text{put-sons-ann}(T,\ Tl) \ = \ \text{put-sons}(T,\ Tl)}$$

$$[03]\quad \frac{\begin{array}{r}\text{foldl}(\texttt{"select-annotations"}, [\,], \mathit{Tl}) \;=\; \mathit{Tl}',\\ \text{map}(\texttt{"remove-annotations"}, \mathit{Tl}) \;=\; \mathit{Tl}'',\\ \text{put-sons}(T, \mathit{Tl}'') \;=\; T'\end{array}}{\text{put-sons-ann}(T, \mathit{Tl}) \;=\; T' \mathbin{@} \mathit{Tl}'}$$

Add annotation from T to Tl.

$$[04]\quad \frac{\text{ann}(T) \;=\; \mathit{Tl}_1}{\text{apply}(\texttt{"select-annotations"}, \mathit{Tl}, T) \;=\; \mathit{Tl} +\!\!+\; \mathit{Tl}_1}$$

Remove annotation from T.

[05] $\text{apply}(\texttt{"remove-annotations"}, T) = \text{term}(T)$

This function has the same behaviour as "treplace" (Van den Brand *et al.*, 1997c), but it will annotate T with the union of the annotations of its sub-terms. After these annotations are collected, they are removed from the sub-terms.

$$[06]\quad \frac{\text{holds}(\mathit{Pred}, T) \;=\; \text{true}}{\text{treplace-ann}(\mathit{Pred}, \mathit{Fun}, T) \;=\; \text{apply}(\mathit{Fun}, T)}$$

$$[07]\quad \frac{\text{holds}(\mathit{Pred}, T) \;=\; \text{false}}{\begin{array}{l}\text{treplace-ann}(\mathit{Pred}, \mathit{Fun}, T) \;=\\ \text{map-sons-ann}(\texttt{"treplace-ann"}(\mathit{Pred}, \mathit{Fun}), T)\end{array}}$$

[08] $\text{apply}(\texttt{"treplace-ann"}, \mathit{Pred}, \mathit{Fun}, T) = \text{treplace-ann}(\mathit{Pred}, \mathit{Fun}, T)$

A.19 Annotations

imports ATerm-Trees
equations

These where missing in the current ATerm library (Van den Brand *et al.*, 1997c), but are required by the renaming tool.

[01] $\text{fun}(\mathit{ACon}\ \mathit{Ann}) = \mathit{ACon}$
[02] $\text{fun}(\mathit{Tl}\ \mathit{Ann}) = [\,]$
[03] $\text{fun}(\mathit{AFun}\ \mathit{Ann}) = \mathit{AFun}$
[04] $\text{fun}(\mathit{AFun}(\mathit{Ts})\ \mathit{Ann}) = \mathit{AFun}$

Appendix B

Conversion to SDF2 Renaming

In this appendix a specification is given that translates a renaming module to a SDF2 compatible renaming.

The function 'sdf2' removes all origin specifications from a given renaming module and adds the originating modules to the list of imported modules. A module name will be included in an import section at most once.

Example B.0.1 Consider the renaming module `N.ren` that defines a renaming on the module `M`. It contains renamings with origin specifications to instantiate the parameters 'P', 'Q' and 'R':

N.ren:

```
M [
     P => Int  <Integers>
     Q => Bool <Booleans>
     R => Int  <Integers>
  ]
```

The function 'sdf2' will convert the renaming module `N.ren` to:

```
exports
  imports Integers
          Booleans
          M [ P => Int Q => Bool R => Int]
```

Observe that the renamings with origin specification are replaced by normal renamings and that each originating module is appended to the import section. Note that the module `Integers` is imported only once, although it occurs in a renaming with origin specification twice. □

B.1 Sdf2-renaming

imports Sdf-renamings[A.2]
exports
  sorts Section ImpSection Grammar Imports
  context-free syntax

"sdf2" "(" Import ")"	$\to$ Section
Sdf-Id	$\to$ Import
Import$*$	$\to$ Imports
"imports" Imports	$\to$ ImpSection
"exports" Grammar	$\to$ Section
ImpSection	$\to$ Grammar

hiddens
  context-free syntax

"origin-spec" "(" Renaming$*$ ")"	$\to$ Imports	
"sdf2" "(" Renaming$*$ ")"	$\to$ Renamings	
"norm" "(" Imports ")"	$\to$ Imports	
Imports "++" Imports	$\to$ Imports	{**left**}
Renaming "++" Renamings	$\to$ Renamings	

  variables

Id	$\to$ Sdf-Id
"*Renaming$*$*"	$\to$ Renaming$*$
Symbol [*0-9*]$*$	$\to$ Symbol
Production [*0-9*]$*$	$\to$ Production
Imports [*0-9*]$*$	$\to$ Import$*$
Renaming	$\to$ Renaming

equations

Conversion of renaming for ASF+SDF modules to compatible renaming in SDF2. We remove all origin specifications and create a new import section from them. Finally, we normalize the import sections in order to remove any double imported modules.

$$[01]\quad \frac{\text{norm}(\text{origin-spec}(\mathit{Renaming}^*)) \;=\; \mathit{Imports}}{\begin{array}{l}\text{sdf2}(\mathit{Id}\,[\mathit{Renaming}^*]) \;= \\ \text{exports imports } \mathit{Imports} \mathbin{+\!\!+} \mathit{Id}\ \text{sdf2}(\mathit{Renaming}^*)\end{array}}$$

$$[02]\quad \text{origin-spec}() =$$

$$[03]\quad \text{origin-spec}(\mathit{Symbol}_1 \Rightarrow \mathit{Symbol}_2\ \mathit{Renaming}^*) = \text{origin-spec}(\mathit{Renaming}^*)$$

$$[04]\quad \text{origin-spec}(\mathit{Production}_1 \Rightarrow \mathit{Production}_2\ \mathit{Renaming}^*) = \text{origin-spec}(\mathit{Renaming}^*)$$

$$[05]\quad \begin{array}{l}\text{origin-spec}(\mathit{Symbol}_1 \Rightarrow \mathit{Symbol}_2 < \mathit{Id} > \mathit{Renaming}^*) \\ =\ \mathit{Id} \mathbin{+\!\!+} \text{origin-spec}(\mathit{Renaming}^*)\end{array}$$

[06] $\text{origin-spec}(\mathit{Production}_1 \Rightarrow \mathit{Production}_2 < \mathit{Id} > \mathit{Renaming}^*)$
$= \mathit{Id} +\!\!+ \text{origin-spec}(\mathit{Renaming}^*)$

[07] $\text{sdf2}() = []$

[08] $\text{sdf2}(\mathit{Symbol}_1 \Rightarrow \mathit{Symbol}_2\ \mathit{Renaming}^*)$
$= \mathit{Symbol}_1 \Rightarrow \mathit{Symbol}_2 +\!\!+ \text{sdf2}(\mathit{Renaming}^*)$

[09] $\text{sdf2}(\mathit{Production}_1 \Rightarrow \mathit{Production}_2\ \mathit{Renaming}^*)$
$= \mathit{Production}_1 \Rightarrow \mathit{Production}_2 +\!\!+ \text{sdf2}(\mathit{Renaming}^*)$

[10] $\text{sdf2}(\mathit{Symbol}_1 \Rightarrow \mathit{Symbol}_2 < \mathit{Id} > \mathit{Renaming}^*)$
$= \mathit{Symbol}_1 \Rightarrow \mathit{Symbol}_2 +\!\!+ \text{sdf2}(\mathit{Renaming}^*)$

[11] $\text{sdf2}(\mathit{Production}_1 \Rightarrow \mathit{Production}_2 < \mathit{Id} > \mathit{Renaming}^*)$
$= \mathit{Production}_1 \Rightarrow \mathit{Production}_2 +\!\!+ \text{sdf2}(\mathit{Renaming}^*)$

[12] $\mathit{Imports}_1 +\!\!+ \mathit{Imports}_2 = \mathit{Imports}_1\ \mathit{Imports}_2$

[13] $\mathit{Renaming} +\!\!+ [\mathit{Renaming}^*] = [\mathit{Renaming}^*\ \mathit{Renaming}]$

[14] $\text{norm}(\mathit{Imports}_1\ \mathit{Id}\ \mathit{Imports}_2\ \mathit{Id}\ \mathit{Imports}_3)$
$= \text{norm}(\mathit{Imports}_1\ \mathit{Id}\ \mathit{Imports}_2\ \mathit{Imports}_3)$

[15] $\text{norm}(\mathit{Imports}) = \mathit{Imports}$ **otherwise**

Appendix C

Tools Included in the Renaming Process

`dependency-list.perl`: Perl script used to determine the depth first order in which a renaming has to be applied to all the modules contained in the module hierarchy of a specification. The implementation of the tool is based on `get-import-graph` from the graph-tool package (Moonen, 1996b).

`dumpasfix.ctsh`: Shell script for dumping AsFix for all modules in a specification. The tool was initially created by P.A. Olivier, M.G.J. van den Brand and E. Visser at the Programming Research Group of the university of Amsterdam, but was slightly modified to be usable within the renaming tool.

`dumptermasfix`: Shell script for dumping AsFix for a term over a module. The tool was initially developed by E. Visser at the Programming Research Group of the university of Amsterdam, but was slightly modified to be usable within the renaming tool.

`get-id.sh`: Shell script that returns the module identifier of a module according to the information in the module table, or allocates a new identifier when no identifier could be found.

`global.sh`: This file contains all exported variables used to configure the renaming tool. The use of exported variables defined in a single file eases the configuration of the system.

`mklink.sh`: This shell script is used to create links from all ASF+SDF modules and renaming modules in a directory hierarchy to a single directory. It is used to create module repositories as discussed in §5.6.

`modulename.perl`: Perl script that returns the name of the module to which a renaming should be applied, according to the information in a renaming module.

`origin.perl`: The perl script returns a list of all module names that where defined in a renaming module as origin specifications.

`ren-tool-makefile`: This is the makefile used for the renaming of single modules and for updating dependency information as described in §5.2.

`ren-tool.sh`: This shell script forms the renaming tool that is executed by the user. This script will perform all necessary steps required for the application of one or more renamings.

`rename-module.sh`: This shell script forms the executable specification of the renaming specification in Appendix A. This script is based on the automatically generated script `int-term` but adapted to remove hard-coded paths.

`rename.sh`: This shell script is executed to apply a single renaming to a module. It creates the AsFix representation of the module and executes the stand-alone specification.

`split-equations.perl`: This perl script is used to obtain the semantical part from the AsFix representation of a module.

`split-sections.perl`: This perl script is used to obtain the syntactical part from the AsFix representation of a module.

Bibliography

Bergstra, J. A. (1987). *Terminologie van algebraïsche specificaties*. Kluwer. In Dutch.

Bergstra, J. A. and Sellink, A. (1996). Sequential data algebra primitives. Technical Report P9602, University of Amsterdam, Programming Research Group.

Bergstra, J. A., Heering, J., and Klint, P. (1989). The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series. ACM Press with Addison-Wesley.

Bergstra, J. A., Heering, J., and Klint, P. (1990). Module algebra. *Journal of the ACM*, **37**(2), 335–372.

Bergstra, J. A., Bethke, I., and Rodenburg, P. (1995). A propositional logic with 4 values: true, false, divergent and meaningless. *J.Applied Non-Classical Logics*, **5**(2), 199–217.

Bert, D. and Jacquet, P. (1992). Design and reuse of software components based on algebraic specification techniques. In *Methods and Tools for Software Reuse*, ERCIM Workshop Reports, pages 17–40.

Van den Brand, M., Heering, J., and Klint, P. (1997a). Renovation of the old ASF+SDF Meta-Environment — current state of affairs. In A. Sellink, editor, *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97).*

Van den Brand, M. G. J. and Visser, E. (1996). Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, **5**(1), 1 – 41.

Van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997b). AsFix. A structured data format for representation of parse trees with an extensive library with generic, language independent functionality. In particular a format for ASF+SDF specifications.

Van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997c). ATerms: Representing structured data for exchange between heterogeneous tools. Technical report, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J., Kuipers, T., Moonen, L., and Olivier, P. (1997d). Implementation of a prototype for the new ASF+SDF Meta-Environment. In A. Sellink, editor, *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97)*.

Budd, T. (1991). *An Introduction to Object-Oriented Programming*. Addison Wesley.

Constantopoulos, P., Dör, M., and Vassiliou, Y. (1992). Repositories for software reuse: The software information base. In *Methods and Tools for Software Reuse*, ERCIM Workshop Reports, pages 17–40.

Van Deursen, A. (1991). A numerical package for ASF+SDF. Technical report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam. unpublished document.

Van Deursen, A., Heering, J., and Klint, P., editors (1996). *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore.

Didrich, K. (1994). Bibliotheca Opalica – A Document on Structured Use and Abuse. Technical report, TU Berlin.

Van der Graaf, M. (1997). *A specification of Box to HTML in ASF+SDF*. Master's thesis, University of Amsterdam, Programming Research Group.

Groenboom, R. (1997). *Formalizing Knowledge Domains*. Ph.D. thesis, Rijksuniversiteit Groningen.

Groenboom, R. and Renardel de Lavalette, G. R. (1996). Formalizing ANesthesiology. Final Report FAN-project (Preliminary version) CSN 9601, Department of Computer Science, University of Groningen.

Guttag, J. V. and Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag. see: www.research.digital.com/SRC/larch/toc.html.

Heering, J. (1992). Implementing higher-order algebraic specifications. In D. Miller, editor, *Proceedings of the 1992 Workshop on the lambda Prolog Programming Language*, pages 141–157, University of Pennsylvania, Philadelphia.

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

Kamperman, J. F. T. and Walters, H. R. (1993). ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

Kamperman, J. F. T. and Walters, H. R. (1995). The new ASF compiler — An exercise in self-applicability. In M. G. J. Van den Brand, A. van Deursen, T. B. Dinesh, J. Kamperman, and E. Visser, editors, *Proceedings of ASF+SDF'95: a workshop on Generating Tools from Algebraic Specifications*, pages 271–293. Technical Report P9504, Programming Research Group, University of Amsterdam.

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

Klint, P. (1995). *The ASF+SDF Meta-Environment User's Guide*. Centrum voor Wiskunde en Informatica (CWI), Amsterdam. Available as ftp://ftp.cwi.nl/pub/gipe/reports/SysManual.ps.Z.

Loeckx, J., Ehrich, H., and Wolf, M. (1996). *Specification of Abstract Data Types*. Wiley Teubner.

Mauw, S. and Veltink, G. J. (1990). A process specification formalism. *Fundamenta Informaticae*, **XIII**, 85–139.

Moineau, T. and Gaudel, M. C. (1992). Software reusability through formal specification. In *Methods and Tools for Software Reuse*, ERCIM Workshop Reports, pages 17–40.

Moonen, L. (1996a). *Data Flow Analysis for Reverse Engineering*. Master's thesis, University of Amsterdam, Programming Research Group.

Moonen, L. (1996b). *Graph tools*. available at http://adam.wins.uva.nl/l̃eon/graph-tools/.

Pepper, P. (1991). The Programming Language OPAL (1st edition). Technical Report 91–10, TU Berlin.

Rouaix, F. (1992). Functiors: Declarative programming in a software library. In *Methods and Tools for Software Reuse*, ERCIM Workshop Reports, pages 125–135.

Stevens, W., Myers, G., and Constantine, L. (1974). Structured design. *IBM Systems Jounal*, **13**(2). Reprinted in Edward Youdon (Ed) Classics in Software Engineering, Prentice-Hall, Englewood Cliffs, NJ, 1979.

Thunem, S. and Sindre, G. (1992). Development with and for reuse: Guidelines from the REBOOT project. In *Methods and Tools for Software Reuse*, ERCIM Workshop Reports, pages 2–16.

Uhl, J. and Schmid, H. A. (1990). *A Systematic Catalogue of Reusable Abstract Data Types*. Sprintger-Verlag.

Visser, E. (1992). Syntax and static semantics of Eiffel, a case study in algebraic specification techniques. Technical report, University of Amsterdam, Programming Research Group.

Visser, E. (1994). *ASF+SDF to LaTeX, User Manual*. University of Amsterdam, Programming Research Group.

Visser, E. (1997). *Syntax Definition for Language Prototyping*. Ph.D. thesis, University of Amsterdam.

Walters, H. R. (1994). aslib 1.0: A library facitility for parameterized ASF+SDF specifications. unpublished document.

Walters, H. R. and Kamperman, J. F. T. (1993). Higher order techniques in first order algebraic specifications. Technical report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

Van Wamel, J. J. (1993). A library for PSF. Technical Report P9301, Programming Research Group, University of Amsterdam.

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site
(`http://www.wins.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.wins.uva.nl`,
directory `pub/programming-research/reports/`.

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*