Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 110 (2004) 149–168

www.elsevier.com/locate/entcs

An Action Environment

Mark van den Brand

Department of Software Engineering, CWI
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands
and
Instituut voor Informatica en Electrotechniek, Hogeschool van Amsterdam
Weesperzijde 190, NL-1097 DZ Amsterdam, The Netherlands

Jørgen Iversen, Peter D. Mosses

BRICS & Department of Computer Science [1]
University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

Abstract

Some basic programming constructs (e.g., conditional statements) are found in many different programming languages, and can often be included without change when a new language is designed. When writing a semantic description of a language, however, it is usually not possible to reuse parts of previous descriptions without change.
This paper introduces a new formalism, ASDF, which has been designed specifically for giving reusable action semantic descriptions of individual language constructs. An initial case study in the use of ASDF has already provided reusable descriptions of all the basic constructs underlying Core ML.
The paper also describes the Action Environment, a new environment supporting use and validation of ASDF descriptions. The Action Environment has been implemented on top of the ASF+SDF Meta-Environment, exploiting recent advances in techniques for integration of different formalisms, and inheriting all the main features of the Meta-Environment.

Keywords: ASF+SDF, ASDF, action semantics, modularity, reuse, language environment

[1] Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

1 Introduction

Action Semantics [18] is a practical framework for describing the dynamic semantics of programming languages. The part of an action semantic description (ASD) concerned with any particular construct is independent of what other constructs are included in the described language, so ASDs enjoy a high degree of inherent modularity, and can easily be extended or modified. It is also possible to reuse parts of the ASD of one language in the ASD of another, without change. With the conventional modular structure of an ASD, however, it is not usually possible to reuse entire modules, so one has to copy and paste the required parts.

Doh and Mosses [11] proposed a flatter modular structure for ASDs, with the description of each construct being a separate module. This new structure allows a complete language to be described simply by listing the names of the modules for the included constructs, and fully supports explicit reuse of parts of semantic descriptions. Doh and Mosses formulated their modules in ASF+SDF [10], and used the ASF+SDF Meta-Environment [6] for checking them.

The approach of Doh and Mosses was feasible, but the direct use of ASF+SDF carried a considerable notational overhead. In this paper, we introduce a new action semantic description formalism, ASDF, which has been designed specifically for giving reusable descriptions of individual language constructs. We also report on the Action Environment, a new environment supporting use and validation of ASDF descriptions. The Action Environment has been implemented on top of the ASF+SDF Meta-Environment, exploiting recent advances in techniques for integration of different formalisms [5], and inheriting all the main features of the Meta-Environment. This was feasible due to the open architecture of the Meta-Environment. The Meta-Environment has a component-based architecture which allows an easy connection of new components in a fairly easy manner. In order to transform the ASF+SDF Meta-Environment into the Action Environment, a number of new components had to be defined, and plugged into the Meta-Environment. The most important ones were the components that took care of translating ASDF into ASF+SDF. In the future, further ASDF-specific components, such as a type-checker, interpreter, and compiler, are to be connected.

Overview:

Section 2 recalls ASF+SDF and the Meta-Environment. Section 3 gives a brief outline of Action Semantics, focusing on modularity. Section 4 introduces ASDF and the Action Environment. Section 5 recalls the architecture of the

Meta-Environment, and explains the novel techniques used to integrate ASDF. Section 6 mentions some related work. Section 7 concludes.

2 ASF+SDF

ASF+SDF is a general-purpose, executable, algebraic specification language. Its main application area has hitherto been in the modular definition of the syntax and the static semantics of (programming) languages, but it has also been used for the modular definition of (dynamic) action semantics of languages (see Section 3) and for defining translations between languages.

As the name indicates, the ASF+SDF formalism is a combination of two previous formalisms: ASF, the Algebraic Specification Formalism [2,10], and SDF, the Syntax Definition Formalism [12]. SDF is used to define the concrete syntax of a language, whereas ASF is used to define conditional rewrite rules; the combination ASF+SDF allows the syntax defined in the SDF part of a specification to be used in the ASF part, thus supporting the use of so-called 'mixfix notation' in algebraic specifications. ASF+SDF allows specifications to be divided into named modules, facilitating reuse and sharing (as in SDF).

In the rest of this section, both SDF and ASF will be discussed, as well as the interactive programming environment that supports the use of ASF+SDF: the ASF+SDF Meta-Environment [6].

2.1 Syntax Definition Formalism

The Syntax Definition Formalism SDF is a declarative formalism used to define concrete syntax of languages: not only programming languages, e.g., Java and COBOL, but also specification languages, e.g., CASL, Elan, and Action Semantics. In contrast to (E)BNF-like formalisms, SDF allows a modular definition of grammars. Furthermore, SDF does not impose a specific class of grammars, like LL(k), LR(k), etc., but allows arbitrary, cycle-free, context-free grammars — the grammars may even be ambiguous. The choice of the class of arbitrary context-free grammars enables the modular definition of grammars, because only this class is closed under union. Although the full power of arbitrary context-free grammars is hardly necessary when defining the syntax of a programming language (except for languages like COBOL, PL/I, etc.), modularity is essential for reuse of specific language constructs in various language definitions.

An SDF definition consists of a collection of modules where modules may import other modules. The import mechanism offers primitive parameterisation and symbol-renaming facilities. This is demonstrated in Figure 1: The formal parameter `X` of the module "`containers/List`" is instantiated with

```
module ListOfIntegers
imports basic/Integers containers/List[Integer]
   ...

module containers/List[X]
imports basic/Booleans basic/Integers
   ...
```

Fig. 1. A small SDF definition demonstrating the parameterisation mechanism

```
module basic/Integers
imports basic/Booleans

exports
   sorts NatCon Integer
   lexical syntax
      [0-9]+ -> NatCon
   context-free syntax
      NatCon                  -> Integer
      Integer "+" Integer     -> Integer {left}
      Integer "-" Integer     -> Integer {left}
      Integer "*" Integer     -> Integer {left}
      "(" Integer ")"         -> Integer {bracket}

   context-free priorities
      Integer "*" Integer -> Integer >
      {left:  Integer "+" Integer -> Integer
              Integer "-" Integer -> Integer}

   lexical restrictions
      NatCon -/- [0-9]

hiddens
   ...
   variables
      "Int"[0-9]* -> Integer
```

Fig. 2. An SDF module of the Integers

the actual parameter `Integer`. The imported modules are automatically exported; the syntax defined in the module can either be exported or hidden.

Figure 2 demonstrates the basic features for defining lexical syntax, context-free syntax, associativities, and priorities.

2.2 Algebraic Specification Formalism

The Algebraic Specification Formalism ASF provides conditional equations, where also negations of equations are allowed as conditions. The concrete syntax defined in the corresponding SDF module and in the transitive closure

```
equations

  [] 0 + Int = Int
  [] Int + 0 = Int
  [] 1 + 1 = 2
  [] 1 + 2 = 3
  ...
  [] Int * 0 = 0
  [] Int * 1 = Int
  [] gt(Int2, 1) = true
     ====>
     Int1 * Int2 = Int1 + Int1 * (Int2 - 1)
   ...
```

Fig. 3. Some ASF equations for the Integers

of the imported modules (only the exported sections, of course) can be used when writing the conditional equations of an ASF module.

2.3 The ASF+SDF Meta-Environment

The development of ASF+SDF specifications is supported by an interactive integrated programming environment, the ASF+SDF Meta-Environment [6]. This programming environment provides syntax directed editing facilities for both the SDF and ASF parts of modules as well as for terms, well-formedness checking of modules, and visualisation facilities of the import graph and parse trees. The environment offers all kinds of refactoring operations at the specification level: renaming of modules, copying of modules, etc. Furthermore, a library of predefined primitive data structures, e.g., Booleans, Integers, Strings, Lists, Sets, etc., is available. The library contains also a growing collection of grammars of programming and specification languages, e.g., Java, C, CASL, SDF itself, etc.

The user interface of the ASF+SDF Meta-Environment is shown in Figure 4. Modules defining the concrete syntax of Pico (a toy language) have been opened. In the left part we see a tree-structured view of the modules, whereas the right pane shows the graph with import relations of the modules.

3 Action Semantics

The main aim of Action Semantics [18] is that descriptions of programming languages should be as easy as possible to work with. Action semantic descriptions (ASDs) scale up smoothly from small idealised languages to full languages [7,21], and they have a high degree of comprehensibility (regarding not only perspicuity of notation, but also underlying concepts). They also

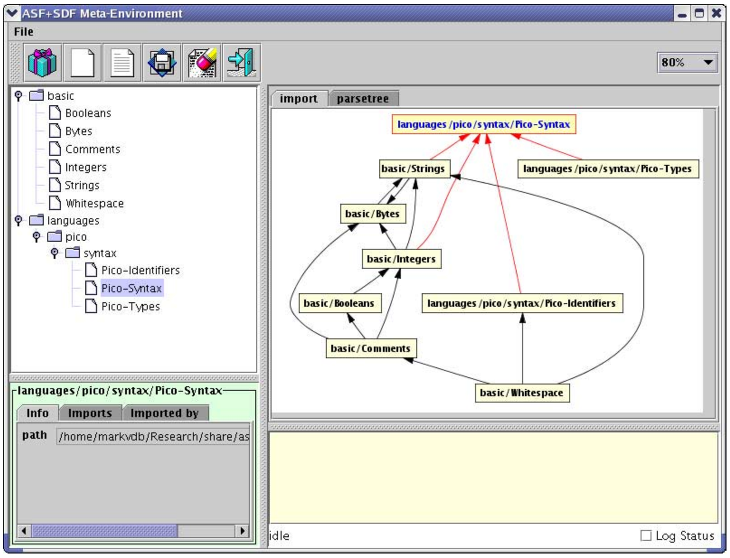

Fig. 4. GUI of the ASF+SDF Meta-Environment.

have inherently good modularity, and can be extended or modified without reformulation of the parts of the description concerned with the unchanged constructs.

Action Semantics (AS) is a hybrid of Denotational Semantics and Operational Semantics, and combines the best features of both approaches. As in a conventional denotational description, inductively defined semantic functions map programs (and declarations, expressions, statements, etc.) compositionally to their denotations, which model their behaviour. The difference is that here, denotations are *actions*, and expressed in Action Notation (AN), which is itself defined operationally (originally [18, App. C] using Structural Operational Semantics, later [19] in a more modular style).

The inherent modularity of ASDs comes from the design of AN, not from their explicit division into named modules. For instance, applications of action combinators remain valid (and meaningful) when the actions that they combine are enriched with new facets of behaviour; and similarly regarding the data processed by actions. The original version of AN [18, App. B] was rather large, but the revised version, AN-2 [15], is much more economical, and the size of the AN-2 kernel notation is comparable to that of the notation used in the monadic style of denotational semantics (e.g., as used in [16]).

Although the division of an ASD into named modules is not essential for extensibility and modifiability, the overall modular structure is of crucial significance for *reusability*. Until recently, the structure was hierarchical, being a refinement of the usual division of semantic descriptions into sections dealing

```
module Exp
  imports Values
  exports
    sorts Exp
    context-free syntax
       "evaluate" "[[" Exp "]]" -> Action
    variables
       "E"[1-9]? -> Exp
```

Fig. 5. Module Exp in SDF

with abstract syntax, auxiliary semantic entities, and definitions of semantic functions. The abstract syntax module had a submodule for each sort of construct (expressions, statements, etc.), and similarly for the semantic functions; the submodules for the auxiliary entities were similarly focused on particular sorts of data. The implementation of a previous environment for AS based on the ASF+SDF Meta-Environment, the ASD Tools [9], relied on this structure to distinguish between the different kinds of submodules.

Doh and Mosses [11] realized that this conventional modular structure was a major impediment to explicit reuse of parts of ASDs. For example, suppose that an AS for Standard ML has already been given [21], and we are writing an AS for Java [7], wanting to reuse the semantic equations for all the constructs that the two languages have in common (modulo notational changes regarding abstract syntax). We cannot import the entire module for expressions from the ASD of ML for reuse in the ASD of Java, since this would include ML constructs not found in Java (e.g., anonymous function abstractions). We could of course simply copy and paste the individual semantic equations – but this leaves no explicit indication of the fact that the two languages have constructs in common, and readers of the two descriptions would have to compare the details to discover exactly which the common constructs are.

Doh and Mosses proposed changing the modular structure of ASDs to support an incremental approach to semantics. The main idea was to introduce a *separate module for each individual construct*, specifying both its abstract syntax and the semantic equation defining its AS, and referring to auxiliary modules for the required auxiliary entities. There was also a separate module for each sort of construct, but in contrast to the previous structure, this module did not combine a particular selection of constructs: it merely introduced the syntactic sort, meta-variables ranging over it, and the symbol used for the corresponding semantic function.

ASF+SDF was used for writing ASDs with the new modular structure [11], and a demonstration involving a small case study has been given [17].

Figure 5 shows the SDF module for the sort `Exp`, introducing the semantic

```
module Exp/if-then-else
  imports Exp
  exports
    context-free syntax
       "if" Exp "then" Exp "else" Exp -> Exp
       Bool -> Value

equations

[10] evaluate [[ if E1 then E2 else E3 ]] =
       evaluate [[E1]] then
       select(
         ( given true then evaluate [[E2]] ) or
         ( given false then evaluate [[E3]] ) )
```

Fig. 6. Module Exp/if-then-else in ASF+SDF

function `evaluate` and the meta-variables ranging over `Exp`. The module introducing the sort `Action` is imported indirectly, via the auxiliary module `Values`.

Figure 6 shows the ASF+SDF module for an ASD of the usual conditional expression where the condition is a boolean-valued expression. It uses SDF to introduce the mixfix notation used for the abstract syntax of the construct, and to require `Bool` to be included in the sort `Value`. Notice that it is necessary to import modules for all the sorts of constructs involved in the construct – here, only `Exp`. The `equations` part uses ASF to define the action semantics of the construct, using the notation introduced in the SDF part of the module and that originating in imported modules.

4 ASDF

ASDF is a language specification formalism designed to make it easier to write ASDs of single language constructs.

4.1 Formalism

We have previously used plain ASF+SDF for writing ASDs, as described in Section 3. The advantage of using ASF+SDF was that it allowed ASDs to be prototyped using the Meta-Environment. Furthermore other tools, like an action interpreter, action type-checker, etc., could be connected to the Meta-Environment. However, using ASF+SDF for writing small modules describing single language constructs was not optimal, and this prompted the development of ASDF. The main problems with using ASF+SDF were related to the

```
Exp   ::=  Ide | if Exp then Exp else Exp |
           Exp Exp | fn Ide => Exp
Dec   ::=  val Ide = Exp | Dec Dec
```

Fig. 7. Small subset of ML

cumbersome notation:

- When using a syntactic sort, e.g., `Exp`, in a production rule, the module introducing the syntactic sort had to be explicitly imported (see Figure 6). Also modules describing AN had to be imported, since it was not part of the SDF language.
- The declaration of metavariables ranging over sorts is somewhat tedious (see Figure 5).
- ASF+SDF requires many keywords and can be misleading, e.g., the signature of a semantic function is introduced by the words '`context-free syntax`'.

ASDF solves these problems, making specifications easier both to write and read.

```
module SmallML

imports

  Exp/Ide
  Exp/Cond
  Exp/App-Seq
  Exp/Abs
  Dec/Bind-Val
  Dec/Accum
```

Module 1

```
module Exp

requires

  E : Exp

  Datum ::= Val

semantics

  evaluate: Exp -> Action
```

Module 2

```
module Exp/Ide

syntax Exp ::= val(Ide)

semantics

  evaluate val(I) =
            give the val bound-to I
```

Module 3

```
module Exp/Cond

syntax Exp ::= cond(Exp, Exp, Exp)

requires Val ::= Bool

semantics

  evaluate cond(E1, E2, E3) =
       evaluate E1 then
       maybe check the boolean then
       evaluate E2 else
       evaluate E3
```

Module 4

```
module Exp/App-Seq

syntax Exp ::= app-seq(Exp, Exp)

requires Val ::= Func | func-no-apply

semantics

  evaluate app-seq(E1,E2) =
     evaluate E1 and-then
     evaluate E2 then
   (apply(action(the func#1), the val#2)
    else (throw func-no-apply))
```

Module 5

```
module Exp/Abs

syntax Exp ::= abs(Ide, Exp)

requires Val ::= Func

semantics

  evaluate abs(I, E) =
      give func(closure(
             furthermore bind(I, the val)
             scope evaluate E))
```

Module 6

```
module Dec

requires

  D : Dec

  Datum ::= Bindings

semantics

  declare : Dec -> Action
```

Module 7

```
module Dec/Bind-Val

syntax Dec ::= bind-val(Ide, Exp)

semantics

  declare bind-val(I, E) =
      evaluate E then bind(I, the val)
```

Module 8

```
module Dec/Accum

syntax Dec ::= accum(Dec+)

semantics

  declare accum(D) = declare D

  declare accum(D D+) =
     declare D before declare accum(D+)
```

Module 9

```
module Data/Func

requires

  Func ::= func(action: Action)
```

Module 10

A semantic description of a language consists of a collection of ASDF modules and a mapping from the concrete syntax used in the language to the abstract syntax described in the modules. Figure 7 shows a small subset of ML and the Modules 2 to 10 can be used to describe the constructs found in the ML subset. The import-relation between the modules can be seen in the screenshot in Figure 8, where the modules at one level import the modules on the lower level, if there is an edge connecting them.

Comparing Module 4 with the module found in Figure 6 one immediately notices that we use abstract syntax with prefix constructors instead of con-

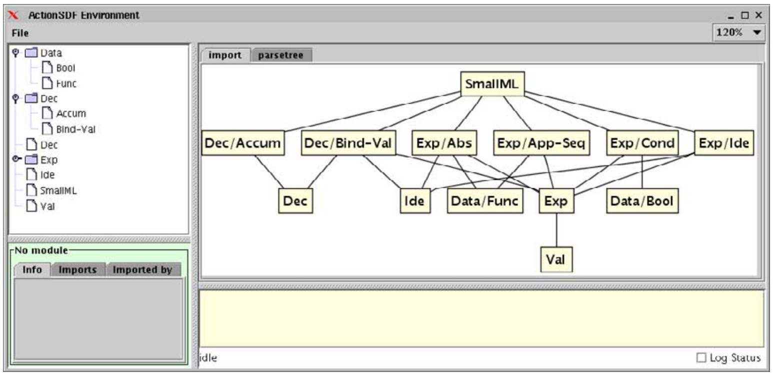

Fig. 8. The Action Environment

equations

```
...
[cond] map(if E1 then E2 else E3) =
                 cond(map(E1), map(E2), map(E3))
...
[let] map(fn I => E) = abs(I, map(E))
...
[seq] map(D1 D2) = accum(map(D1) map(D2))
```

Fig. 9. Mapping concrete to abstract syntax

crete syntax, when describing constructs in ASDF. The advantage of using prefix constructors for abstract syntax is greater reusability. For instance, a description of the if-then-else expression from Standard ML might be reused for describing the '?:' expression in Java, since they have the same compositional structure and intended interpretation even though their concrete syntax differs. Part of the definition of the function *map* that maps from concrete ML syntax to abstract syntax is shown in Figure 9.

An ASDF module consists of a name (after the keyword **module**) and three optional sections. The **syntax** section defines the abstract syntax of the construct. This is illustrated in Module 3 with the identifier expression constructor `val`, which takes an Identifier (*Ide*) as argument. When writing production rules the separator '`::=`' is used, instead of the unfamiliar '$\longrightarrow$' found in SDF.

The **requires** section is used for introducing data sorts, operators, and variables used in the **semantics** section. This is illustrated in Module 6, where the sort *Val* is extended with the sort *Func*, such that actions can produce functions. The syntax for declaring variables is illustrated in Module 7, where 'D : *Dec*' declares the variable D to range over the syntactic sort *Dec*. When declaring the variable X to range over a sort S the variables Xn, $X*$, and $X+$, where n is a positive integer, are automatically declared to range over the sorts S, S^*, and S^+. The use of these variables is illustrated in Module 4 and Module 9.

Module 10 illustrates how data sorts and operators are introduced. The declaration '*Func* `::= func(action:` *Action*`)`', results in the type *Func*, and the data operators *func* and *action* becoming available in actions, such that we can write actions as '`give the func`' and '`give action(...)`' (the syntax of the action `give` is '`give` *DataOp*', where *DataOp* contains among other terms '`the` *Sort*'). The operator *func* is a data constructor, and *action* selects the action component of such data.

The semantic function, mapping the abstract syntax construct introduced in the **syntax** section to an action, is defined, using an equation, in the **semantics** section. In the equation, terms from AN and imported modules can be used. For instance, in Module 5 the semantic function contains action combinators and constants, together with the value `func-no-apply` from the **requires** section, and the action semantics sort `func`, declared in the imported module *Data/Func*. Notice that it is possible to define the function using more than one equation, as illustrated in Module 9. The **semantics** section can also contain the signature of a semantic function, as we see in Modules 2 and 7. It is required that the signature of a function, used in a module, is defined in the same module or an imported module. Since the syntax of a semantic function depends on the syntax declared in other parts of the module, parsing a module must be done in two steps, where the first step builds a parsetable based on the **syntax** and **requires** section. More about this in Section 4.2 and 5.2.

Syntactic sorts used in the **syntax** section result in implicit imports, so for instance in Module 8 the modules *Dec* (Module 7), *Ide* (not shown), and *Exp* (Module 2) are automatically imported. Implicit imports are also generated from the sorts used in the **requires** section, with the difference that only syntactic sorts used on the right hand side of the production results in imports, and the imported modules always start with *Data/*, for instance Module 6 imports *Data/Func* (Module 10). The automatically imported modules, like *Exp* or *Data/Func*, may provide further sorts than those that caused their importation.

ASDF also allows explicit imports. This is mostly used in the top module that imports all the modules used to describe a language (see Module 1). Another rarely used feature is the "back door" for including raw SDF, the **sdf-section**.

The modules, presented in this section, are simplified versions of the modules used in a semantic description of core ML, which can be found at [14]. The description contains both ASDF modules and ASF+SDF modules mapping ML concrete syntax to abstract syntax.

4.2 Environment

Being built on top of the ASF+SDF Meta-Environment, the Action Environment inherits most of its features (described in Section 2.3).

On the surface the differences between the Meta-Environment and the Action Environment seem negligible. A few menus have changed, because we have not yet implemented all the operations available for ASF+SDF for ASDF (e.g., changing module name and imports). When editing a module one notices more differences, since the syntax directed editor now uses an ASDF grammar for parsing. Furthermore, the grammar defined in a module (and in the modules it imports) is used when parsing the semantic equations in a module. This means that when highlighting a piece of syntax in a semantic function, the editor displays its sort according to the grammar defined in the syntax- and requires-sections of the same module (and imported modules). As in the Meta-Environment, it is possible to employ the given language specification for parsing and rewriting terms over the language. Due to the way we implemented the Action Environment, everything concerning terms works as in the Meta-Environment.

Combining the ASF+SDF Meta-Environment and the Action Environment gives us tool support for mapping the concrete syntax of a language to actions. The idea is that one describes a mapping from concrete syntax to abstract syntax in ASF+SDF (see Figure 9), and the mapping from abstract syntax to actions is described using ASDF as we have already seen.

We are planning to implement support for working simultaneously with ASF+SDF and ASDF modules in the Action Environment such that full language descriptions are supported. Future plans also include integrating different tools into the Action Environment. Integrating an existing action evaluator will allow us to evaluate programs written in a language that we are designing. An action compiler will turn the environment into a compiler generator. Integration of an existing type-checker for action semantic functions will give us a better check of the well-formedness of the ASDF modules, and thereby the correctness of the ASD of the language. All in all, the Action Environment combined with other tools should provide a particularly useful environment for developing semantic descriptions and documenting the design of programming languages.

5 Implementation Overview

The Action Environment is built on top of the ASF+SDF Meta-Environment. Discussing the implementation details of the Action Environment involves discussing the architecture of the Meta-Environment.

5.1 *ASF+SDF Meta-Environment Architecture*

The Meta-Environment has a layered architecture as displayed in Figure 10. In this section we will discuss each of these layers in more detail. The first step towards a layered design of the ASF+SDF Meta-Environment is discussed in [5]. That paper discusses how ASF can be replaced by another rewriting formalism. This development has been taken a step further, resulting in the architecture discussed here.

Kernel layer

The kernel of the Meta-Environment is completely language independent. It consists of the software coordination architecture, the ToolBus [3], which takes care of all the communication between the components that make up the Meta-Environment. The ToolBus allows a full separation of coordination and computation, it is a programmable software bus where the coordination between the components is formally described using a Process Algebra based formalism. The computation is performed within the connected components, which can be implemented in any programming language. The exchange of data between the components is based on a representation format, ATerms [4], specially designed for representing tree-like data structures. This formalism provides maximal subterm sharing and efficient linearisation operations.

Besides the ToolBus the kernel of the Meta-Environment consists of a parser, text and structure editors, graphical user interface components, a term store to store parse tables and parse trees, a component which takes care of the communication with the file system, etc. Each of the components is fully language independent and will be instantiated via the next layer, which provides language specific functionality. The kernel is fully prepared to deal with modular languages and specification formalisms.

SDF layer

The next layer instantiates the kernel Meta-Environment with SDF functionality. This is achieved by adding SDF-specific components to the kernel and by adding actions, via buttons and clickable icons in the user interface, to activate editors for SDF modules. Examples of SDF-specific components

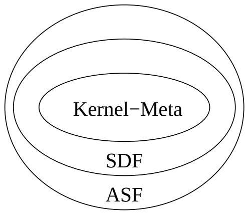

Fig. 10. The layered architecture of the ASF+SDF Meta-Environment

are the SDF parse table, the import relation calculator, and the parse table generator. The latter is needed because of the fact that SDF is designed to describe syntax of programming languages, and in order to use these language descriptions it is necessary to generate parse tables for parsing programs. Furthermore, the term store has to be instantiated in such a way that both the parse trees of SDF modules and their corresponding parse tables can be stored.

ASF layer

This layer extends the SDF Meta-Environment with ASF functionality. Again this is achieved by adding ASF-specific components and actions to activate for instance editors for ASF modules. An example of an ASF-specific component is a component, which extends every SDF specification with the syntax rules to parse the ASF equations; in this way the user defined syntax in the equations is obtained. Using SDF in combination with ASF poses some restrictions on the grammar rules one can write in SDF, e.g., the separator in a list may only be a literal and not an arbitrary symbol. These restrictions are checked by an ASF+SDF-syntax-checker. Finally, this layer provides an ASF checker to check the well-formedness of the equations, and an ASF interpreter and compiler are added to the SDF Meta-Environment. The term store has to be extended to store ASF modules, corresponding parse tables, etc., as well.

Implementation

Figure 11 shows an abstraction of the kernel Meta-Environment with each of the extensions described above. In this section we will briefly describe how we achieve these extensions in a flexible way.

The messages that can be received by the kernel layer are known in advance, simply because this part of the system is fixed. The reverse is not true: the generic part can make no assumptions about the functionality provided by the other layers.

We identify messages that are sent from the kernel of the Meta-Environment to the extensions as so-called *hooks*. The SDF layer can and will introduce new hooks for the next layers. Each instance of the environment should *at*

Hook	**Description**
`environment-name(Name)`	The main GUI window will display this name
`extensions(Sig, Sem, Term)`	Declares the extensions of different file types
`stdlib-path(Path)`	Sets the path to a standard library
`top-sort(Sort)`	Declares the top non-terminal of a specification

Table 1
The Meta-Environment hooks: hooks that parameterise the GUI

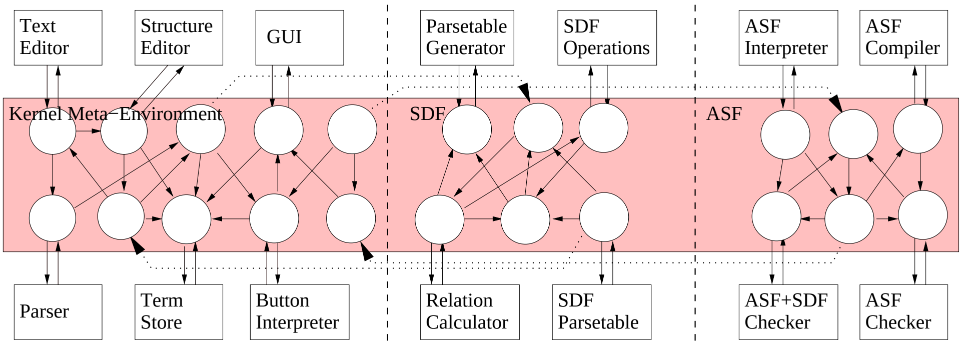

Fig. 11. The layered implementation of the ASF+SDF Meta-Environment

least implement a receiver for each of these hooks. Implementing these hooks involves writing small pieces of ToolBus specifications. Table 1 shows a few kernel hooks. They are all related to the GUI and editors. The dashed arrows in the Figure 11 between the kernel layer and the ASF or SDF layer denote the hooks and the service requests.

Adding a layer involves some implementation effort. Of course, the components themselves have to be implemented. In a number of cases it is necessary to write ToolBus scripts, but the kernel Meta-Environment also provides a powerful *button language*, which can be used to connect new components and functionality. The button language enables a flexible way of adding buttons and icons to the GUI and adding buttons to the various types of editors.

5.2 The Action Environment

In the Action Environment the layered design of the Meta-Environment is extended with an extra layer, the ASDF layer, illustrated in Figure 12. Notice that we do not replace any parts of the ASF+SDF Meta-Environment, we just extend it with an extra layer on the top. The alternative to this approach would be to replace the ASF and the SDF layer with an ASDF layer, similar to what was described in [5], but with our approach we reuse advanced technologies and thereby save implementation effort. Another way of viewing the

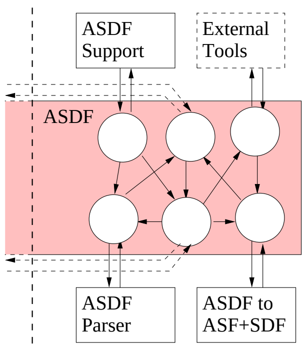

Fig. 12. The ASDF layer

ASDF layer is as an ASDF interface to the ASF+SDF Meta-Environment.

The ASDF layer consists of several components: an ASDF parser, tools for retrieving the module name and imported modules from an ASDF module, and two ASDF to ASF+SDF mappings. As with the other layers we also have to extend the term store, in this case to hold ASDF modules. Based on the grammar of the ASDF language, a parse table has been generated, which is used in the ASDF parser. The tools for getting the module name and imported modules from an ASDF module are implemented in ASF+SDF and are almost trivial (this is the *ASDF Support* component in the illustration). Here we shall focus on the generation of ASF+SDF and the possibility of adding external tools in the future.

Mapping ASDF to ASF+SDF

The Action Environment contains two mappings of ASDF to ASF+SDF. The result of one mapping is used for parsing and rewriting terms. By mapping every ASDF module to an ASF+SDF module we get the same effect, with respect to working with terms, as if we had opened the generated ASF+SDF modules in the Meta-Environment, so editing of terms is independent of the ASDF layer. The result of the other mapping is used for the second parse of the ASDF module itself (the parse that allows us to parse the semantic equations with the grammar defined in the same module). Figure 13 gives some examples, which are included in both mappings. Generating ASF+SDF makes imports explicit, for instance using the sorts S_1 and S_2 in a production rule results in imports of the modules S_1 and S_2. The ASF+SDF is generated on demand (i.e., when we need to parse a term or a module), and has to be regenerated for an ASDF module every time the module changes. The mappings to ASF+SDF are implemented in ASF+SDF; this was an obvious choice since a grammar for ASF+SDF already exists, which made it easy to construct a type-safe translation.

Integration of external tools

Due to the configurability of the Meta-Environment, it is possible to attach external tools, like an action type-checker or interpreter. This is an easy task using the button language, under the assumption that the tools just take the contents of an editor as input, and return a text string as result. It becomes more complicated when the tool needs global information (like a semantic function type-checker, which might need all defined function signatures to check a function definition), and in these cases we need to traverse the import graph to collect the necessary information from each module.

6 Related Work

An enormous amount of work has been performed in the field of defining the syntax and semantics of programming languages and systems supporting the development of such language definitions. We refer to Heering and Klint [13] for a fairly complete and up-to-date overview.

In the discussion of related work we will focus on environments which can be used to describe single language constructs in a modular way, or to give ASDs of languages.

The GEM-MEX system [1] allows description of languages using a collection of MONTAGES, a formalism based on Abstract State Machines. The idea of describing single language constructs in separate modules is encouraged by GEM-MEX, but due to the lacking modularity of the syntax formalism used (the

S_1 ::= S_2	$\Rightarrow$	imports S_1 S_2 ... context-free syntax $S_2 \rightarrow S_1$
V : S	$\Rightarrow$	imports S ... context-free syntax "V"[0-9]? $\rightarrow$ S "V*" $\rightarrow$ S* "V+" $\rightarrow$ S+
L : $S_1 \rightarrow S_2$	$\Rightarrow$	L $S_1 \rightarrow S_2$

Fig. 13. Examples of mapping ASDF to SDF

semantic descriptions of individual constructs are based on concrete syntax, and the collected syntax has to be LALR(1)) a Montage is not often reusable in practice.

The Abaco system [20] is an AS tool for programming language designers. The main components of Abaco are an algebraic specification compiler, specification editors, action libraries, action editors, and a GUI. Furthermore, it offers a help system, an action debugger and facilities to export specifications to readable output. The main component is the algebraic specification compiler, which provides syntax checking of specifications and interpretation. The Abaco system and the Action Environment have a strong resemblance, but the Action Environment offers more flexibility in adding external components by means of openness of the underlying architecture.

The action semantics of individual constructs can be presented with an object-oriented perspective [8]. Then the introduction of each syntactic sort and its corresponding semantic function is given as a class definition; the syntax of an individual construct and its action semantics are defined in a subclass that extends the class defining the sort of the construct. The use of conventional object-oriented class definitions does not allow as much to be left implicit as in ASDF, but otherwise the collections of class and subclass definitions are directly comparable to collections of modules in ASDF. However, tool support for the approach has not yet been provided.

The ASD toolset [9] supported the creation, editing, checking, and use of ASDs. This toolset had a very strong relation with an older version of ASF+SDF, and its implementation has become obsolete.

7 Conclusions and Future Work

In this paper, we have presented ASDF, a new formalism for action semantic descriptions supporting reuse of descriptions of individual constructs. We have also reported on the Action Environment, a new environment supporting the use of ASDF, and explained how it is implemented on top of the ASF+SDF Meta-Environment. Two of the authors have already carried out an initial case study in the use of ASDF and the Action Environment, providing ASDF modules for all the basic constructs underlying Core ML (submitted for publication).

Plans for future work include further case studies in the use of ASDF, and the integration in the Action Environment of existing type-checkers, interpreters, and ultimately, compiler generation.

References

[1] M. Anlauff, P. W. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In *PSI'99*, LNCS Vol. 1755, pages 40–53. Springer, 2000.

[2] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. Addison-Wesley, 1989.

[3] J. A. Bergstra and P. Klint. The discrete time ToolBus – A software coordination architecture. *Sci. Comput. Programming*, 31(2-3):205–229, 1998.

[4] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[5] M. G. J. van den Brand, P. Moreau, and J. J. Vinju. Environments for term rewriting engines for free! In *RTA 2003*, LNCS Vol. 2706, pages 424–435. Springer, 2003.

[6] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In *CC 2001*, LNCS Vol. 2027, pages 365–370. Springer, 2001.

[7] D. Brown and D. A. Watt. JAS: A Java action semantics. In *AS'99*, BRICS NS-99-3, pages 43–55. Dept. of Computer Science, Univ. of Aarhus, 1999.

[8] C. Carvilhe and M. Musicante. An object-oriented view of action semantics. In *AS 2002*, BRICS NS-02-8, pages 45–64. Dept. of Computer Science, Univ. of Aarhus, 2002.

[9] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, Univ. of Amsterdam, 1994.

[10] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing Vol. 5. World Scientific, 1996.

[11] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Programming*, 47(1):3–36, 2003.

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

[13] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, 2000.

[14] J. Iversen and P. D. Mosses. Core ML action semantics description. http://www.brics.dk/Projects/AS/LDTA-2004/.

[15] S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In *AS 2000*, BRICS NS-00-6, pages 19–36. Dept. of Comput. Sci., Univ. of Aarhus, 2000.

[16] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96*, LNCS Vol. 1058, pages 219–234. Springer, 1996.

[17] P. Mosses. Action Semantics and ASF+SDF. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[18] P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[19] P. D. Mosses. A modular SOS for Action Notation. BRICS RS-99-56, Dept. of Comput. Sci., Univ. of Aarhus, 1999.

[20] H. Moura, L. C. Menezes, M. Monteiro, P. Sampaio, and W. Cansanção. The ABACO system: An action tool for programming language designers. In *AS 2002*, BRICS NS-02-8, pages 1–8. Dept. of Computer Science, Univ. of Aarhus, 2002.

[21] D. A. Watt. The static and dynamic semantics of SML. In *AS'99*, BRICS NS-99-3, pages 155–172. Dept. of Computer Science, Univ. of Aarhus, 1999.