A Framework for Debugging
Olivier, P.A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
8 Debugging ASF+SDF specifications

ASF+SDF specifications can be executed in two ways. By interpretation as discussed in Section 5.9, or by executing a compiled specification as discussed in Chapter 7. In this chapter, we will show how interpreted specifications can be debugged using TIDE. We will present some ideas on debugging compiled specifications, and we will show how TIDE support can be added to interpreters specified in ASF+SDF.

8.1 Introduction

The ASF+SDF Meta-Environment is a versatile system that can be used to develop executable specifications. An executable specification is nothing more than a computer program written in a formal language, so it is not surprising, that it is just as hard to write a fault-free specification as it is to write a fault-free program. Debugging support for developers of ASF+SDF specifications is therefore just as important as it is for any other software developer.

In this chapter, the two main themes of this thesis will come together. In the first part, we have presented a framework for 'generic debugging'. This framework significantly reduces the effort to build new debuggers. In the second part, we showed how ASF+SDF specifications can be executed, both using interpretation and compilation. In this last chapter, we will show how support for our generic debugging framework can be added to the interpreter discussed in Section 5.9. We will also give some ideas on how debugging support can be added to specifications compiled using the ASF+SDF compiler discussed in Chapter 7.

Because of the support for user-definable syntax, the ASF+SDF Meta-Environment is very well suited for the development of new programming languages. All kinds of tools can be generated for such specified languages, based on their specification in ASF+SDF. Typical tools include parsers, syntax-directed editors, pretty printers, and type checkers. It is also possible to specify tools that can be used to execute programs written in these specified languages: compilers and interpreters. But the possibility to execute these programs implies the need for debugging support at this level as well. To explore the possibilities in this area, this chapter contains a case study that shows how a compiler specified in ASF+SDF can be
extended to instrument the code it generates with TIDE support.

8.2 Debugging Specifications

When designing TIDE support for a specific language, there are two important issues to resolve. The first is at which points during the execution of a program specific debug events have to be generated. The second issue is how to relate these events to the original source code using the cpe function discussed in Chapter 4.

To enable basic debugging support, three event ports are crucial: stopped, started, and step. The activation of stopped and started event rules is often straightforward. The real decisions involve when and how to activate event rules that have a step port, and which source coordinates should be returned by the cpe function at each activation of the step event rules.

In most imperative languages, step rules are either activated before or after execution of a single statement. The source code area that should be highlighted is most often the complete source line that contains the current instruction. In some cases this highlighting can be more precise, and the actual instruction being executed can be highlighted. Optionally, loops can generate more step events to provide a better understanding of what is going on, and to give the user more control over the execution. More step events might be generated at function entry or exit for the same purposes.

The semantics of ASF+SDF is based on innermost rewriting, as discussed in Chapter 7. This means that there is no notion of individual “statements” being executed, and so we cannot use the same debugging semantics as in the imperative case. In order to decide which semantics we want in this case, we have to take a close look at the different constructs that can appear in ASF+SDF specifications.

8.2.1 Unconditional Equations

Unconditional equations consist of a left-hand side and a right-hand side. The left-hand side consists of a term that might contain holes in the form of variables. With innermost rewriting, a term (or tree) is rewritten (or reduced) “inside out”. This means that rewriting starts at the leaves of the tree that is being reduced. The current subterm being reduced is called the redex. The equations are tried one by one until the left-hand side of one of them matches the redex. In ASF+SDF, equations are tried in no particular order, except that special default equations are only tried when all other equations fail.

When a match is found, the matching equation will be applied. This means that the variables in the left-hand side of the matching equation are assigned a value corresponding to the subterms of the redex they matched with. Variables occurring in the right-hand side will be replaced by their values resulting in what is called the reduct. The redex in the original term will then be replaced by this reduct. Note that ASF+SDF only allows bound variables in the right-hand side of equations. This means that any variable that occurs in the right-hand side
module Naturals
imports Layout
exports
sorts Nat
context-free syntax
  zero         -> Nat
  succ(Nat)   -> Nat
  plus(Nat,Nat) -> Nat

variables
  [IJKL]     -> Nat

equations
  [n1] plus(zero, I) = I
  [n2] plus(succ(I),J) = succ(plus(I,J))

Figure 8.1: Specification of successor naturals

of an equation must be assigned a value using matching in the left-hand side of that equation, or in a condition as will be shown in the next subsection. A (sub-)term for which no equations are applicable is said to be in normal form.

Figure 8.1 shows an example of unconditional equations n1 and n2 that define the semantics of the plus operator over successor integers. Note that zero and succ have no equations defined over them and therefore they are called constructors, as they have no functionality of their own. They can only be used to construct and match terms. plus has two equations, n1 and n2.

Suppose we want to reduce the term plus(succ(zero), zero). As ASF+SDF rewrites left-most innermost, the left-most occurrence of zero is reduced first. As this is a constructor, it is already in normal form, and therefore left intact. Then succ(zero) is reduced, which is also left intact because succ is also a constructor. The second occurrence of zero is also in normal form. Then the complete term is reduced. This term does not match with the left-hand side of n1, but it does match with n2, so n2 can be applied. During matching, I and J both are assigned the value zero. The redex (in this case the whole term) is replaced by the right-hand side of n2, after I and J are replaced by their values. This yields the term succ(plus(zero, zero)). At this point, the only term to which equations can be applied is plus(zero, zero). Only n1 matches with this subterm, resulting in the reduct zero. The normal form of the term now is succ(zero).
8.2.2 Conditional Equations

ASF+SDF supports conditional equations where the conditions may be both positive and negative. This means that equation n2 in the previous example can be rewritten to:

\[
\begin{align*}
[n2] & \quad I \neq \text{zero}, \\
& \quad I = \text{succ}(K), \\
& \quad L = \text{succ}(\text{plus}(K,J)) \\
& \quad \text{plus}(I,J) = L
\end{align*}
\]

The first condition, \( I \neq \text{zero} \) is a negative condition. It only succeeds when \( I \) is not equal to zero. The second condition is a positive condition that introduces a new variable. The value of \( I \) is matched against \( \text{succ}(K) \), and if the match succeeds the subterm of \( I \) that matches with \( K \) is assigned to \( K \). If the match does not succeed, the condition fails. When a side of a condition does not introduce new variables, it is first reduced before it is matched against the other side. This means that in the last condition, \( \text{succ}(\text{plus}(K,J)) \) is first reduced before its result is assigned to the new variable \( L \). Note that the first condition is superfluous in this case, because the second condition also fails when \( I \) equals zero. Note also that the last condition is guaranteed to succeed, as the match between a term and a new variable is always successful.

8.2.3 List Matching

Yet another feature of ASF+SDF that has great impact on debugging is list matching. This powerful construct introduces the notion of backtracking in the semantics of ASF+SDF. Consider the example in Figure 8.2. This example specifies a list datatype with a “half” operator that returns the first half of a list. The function “size” is not exported but only used to calculate the length of a list within this module.

Equation \( h1 \) is applicable for lists whose length is even. Equation \( h2 \) is applicable for lists whose length is odd. Both equations make essential use of backtracking. When the left-hand side of \( h1 \) matches, the list in the redex is split into two arbitrary sublists. The only condition of \( h1 \) than determines whether these parts are of equal length using the \( \text{size} \) function. If not, backtracking is used to find a different division of the list in the redex until all possible combinations are tried.

8.2.4 Stepping through Equations

Now we turn our attention to the issue of generating step events. In the previous examples a lot of individual steps were needed to reduce a simple term like \( \text{plus}(\text{succ}(\text{zero}), \text{zero}) \). It should be clear that if we generated a step event for each of these, stepping through all but the smallest reductions would take a long time. We need to keep the number
module Lists
imports Layout Naturals
exports
  sorts Elem List
  lexical syntax
    [a-z] -> Elem
  context-free syntax
    "[" { Elem "," }* "]" -> List
    "halve" "(" List ")" -> List
  variables
    "El" [\']* -> Elem
    "Els" [\']* -> { Elem "," }*
  hiddens
    context-free syntax
    "size" "(" List ")" -> Nat
  equations
    [h1] size([Els]) = size([Els'])
    halve([Els,Els']) = [Els]
    size([Els,Els']) = succ(size([Els']))
    halve([Els,Els']) = [Els]
    size([El,Els]) = succ(size([Els]))
    size([]) = zero

Figure 8.2: Specification of a list datatype with a "halve" operator

of step events as low as possible, without losing the ability to mentally trace the execution of our specification.

Unconditional Equations When applying an unconditional equation, two successive step events are generated. The first event is generated when the left-hand side of the equation is matched against the current redex and the match succeeds. In this case the left-hand side of
the equation is considered as the current point of execution. The second step event is generated after the reduct is constructed, but before it is reduced. In this case, the right-hand side of the equation is considered as the current point of execution. When reducing the term plus(succ(zero), zero) as described in subsection 8.2.1, the generated step events can be used to produce the following highlight sequence:

1. \[ \text{plus}(\text{succ}(I), J) = \text{succ}(\text{plus}(I, J)) \] When \( n_2 \) matches with the input term, its left-hand side is highlighted.

2. \[ \text{plus}(\text{succ}(I), J) = \text{succ}(\text{plus}(I, J)) \] The reduct is constructed, so the right-hand side of \( n_2 \) is highlighted.

3. \[ \text{plus}(\text{zero}, I) = I \] \( \text{plus}(\text{zero}, \text{zero}) \) will be reduced, so the next step event is generated when the left-hand side of \([n_1]\) matches with this term.

4. \[ \text{plus}(\text{zero}, I) = I \] The last step event is generated when the reduct \( \text{zero} \) is returned.

**Conditional Equations** When debugging conditional equations, more step events need to be introduced to keep track of the execution path. Especially because evaluation of conditions might involve reduction of subterms. This may cause the debugger to display the equations involved in this "subreduction". It is crucial that the user receives concrete feedback after this subreduction is finished whether the condition that started the subreduction failed or succeeded.

We have identified three locations that need to generate step events to do this tracking:

- The first location is before reduction of the left-hand side of each condition. The complete left-hand side of the condition is considered as the current point of execution.

- The second location is before reduction of the right-hand side of each condition. The complete right-hand side of the condition is considered as the current point of execution.

- The third location is before checking whether the left-hand side and right-hand side of a condition match. The (in-)equality sign of the condition is considered as the current point of execution.

It can be argued that the last step event is not needed to keep track of the execution path. When after a subreduction a condition succeeds, the left-hand side of the next condition (or the right-hand side of the equation when no more conditions are present) is highlighted,
so separate highlighting of the equality or inequality sign is not needed. A problem arises however when such a subreduction causes the condition to fail. If no more equations match with the current redex, a condition higher on the call stack could fail as well, transferring control to an equation several levels higher on the call stack.

Because this “equation hopping” can make it extremely difficult for the user to keep track of what is going on, we have decided to inform the user explicitly of success or failure of all conditions. Transitions between equations now only occur between equations one level below or above the current equation on the call stack.

To make the above discussion more comprehensible, we present the steps involved in reduction of the term $\text{plus}(\text{succ}(\text{zero}), \text{zero})$ using the second version of $n_2$.

1. $\text{plus}(I,J) = L$ 
   The left-hand side of $n_2$ is highlighted to indicate a successful match.

2. $I \neq \text{zero},$ 
   The left-hand side of the first condition is highlighted.

3. $I = \text{zero},$ 
   The left-hand side does not need reducing, so the right-hand side is highlighted next.

4. $I \neq \text{zero},$ 
   The two sides of the condition are matched.

5. $I = \text{succ}(K),$ 
   After this successful match, the left-hand side of the next condition is highlighted.

6. $I = \text{succ}(K),$ 
   followed by the right-hand side

7. $L = \text{succ}(\text{plus}(K,J)),$ 
   and the equality sign as the sides are matched.

8. $L = \text{succ}(\text{plus}(K,J))$ 
   Now the left-hand side of the last condition of $n_2$ is highlighted.

9. $L = \text{succ}(\text{plus}(K,J))$ 
   The right-hand side of the last condition is the only one that needs reducing. Before this reduction starts, the right-hand side is first highlighted. After this, we start reduction of the subterm $\text{plus}(K,J)$, where $K$ and $J$ are both equal to zero.

10. $\text{plus}(I,J) = L$ 
    The left-hand side of both $n_1$ and $n_2$ match with the term $\text{plus}(\text{zero}, \text{zero})$. In this example we assume that $n_2$ is tried first, so the left-hand side of $n_2$ is highlighted.

11. $I \neq \text{zero},$ 
    Next the left-hand side of the first condition of $n_2$ is highlighted,

12. $I = \text{zero},$ 
    followed by its right-hand side

13. $I \neq \text{zero},$ 
    and the inequality sign.

14. $\text{plus}(\text{zero},I) = I$ 
    As the above condition fails, the $n_1$ is tried next to reduce $\text{plus}(\text{zero}, \text{zero})$.

15. $\text{plus}(\text{zero},I) = I$ 
    This equation succeeds immediately, so the reduct $\text{zero}$ is returned.

16. $L = \text{succ}(\text{plus}(K,J))$ 
    Now execution continues with the evaluation of the third condition of $n_2$ that started the reduction of $\text{plus}(\text{zero}, \text{zero})$ in step 9.
17 \text{plus}(\text{zero}, I) = [I] \quad \text{Reduction now finishes and returns the normal form succ(zero).}

\textit{Debugging of Equations with List Matching.} When an equation performs list matching, backtracking can occur when more than one match is possible. After the first match has been tried but this match caused a condition to fail, the next match is tried. Although the semantics of ASF+SDF do not predefine a particular order in which list matches are tried, most implementations will use a predefined ordering to simplify the implementation. The interpreter on which these examples are based orders the list matches by trying the smallest match first, from left to right. So, for instance, when we match the list pattern \(L_1, L_2\) where \(L_1\) and \(L_2\) are list variables, against the list \(a, b\), there are three possible matches which are tried in the following order:

1. \(L_1 = L_2 = a, b\)
2. \(L_1 = a , L_2 = b\)
3. \(L_1 = a, b , L_2 = \)

By generating a single step event each time a new match is found, the user can keep track of the execution path. To demonstrate this, we will show the steps involved in reduction of the term \(\text{halve}([a, b])\) using the equations in Figure 8.2.

1 \[\text{halve}([\text{Els}, \text{Els}']) = [\text{Els}]\] The left-hand side of \(hl\) matches with the input term for the first time. The input list is split into two sublists. The empty list is assigned to \(\text{Els}\). The rest of the list: \(a, b\) is assigned to \(\text{Els}'\).

2 \[\text{size}([\text{Els}]) = \text{size}([\text{Els}'])\] The left-hand side of the condition is reduced first. This involves reduction of \(\text{size}([])\).

3 \[\text{size}([]) = \text{zero}\] \(s_2\) matches, resulting in the reduct \(\text{zero}\).

4 \[\text{size}([]) = \text{zero}\]

5 \[\text{size}([\text{Els}]) = \text{size}([\text{Els}'])\]

The right-hand side of \(hl\) is tried, which involves reducing \(\text{size}([a, b])\).

6 \[\text{size}([\text{Els}, \text{Els}']) = \text{succ}(\text{size}([\text{Els}']))\] \(s_1\) matches.

7 \[\text{size}([\text{Els}, \text{Els}']) = \text{succ}(\text{size}([\text{Els}']))\] Construction of the right-hand side of \(s_1\) involves reduction of \(\text{size}([b])\).

8 \[\text{size}([\text{Els}, \text{Els}']) = \text{succ}(\text{size}([\text{Els}']))\] Again \(s_1\) is the only equation that matches.

9 \[\text{size}([\text{Els}, \text{Els}']) = \text{succ}(\text{size}([\text{Els}']))\] This time, the construction of the reduct involves reduction of \(\text{size}([])\).
Onl y  s 2 matches.

The right-hand side of the condition of h 1 is also in normal form.
The left-hand side of the condition reduced to zero, the right-hand side to succ(succ(zero)), so the condition fails.

At this point, the next match is tried. This time, the singleton list a is assigned to Els, and the singleton list b is assigned to Els'.

Evaluating the left-hand side of the condition of h 1 involves reduction of size([a]).

The left-hand side of s 1 matches. a is assigned to El, and the empty list is assigned to Els.

Construction of the right-hand side involves reduction of size([]). s 2 matches.

s 2 is used to reduce size([]) to zero.

Now the right-hand side of the condition of h 1 needs to be reduced.
The left-hand side of s 1 matches. b is assigned to El, and the empty list is assigned to Els.

The construction of the reduct involves reduction of size([]). s 2 matches.

s 2 is used to reduce size([]) to zero.

Both sides of the condition of h 1 have been reduced to a normal form, resulting in the condition succ(zero) = succ(zero) which succeeds.

At this point the reduct can be constructed: [a], which represents the first half of the input list [a, b].
8.2.5 TIDE Support in the Interpreter

The interpreter discussed in Section 5.9 can easily be instrumented to generate step events as discussed above. The only real problem is how to determine the source coordinates as returned by the \texttt{cpe} function. Fortunately, all layout information is still available in the AsFix representation of the equations (see Section 5.4.2). This allows us to \textit{annotate} the AsFix representation of the equations with positional information before rewriting starts. All parts that are “interesting” for debugging purposes are annotated: left-hand side, right-hand side, and both sides of conditions. To store these annotations we use the generic ATerm annotation mechanism discussed in Chapter 6.

When an event rule is activated, the positional information is retrieved from the construct that caused the event rule to be fired. This construct can either be the left-hand side of the equation, the right-hand side, or the left-hand side or right-hand side of a condition. This positional information is then returned by invocations of the \texttt{cpe} function by the evaluation of the event condition or event actions. Figure 8.3 shows the TIDE support in the interpreter “in action”.

![Figure 8.3: The List example in TIDE](image)

As always in TIDE, the user can right-click on a variable to view its current value.
8.2.6 TIDE Support in the Compiler

So far we have shown how we implemented basic debugging support for the ASF+SDF interpreter. We would also like to implement debugging support for specifications compiled with the ASF+SDF compiler presented in Chapter 7. In this section we will discuss two possible approaches to reach this goal.

Generating instrumented code  The first approach is to modify the compiler to generate code that is instrumented with extra debugging statements. Every code chunk that might give rise to the activation of step events can be instrumented with calls to the debugger.

This approach suffers from a problem that is common to most compilers: optimizations performed during compilation can influence the debugging semantics. In the case of the ASF+SDF compiler, identical left-hand sides and conditions of different equations can potentially result in generation of a single chunk of code. We cannot relate such a chunk back to a single location in the source code, so debugging is hampered. It is therefore important to disable this kind of optimizations when generating debug code.

Specification transformation  A much cleaner but potentially less efficient approach is to implement the debugging of compiled specifications using transformation of the original specification. Figure 8.4 shows a transformed version of the equations of the naturals example, based on the conditional version of \([n2]\).

Note that each call to the step function contains the filename and coordinates of the corresponding area in the original source code. The coordinates consist of two pairs of integers, the first pair supplying the line and column of the start of the area, and the second one representing the line and column of the end of the area.

The syntax of the step function and its argument are specified in a special module Debug that is imported by the transformed module. The Debug module also contains a default definition of the step function:

\[
\text{[step] step(Info) = true}
\]

Basically, for each position that could possibly generate a step event, a new condition is introduced. These new conditions are guaranteed to succeed, as the step function always returns true. This means that the semantics of the original specification are kept intact. The calls to the step function takes the positional information of the step event as an argument.

Without any support from the run time system, the transformed specification behaves exactly like the original specification. When the run time system is augmented with debugging support, the invocation of the step function is caught, so the appropriate step events can be activated.

At first glance, the transformational approach looks very clean. No changes to the compiler are needed, only a small extension of the run time system is required. There are some disadvantages however. As is clear from this small example, the amount of extra code that
8 / DEBUGGING ASF+SDF SPECIFICATIONS

equations

\[\text{step("Naturals:2,7-2,20") = true,}\]
\[\text{step("Naturals:2,23-2,24") = true}\]
\[\text{plus(zero, I) = I}\]

\[\text{step("Naturals:8,7-8,17") = true,}\]
\[\text{step("Naturals:4,7-4,8") = true,}\]
\[\text{X0 = zero,}\]
\[\text{step("Naturals:4,12-4,16") = true,}\]
\[\text{I := X,}\]
\[\text{step("Naturals:5,7-5,8") = true,}\]
\[\text{X1 = succ(K),}\]
\[\text{step("Naturals:5,11-5,18") = true,}\]
\[\text{I = X1,}\]
\[\text{step("Naturals:6,7-6,8") = true,}\]
\[\text{X2 = succ(plus(K,J)),}\]
\[\text{step("Naturals:6,11-6,26") = true,}\]
\[\text{L = X2}\]
\[\text{step("Naturals:7,19-7,20") = true}\]
\[\text{plus(I,J) = L}\]

Figure 8.4: A transformed version of the successor naturals

needs to be generated is substantial. In general, the size of the specification is increased by a factor 2.

In [Alb97] the transformational approach to debugging ASF+SDF specifications is pursued further. This work shows that if we add more debug features, the complexity of this approach increases dramatically. For instance, it is shown that it is possible to add transformations that enable the debugger to access the value of variables. In order to do this extra arguments need to be generated for the step function. But to do this, we also need to generate extra syntax rules to inject the sort of each variable into one “super sort” so variables can be passed to the step function in a type-safe manner.
8.3 Adding Debugging Support to a Specified Compiler

The ASF+SDF Meta-Environment is first and foremost a tool to specify tools for programming languages. Some of these tools like compilers and interpreters are primarily aimed at executing programs written in specified languages. But if we are able to execute programs using these specified tools, a natural need arises to debug these programs as well. In this section we will present a case study that uses TIDE for this purpose. We show how an existing compiler for a toy language called PICO can be extended in such a way that this compiler is capable of generating code augmented with TIDE support. In this way, TIDE provides us with a fully featured interactive debugger for PICO programs at a fraction of the cost it would take to develop a fully featured PICO debugger from scratch.

8.3.1 The PICO to C Compiler

PICO is a small Pascal-like language whose key syntax elements can be written down in about two dozen lines of SDF, as shown in Figure 8.5. Note that a number of modules are imported which are not shown here. These modules define the syntax of identifiers, integers and string constants.

Figure 8.6 shows an example of a PICO program that calculates the factorial of the number four. We will use this example throughout this section to explain the code generation of the PICO to C compiler and the generation of debug code.

The PICO to C compiler compiles the factorial PICO program into a single C function shown in Figure 8.7. This function is compiled together with a small run time library for PICO (less than 150 lines of code) yielding an executable program. The run time library defines the PICONATURAL macro that is used to register variables, and the finish function that prints the value of all variables at the end of the program. It is also responsible for providing a main function that calls the actual picomain function after doing some initialization.

When we run the resulting executable we get the following output:

```
~/Research/pico/non-debug> ./fac
natural input = 1
natural output = 24
natural repnr = 1
natural rep = 12
~/Research/pico/non-debug>
```

which is a list of all variables and their values at the end of the run.

8.3.2 Adding TIDE Support

To add TIDE we need to do two things. Extra debug statements need to be inserted in the generated C code, and the run time library needs to be extended to handle these extra debug statements and to make the value of variables available to TIDE.
module Pico-syntax
imports Pico-Identifiers Pico-Integers Pico-Strings Types
exports
sorts PROGRAM DECLS ID-TYPE STATEMENT EXP
context-free syntax
"begin" DECLS {STATEMENT ";"} * "end"  -> PROGRAM
"declare" {ID-TYPE ","} * ";"  -> DECLS
PICO-ID ":" TYPE  -> ID-TYPE
PICO-ID ":=" EXP  -> STATEMENT
"if" EXP "then" {STATEMENT ";"} * "else" {STATEMENT ";"} * "fi"  -> STATEMENT
"while" EXP "do" {STATEMENT ";"} * "od"  -> STATEMENT
PICO-ID  -> EXP
PICO-NAT-CON  -> EXP
PICO-STR-CON  -> EXP
EXP "+" EXP  -> EXP {left}
EXP "-" EXP  -> EXP {left}
EXP "||" EXP  -> EXP {left}
"(" EXP ")"  -> EXP {bracket}

Figure 8.5: The syntax of PICO

Figure 8.8 shows what the generated code looks like when debug statements are inserted at compile time. Close inspection of this code reveals that it is exactly the same as the code in Figure 8.7, except that one debugstep(\textit{xx}) statement has been added for each PICO statement in the original PICO program. Moreover, the number \textit{xx} corresponds to the line number of the original PICO statement.

At first glance, it might seem straightforward to insert these extra debugstep statements, but unfortunately there is a problem. The current version of ASF+SDF in which the PICO to C compiler has been implemented does not make layout information available to the specification writer. This means that it is not possible to determine the current line number during compilation, so it is impossible to insert the correct line numbers in the debugstep calls. We circumvented this problem by first pre-processing the original PICO program before compiling it. During this pre-processing stage we simply insert a `!' character at the start of every line. This makes it possible for the compiler to count `!' characters to determine the current line number.

Unfortunately this solution implies that the original specification of the PICO to C compiler has to change drastically before it can generate debug information. First of all, the syntax
Adding Debugging Support to a Specified Compiler

begin
declare
    input : natural,
    output : natural,
    repnr: natural,
    rep: natural;

input := 4;
output := 1;
while input - 1 do
    rep := output;
    repnr := input;
    while repnr - 1 do
        output := output + rep;
        repnr := repnr - 1
    od;
    input := input - 1
od
end

Figure 8.6: Calculation the factorial of four in PICO

definition of PICO itself has to be adapted to allow the '!' characters, as shown in Figure 8.9. Note that we do not allow newlines in expressions to simplify the compiler specification.

In addition the actual compiler specification needs to be adapted to cope with the appearance of '!' characters, as well as to insert debugstep calls at the correct places. This results in a 50% increase in the size of the compiler, most of which is due to the extra complexity of keeping track of the current line number.

The PICO run time library is extended with debugstep function, in such a way that calls to this function result in step events in TIDE. In addition, each call to debugstep also generates a location event to support breakpoints in PICO programs. The run time library is also extended to support the var function in debug actions and conditions to make the value of variables available to TIDE. These extensions to the PICO run time library are implemented using only 120 lines of code.

This case study shows that adding debugging support to an existing compiler specified in ASF+SDF can be straightforward but for one problem: keeping track of positional information during compilation. This problem can be solved by preprocessing the original source
void picomain ( )
{
    PICONATURAL ( input ) ;
    PICONATURAL ( output ) ;
    PICONATURAL ( repnr ) ;
    PICONATURAL ( rep ) ;
    input = 4 ;
    output = 1 ;
    while ( input - 1 ) {
        rep = output ;
        repnr = input ;
        while ( repnr - 1 ) {
            output = output + rep ;
            repnr = repnr - 1 ;
        }
        input = input - 1 ;
    }
    finish() ;
}

Figure 8.7: The factorial example compiled to C code

code, but this severely increases the complexity of the original compiler specification. Ideally, this positional information should be accessible directly from an ASF+SDF specification, for instance by exposing layout information at the specification level.

After tackling this problem, it takes a minimal amount of work to connect a language run time system to TIDE, immediately yielding a fully functional debug implementation. Figure 8.10 shows a screenshot of a PicO debugging session using TIDE. It is clear that the conventional method of developing a new debugger for a programming language would require considerably more human resources than are needed using the TIDE approach.

8.4 Related Work

In [Deu94, Tip95, DKT96] a technique called origin tracking is used to generate languagespecific debugging tools for algebraic specifications. In origin tracking, at each reduction step from input term to normal form relations are maintained between subterms in the redex and subterms in the reduct. When combined properly, these relations can for instance be used to find the relation between PicO statements and corresponding C statements in the generated C program in our case study. It will be interesting to see if we can combine origin tracking with
void picomain ( )
{
    PICONATURAL ( input );
    PICONATURAL ( output );
    PICONATURAL ( repnr );
    PICONATURAL ( rep );
    debugstep(8);
    input = 4;
    debugstep(9);
    output = 1;
    debugstep(10);
    while ( input - 1 ) {
        debugstep(11);
        rep = output;
        debugstep(12);
        repnr = input;
        debugstep(13);
        while ( repnr - 1 ) {
            debugstep(14);
            output = output + rep;
            debugstep(16);
            repnr = repnr - 1;
        }
        debugstep(18);
        input = input - 1;
    }
    finish();
}

Figure 8.8: The factorial example compiled to C code with debug information

our techniques to derive TIDE support automatically without the need to adapt the original compiler specification.

Another example of a system that is capable of generating debugging tools is described in [BMS87]. In this work specifications of the denotational semantics of a programming language are compiled into a functional language. Debugger behavior can be expressed using a set of built-in debugging concepts, not unlike TIDE. Examples of these notions are trace functions, breakpoint definitions and state inspection primitives.
module Pico-syntax
imports Pico-Identifiers Pico-Integers Pico-Strings Types
exports
sorts PROGRAM DECLS ID-TYPE STATEMENT EXP SOL L LSTAT
lexical syntax
"!" -> SOL
context-free syntax
SOL* -> L
L "begin" DECLS { LSTAT ";" }* L "end" -> PROGRAM
L STATEMENT -> LSTAT
L "declare" { ID-TYPE "," }* ";" -> DECLS
L PICO-ID ":" TYPE -> ID-TYPE
PICO-ID ":=" EXP -> STATEMENT
"if" EXP L "then" {LSTAT ";" }* L
"else" {LSTAT ";" }* L "fi" -> STATEMENT
"while" EXP L "do" {LSTAT ";" }* L "od" -> STATEMENT
PICO-ID -> EXP
PICO-NAT-CON -> EXP
PICO-STR-CON -> EXP
EXP "+" EXP -> EXP {left}
EXP "-" EXP -> EXP {left}
EXP "||" EXP -> EXP {left}
"(" EXP ")" -> EXP {bracket}

Figure 8.9: The syntax of PICO where each line can start with a `!` character. Note that the occurrence of line breaks is restricted to specific positions. For instance, a single expression cannot be spread over several lines.

In [Ber91] the operational semantics of a programming language are extended with *semantic display rules* to generate *animators*. Different views on the execution of a program can be generated using different display rules. Static views are based on the abstract syntax tree of a program, dynamic views are based on the program state during execution.

8.5 Conclusions

We have added TIDE support to the ASF+SDF Meta-Environment on multiple levels. TIDE is used to create a debugger for ASF+SDF specifications. We also showed that it is possible to create a debugger for specified languages. In all these cases, the effort needed to build a
Conclusions

In addition a C library is used that implements generic TIDE debug-adapter support. This library consists of 900 lines of C code and can be reused for every debug-adapter written in C (see also Figure 4.4).

Figure 8.11: Code increase when TIDE support is added

Table 8.4: Size of code and run-time support

<table>
<thead>
<tr>
<th>Component</th>
<th>Implementation language</th>
<th>Original size (lines of code)</th>
<th>Tide support (lines of code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASF+SDF interpreter</td>
<td>C</td>
<td>3445</td>
<td>185†</td>
</tr>
<tr>
<td>Pico2C compiler</td>
<td>ASF+SDF</td>
<td>465</td>
<td>30</td>
</tr>
<tr>
<td>Pico2C run time support</td>
<td>C</td>
<td>183</td>
<td>105†</td>
</tr>
</tbody>
</table>

†: In addition a C library is used that implements generic TIDE debug-adapter support. This library consists of 900 lines of C code and can be reused for every debug-adapter written in C (see also Figure 4.4).

These figures show that the code needed to actually add TIDE support to a system can be very small. If we compare the two run time systems (ASF+SDF interpreter and the Pico2C run time support), it is interesting to notice that both need about the same amount of extra code to support the same TIDE features, even though the ASF+SDF interpreter run time system is vastly more complex than the Pico2C run time system.

145
On the negative side, we observe that lack of line number information on the level of ASF+SDF specifications complicates the addition of TIDE support in specified compilers. Given the fact that we have used TIDE to build realistic debuggers for languages like C and Java, we speculate that adding TIDE support to compilers specified in ASF+SDF is within reach once the line number problem has been solved.