Animators for Generated Programming Environments

Frank Tip*

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
tip@cwi.nl

Abstract. Animation of execution is a necessary feature of source-level debuggers. We present a framework where animators are *generated* from existing algebraic specifications of interpreters. To this end, a pattern-matching mechanism is used in conjunction with origin tracking, a generic tracing technique. The generation of animators is illustrated using an example language named CLaX, a Pascal relative. We study how our approach can be extended to the generation of source-level debuggers and algorithmic debuggers from specifications of interpreters.

1 Introduction

We study animators for generated programming environments. An *animator* is a tool which visualizes program execution; typically, it highlights the statement that is currently executing. Animators are especially useful for (automated) debugging and tutoring.

We use the ASF+SDF Meta-environment [14] to generate programming environments, consisting of syntax-directed editors, type-checkers, and interpreters, from algebraic specifications. Specifications are written in the formalism ASF+SDF, a combination of the Algebraic Specification Formalism ASF [4], and the Syntax Definition Formalism SDF [12]. Specifications can be executed in the ASF+SDF Meta-environment as term rewriting systems [15].

Instead of explicitly extending specifications with animation facilities, we *generate* animators from existing specifications of interpreters. We present a generic mechanism for defining animators, consisting of two parts. First, we define the *events* we are interested in. A typical example of such an event is the execution of a statement. Second, the *subjects* of the events, i.e., the language constructs involved, are determined. Events are defined by way of a pattern-matching mechanism. Origin tracking [9] is used for determining the subjects.

We illustrate our techniques using an example language named CLaX, a Pascal relative. In [10], the specification of a programming environment for this language is described in detail.

Finally, we study how our approach can be extended to the generation of source-level debuggers and algorithmic debuggers from specifications of interpreters. It is shown how, for CLaX, several debugger features can be defined.

* Partial support received from the European Communities under ESPRIT project 5399: Compiler Generation for Parallel Machines – COMPARE.

2 Related Work

Often, animation is dealt with in an ad-hoc manner, such as keeping track of line-numbers. Below, we discuss some generic approaches.

The program animation system PASTIS [16] allows the animation of Fortran, C, and C++ source code without requiring changes to the program. The system is built as an extension of the GNU source-level debugger, gdb [19]. This debugger sends program data to an animation server. *Visualization scripts* serve to determine which data is to be extracted from the program, and to which animators this data is to be sent. Information is represented by way of a relational model. *Animation scripts* define how information is to be visualized: both textual and graphical display of information is possible. Moreover, several animators can execute in parallel. The main difference with our approach is that PASTIS relies on the ad-hoc extension of a debugger. As a result, only languages that are supported by gdb can be supported by PASTIS. By contrast, we derive animators from specifications. This means that, at least in principle, we can support any language for which a specification is written.

In the context of the PSG system [3], a generator for language-specific debuggers was described in [2]. Language-specific compilers are generated by compiling denotational semantics definitions to a functional language. A standard, language-independent interpreter is used to execute the generated functional language fragments. Correspondences between the abstract syntax tree and the generated fragments are maintained during compilation. To define debuggers, a set of built-in debugging concepts is available. In particular, trace functions are provided for the visualization of execution. Other notions enable one to inspect the state of the interpreter, and to define breakpoints.

Bertot [6] contributes a technique called *subject tracking* to the specification language Typol [7, 13] for animation and debugging purposes. A key property of Typol specifications is that the meaning of a language construct is expressed in terms of its sub-constructs. A special variable, *Subject*, serves to indicate the language construct currently processed. This variable may be manipulated by the specification writer, when different animation or debugging behavior is required.

Berry [5] presents an approach where animators are generated from structured operational semantics definitions. These specifications are augmented with *semantic display rules* which determine how to perform animation when a particular semantic rule is being processed. Various views of the execution of a program can be obtained by defining the appropriate display rules. Static views consist of parts of the abstract syntax tree of a program, and dynamic views are constructed from the program state during execution. As an example of a dynamic view, the evaluation of a control predicate may be visualized as the actual truth value it obtains during execution.

Apart from differences in the underlying specification formalisms, there are two major differences between our approach and Berry's. First, we only consider the highlighting of the language construct which is currently being executed, whereas Berry also considers very advanced animation features such as dynamic call graphs, and reversible execution. The price he pays for this is the fact that

he needs to store the entire evaluation history. This contrasts with our method which only involves a small linear run-time space overhead, and no global history at all. Second, Berry's Animator Generator generates animators as stand-alone tools, whereas our animators are smoothly integrated in the programming environments generated by the ASF+SDF system.

3 Specification of an Interpreter

Our example language, CLaX, features the following language concepts: types, type coercion, overloaded operators, arrays, procedures with reference and value parameters, nested scopes, assignment statements, loop statements, conditional statements, and goto statements. In Figure 1, an example of a CLaX program is shown.

```
PROGRAM example;
DECLARE
  i: INTEGER; j: INTEGER;

  PROCEDURE incr(in: INTEGER; VAR out: INTEGER);
  BEGIN { incr }
    out := in + 1;
  END   { incr }

BEGIN { example }
  i := 3;
  incr(i, j)
END.  { example }
```

Fig. 1. Example of a CLaX program.

The interpreter for CLaX is based on the well-known concept of a stack of activation records (see e.g., [1]). This stack contains one record for every procedure that is being executed. Each record contains the code of that procedure, a 'pointer' to the current statement, and a set of (references to) values defined in the procedure. In our specification, two distinct stacks are used to model the stack of activation records, allowing us to separate control flow issues from operations on the data:

- The *code stack* consists of zero or more code records, where each code record is a pair containing the name of the procedure, and a list of statements that remains to be executed.
- The *data stack* consists of zero or more data records, with each data record containing (i) the name of the procedure, (ii) scope information, (iii) label continuations, and (iv) a list of zero or more identifier-value pairs.

As an example, we consider the execution of the CLaX program of Figure 1. When the assignment statement in the procedure body is executed, the state looks as follows:

```
< [ incr, out := in + 1 ] [ example, incr(i, j) ],

  [ incr, 1, in : 3  out: ref(j, ) ]
  [ example, , i : 3  j : 0  incr : ··· ] >
```

The first line shows the code stack, containing two records: one for procedure `incr`, and one for the main program. The first of these records, `[ incr, out := in + 1 ]` tells us that the current procedure is named `incr`, and that the list of statements that remains to be executed consists of the single statement `out := in + 1`. The second and third line show (parts of) the data stack. The first data record, `[ incr, 1, in : 3  out: ref(j, ) ]` contains the value `3` for `in`; moreover, the value for `out` is a reference to the value of `j` in the next record.

The CLaX interpreter is invoked by applying a function `eval-program` to the abstract syntax tree (AST) of a program. First, an initial state is computed; then, a recursive evaluation function `eval` is repeatedly applied to the state. Applications of `eval` can be regarded as execution steps of the interpreter. These steps are: (i) the execution of a statement, (ii) the return from a procedure, and (iii) program termination (i.e., extraction of the values of global variables from the final state). The interpreter computes the following list of variable-value pairs for the example program of Figure 1:

```
i : 3  j : 4
```

To give an example of the flavor of the specification, Figure 2 shows equations `[ev7]` and `[ev8]` which define the execution of an `IF-THEN-ELSE` statement. Depending on the result of evaluating the predicate (by way of an auxiliary function `eval-predicate`, not shown here), the `IF` statement is replaced by the statements in either the `THEN` or in the `ELSE` branch. The complete specification of the CLaX interpreter consists of approximately 200 equations. Basic arithmetic operations and I/O are performed in Lisp.

Algebraic specifications can be executed as term rewriting systems [15]. A term rewriting system (TRS) is obtained from an algebraic specification by orienting the equations from left to right; such an oriented equation is referred to as a *rewrite rule*. Term rewriting is a cyclic process; it consists of the transformation of an initial term (in our setting: a function `eval-program` applied to the AST of a CLaX program) by repeatedly matching subterms against left-hand sides of rewrite rules. If a match succeeds, a reducible expression (*redex*) is established, and the variables in the rewrite rule obtain a binding. The redex is replaced by the instantiation of the right-hand side of the rewrite-rule, and the term rewriting process proceeds by looking for a new match. A rewriting process terminates when no more redexes can be found, the term is then said to be in *normal form*.

In the case of conditional TRSs, conditions have to be evaluated after a match has been found. A conditional rewrite rule is only applicable if all its conditions

```
[ev7] eval-predicate(_Exp, _DStack) = TRUE
      ==========================================================
      eval( <[_Id, IF _Exp THEN _Stat*' ELSE _Stat*'' END; _Stat*]
            _CRec*, _DStack> ) =
      eval( <[_Id, _Stat*';_Stat*] _CRec*, _DStack> )

[ev8] eval-predicate(_Exp, _DStack) = FALSE
      ==========================================================
      eval( <[_Id, IF _Exp THEN _Stat*' ELSE _Stat*'' END; _Stat*]
            _CRec*, _DStack> ) =
      eval( <[_Id, _Stat*'';_Stat*] _CRec*, _DStack> )
```

Fig. 2. Equations defining the execution of an `IF-THEN-ELSE` statement.

succeed. The evaluation of a condition consists of the instantiation and rewriting of the condition sides, and the comparison of the resulting normal forms.

4 Definition of Events

We mentioned that an application of `eval` corresponds to an execution step of the interpreter. This can be restated as follows: an execution step takes place when a redex *matches* the pattern `eval( <[`*_Id*`, `*_Stat**`] `*_CRec**`, `*_DStack*`> )`. Here, the variables *_Id*, *_Stat**, *_CRec**, and *_DStack* match any identifier, any list of statements, any list of code records, and any data stack, respectively. Specific applications of `eval` can be recognized by specializations of this pattern. In particular, we propose that an application of `eval` corresponds to:

- *the execution of a statement* if the current code record contains *at least one* more statement which is to be executed. The corresponding pattern, `eval( <[`*_Id*`, `*_Stat*`; `*_Stat**`] `*_CRec**`, `*_DStack*`> )`, is obtained by replacing *_Stat** by the more specific list *_Stat*`; `*_Stat** which matches *one* or more statements.
- *the return from a procedure* if the current code record contains no more code to be executed, and there is *more than one* record on the code stack. This event corresponds exactly to a match with the pattern `eval( <[`*_Id*`, ] [`*_Id'*`, `*_Stat*`; `*_Stat**`] `*_CRec**`, `*_DStack*`> )`. Note that the variable *_Stat** in the general pattern is replaced by the empty list, and the variable *_CRec** by `[`*_Id'*`, `*_Stat*`; `*_Stat**`] `*_CRec**`.`
- *program termination* if the current code record contains no more code to be executed, and there is *exactly one* record on the code stack. The pattern which describes this event is `eval( <[`*_Id*`, ], `*_DStack*`> )`. This time we have replaced both *_Stat** and *_CRec** by empty lists.

Table 1 summarizes some events and corresponding patterns for CLaX.

Table 1. Events and corresponding patterns for CLaX.

Event	**Pattern**
execution of a statement	`eval( <[`*_Id*`, `*_Stat*`; `*_Stat∗*`] `*_CRec∗*`, `*_DStack*`> )`
return from a procedure	`eval( <[`*_Id*`, ] [`*_Id′*`, `*_Stat*`; `*_Stat∗*`] `*_CRec∗*`, `*_DStack*`> )`
evaluation of a predicate	`eval-predicate(`*_Expr*`, `*_DStack*`)`
processing of a declaration	`init-decls(`*_Decl*`; `*_Decl∗*`, `*_Stat∗*`, `*_Path*`)`
processing of a parameter	`init-params(`*_Actual*`, `*_Actual∗*`, `*_Formal*`; `*_Formal∗*`, `*_DStack*`)`

5 Determining Subjects

By matching patterns against redexes, one can determine that a particular program construct is executing. Next, we deal with the remaining question of finding *which* construct is being executed. To this end, we need the following ingredients:

1. A means to trace subterms of redexes back to subterms of the program's abstract syntax tree.
2. A mechanism to indicate subterms of patterns that 'correspond' to subjects.

We use *origin tracking* to deal with the first issue. Informally stated, origin tracking comprises of the following:

- Before the rewriting process is started, each symbol of the initial term is *annotated* with positional information.
- Before the rewriting process is started, a *propagation rule* is derived automatically for each equation of the specification.
- Whenever an equation is applied, positional information *propagated* from the redex to the newly created subterm according to the corresponding propagation rule.

For a formal definition of the origin function, its implementation, and a discussion of its properties, the reader is referred to [9].

For the second issue, we use the well-known notion of *paths* (occurrences). A path is a sequence of natural numbers corresponding to argument positions of function symbols. A path uniquely identifies a function symbol in a term/pattern. As an example, we consider the pattern `eval( <[`*_Id*`, `*_Stat*`; `*_Stat∗*`] `*_CRec∗*`, `*_DStack*`> )`, which corresponds to the execution of a statement. Path (1 1 1 2 1) indicates the subterm *_Stat*, which is matched against the statement that is currently executing. Figure 3 shows the pattern as a tree structure where edges are labeled with argument positions of function symbols. Path (1 1 1 2 1) corresponds to the traversal of this tree from the root to the subterm *_Stat*.

Table 2 summarizes the paths to the subjects for each of the patterns shown in Table 1.

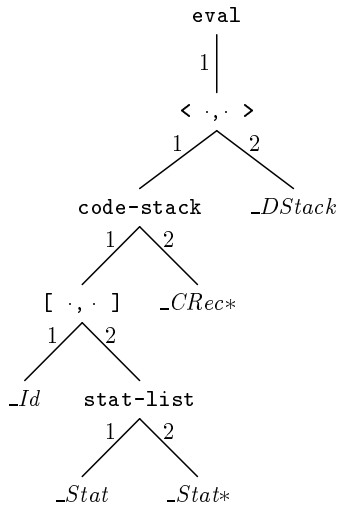

Fig. 3. Pattern `eval( <[`*_Id*`, _Stat; _Stat*`] _CRec*, _DStack*`> )` as a tree structure.

Table 2. Subjects of events. The first column contains paths indicating the subterms of the patterns shown in boxes in the second column. The third column lists the corresponding subjects.

Path(s)	Pattern and subterm(s) indicated b paths	Subject(s)
(1 1 1 2 1)	`eval( <[`*_Id*`,` **_Stat** `; _Stat*] _CRec*, _DStack*`>)`	the statement
(1 1 2 2 1)	`eval( <[`*_Id*`, ] [`*_Id′*`,` **_Stat** `; _Stat*] _CRec*,` *_DStack*`> )`	the procedure call
(1)	`eval-predicate(`**_Expr**`, `*_DStack*`)`	the predicate
(1 1)	`init-decls(`**_Decl**`;`*_Decl*`*, _Stat+, _Path)`	the declaration
(1 1), (2 1)	`init-params(` **_Actual**`, `*_Actual*`*, **_Formal**`; `*_Formal*`*, _DStack)`	the actual and the formal parameter

6 Example

As an example, we show some snapshots of the animator for CLaX that was generated by the ASF+SDF system. The patterns and paths of Table 2 are used to animate the execution of the program of Figure 1.

First, a match with pattern `init-decls(`*_Decl*`; _Decl*, _Stat*, _Path*`)` is found. The origin of the subterm at path (1 1), which is matched against variable *_Decl* is retrieved; the animation of the corresponding declaration is shown in

Figure 4. The next two animation steps (not shown) highlight the declaration of `j` and the (procedure) declaration of `incr`, respectively.

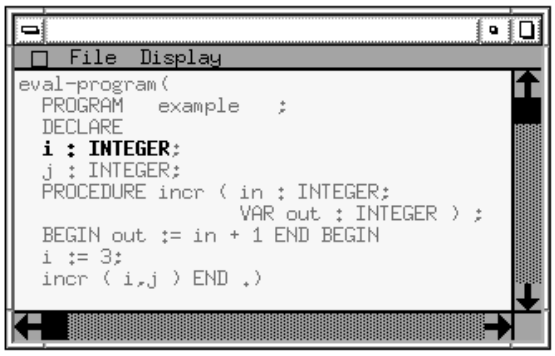

Fig. 4. Animation of the execution of a declaration.

Subsequently, two matches with the 'execute statement' pattern are found, resulting in the animation of the assignment statement of the main program (see Figure 5), followed by the animation of the call statement (not shown).

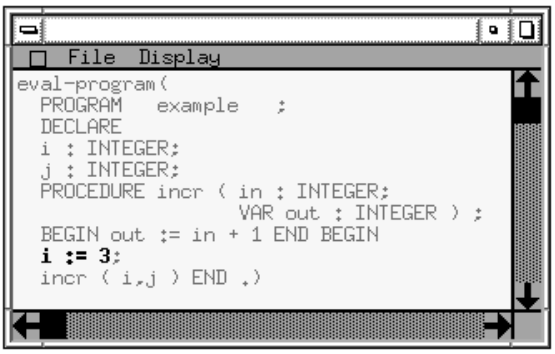

Fig. 5. Animation of the execution of a statement.

Then, the parameters of the procedure call are processed; the first of these two animation steps is shown in Figure 6.

The remaining steps consist of highlighting the statement in the procedure body, followed by the highlighting of the call statement in the main program again. The latter step is due to a match with the 'return from procedure' pattern.

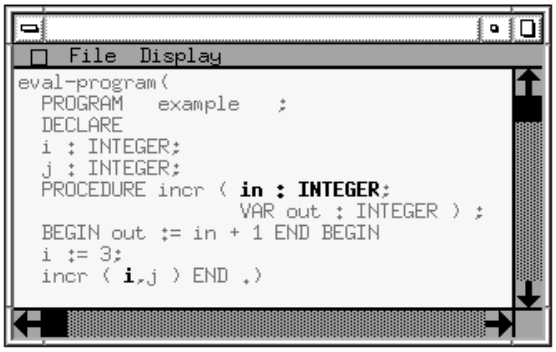

Fig. 6. Animation of formal and actual parameters.

7 Limitations

Experience with our example language CLaX has taught us that our method is in principle suitable for a wide range of programming languages. Nevertheless, there are a few limitations.

The fact that we generate animators from specifications for interpreters implies that we can only perform animations which 'correspond' to execution steps of the interpreter. Moreover, we can only detect events which can be defined as syntactic constraints on redexes. These limitations do not appear to be very restrictive, as is illustrated by the animator we have derived for CLaX.

The use of origin tracking for determining subjects works well for interpreter specifications with a compositional structure, where the execution of a language construct is either expressed in terms of the execution of its sub-constructs, or, recursively, in terms of the execution of the same construct in a modified environment. However, when the execution of a language construct is expressed in terms of the execution of *other* constructs, origin tracking fails to establish relations. As a result, it will not be possible to determine the subject, and animation steps will be missing.

We would like to emphasize that we do not consider this to be a major problem, because most interpreter specifications are – at least to a very large extent – written in a compositional manner. In the case of the CLaX interpreter (which was written before we studied animators), we have changed only 2 equations to obtain the desired animation behavior.

As an example of a problem case, we show an equation which expresses the execution of a `REPEAT` construct in terms of the execution of a `WHILE` construct:

```
[evX] eval( <[_Id, REPEAT _Stat* UNTIL _Exp END; _Stat*']
            _CRec*, _DStack> ) =
      eval( <[_Id, _Stat*; WHILE NOT(_Exp) DO _Stat* END; _Stat*']
            _CRec*, _DStack> )
```

Naturally, the problem can be remedied by re-defining the execution of a `REPEAT` statement in terms of itself and its sub-constructs. We are currently investigating a more appealing solution to this problem, which consists of an extension of the origin function. This would enable us to determine useful subjects in cases such as the one described above. Some ideas in this direction are discussed in [10, 8].

8 Generation of Source-level Debuggers

As an extension of the generation of animators, we are currently investigating the generation of source-level debuggers from specifications of interpreters. Basic debugger features such as single-stepping, breakpoints, state inspection, and providing backtrace information can be expressed in terms of our pattern-matching approach. Below, we discuss how each of these features can be defined for CLaX. For reasons of clarity, we will only pay attention to statement-level debugging features (e.g., breakpoints on statements, single-stepping at statement level). Obviously, each of these features can be defined for all appropriate language constructs. For example, we could define a breakpoint on the 'execution' of a control predicate.

Single Stepping
: There is little difference between single-stepping, and the animation steps we have presented earlier. A single CLaX statement is executed by continuing the term rewriting process until the next match with pattern `eval( <[`*_Id*`,` *_Stat*`; `*_Stat*`*]` *_CRec*`*,` *_DStack*`> )` is encountered.

Breakpoints
: A breakpoint on a CLaX statement can be defined as follows:
  1. The user indicates a statement subterm of the AST of the program (e.g., in a syntax-directed editor).
  2. The path p from the root of the AST to the designated statement is automatically determined by the debugger.
  3. The rewriting process is continued until a match with pattern `eval( <[`*_Id*`,` *_Stat*`; `*_Stat*`*]` *_CRec*`*,` *_DStack*`> )` is encountered *and* the origin of the subterm matched against variable *_Stat* contains path p.

State Inspection
: Using the function `eval-exp` of the CLaX specification, which computes the value of an expression, the current value of an arbitrary source-level expression can be computed as follows. Suppose that variable *_Exp* is bound to the expression we want to evaluate. Then, its value can be computed by *rewriting* the (instantiation of the) term `eval-exp(`*_Exp*`,` *_DStack*`)`, where the binding of variable *_DStack* is obtained during the last match with pattern `eval( <[`*_Id*`,` *_Stat*`; `*_Stat*`*]` *_CRec*`*,` *_DStack*`> )`. The rewriting process for the interpreter itself is suspended for the duration of the `eval-exp` rewriting process.

Backtrace Information

In Section 3, we outlined how an interpreter state contains one code record for each procedure call. Each code record contains a list of statements remaining to be executed in that procedure; the first element of this list is the current statement. All information needed for a backtrace is available, since we can do the following for *each* code record:

- Retrieve the origin of the current statement, and highlight the corresponding subterm of the AST.
- The values of parameters and local variables can be obtained by considering the corresponding record on the data stack.

More advanced features such as watchpoints [19], conditional breakpoints, and breakpoints on a reference to a designated variable can be expressed in similar ways. Moreover, since we operate in an interactive setting, *changing* the values of variables or even the program itself is conceivable. Both of these features can be implemented by physical modification of the interpreter state during the rewriting process.

From the above discussion, we conclude that obtaining the functionality of a source-level debugger is a feasible task. However, before we are able to generate source-level debuggers, more work remains to be done on the development of a formalism to *specify* both the behavior, and the user-interface of a debugger. For example, if one wants to set a breakpoint on a particular language construct, this could be done by selecting that construct in the ASF+SDF system's structure editor. Then, the system could infer the associated pattern from the type of the selected language construct and the debugger specification.

9 Generation of Algorithmic Debuggers

An interesting extension of our work on the generation of conventional source-level debuggers would be the generation of algorithmic debuggers [18, 11]. An algorithmic debugger partially automates the task of localizing a bug by comparing the *intended* program behavior with the *actual* program behavior. The intended behavior is obtained by asking the user whether or not a program unit (e.g., a procedure) behaves correctly. Using the answers given by the user, the location of the bug can be determined at the unit level.

In order to generate algorithmic debuggers from specification of interpreters, the following issues have to be dealt with:

1. The original program has to be transformed into an equivalent, side-effect free program. A possible approach is described in [17]. An extension of the origin function can be used to relate subterms of the ASTs of the original program, and the transformed one.
2. Algorithmic debugging is a post-mortem technique, based on the analysis of the execution tree of a program. Our approach to generate animators and debuggers, on the other hand, can be regarded as interactive debugging. However, patterns could be used to intercept the events when information

(for the execution tree) needs to be stored. This information should include the *origins* of the procedure calls, so as to be able to animate the algorithmic debugging process.

3. A separate algorithmic debugger has to be constructed which interprets the execution tree information, and uses the origins stored in (2) and the relations between the original and the transformed program stored in (1) to animate the algorithmic debugging process.

10 Conclusions and Future Work

We have presented a framework for incorporating animation features in generated programming environments where animators are generated from specifications of interpreters. We have considered simple animators which highlight the language constructs that are currently being executed. Origin tracking and a pattern-matching mechanism are used to define animators. The successful generation of an animator for CLaX shows the feasibility of our approach.

Section 7 describes some limitations, which can be summarized as follows. First, we can only detect events which correspond to syntactic constraints on redexes. Second, the use of origin tracking restricts us to specifications which have a compositional structure. As it turns out, these limitations do not cause much problems.

We claim that our approach for generating animators is suitable for a wide range of imperative programming languages, including realistic languages such as Pascal and C. We will investigate if animators can be generated for languages with parallel and object-oriented features in a similar way. We conjecture that these features will not cause fundamental problems.

A possible criticism is that patterns for defining events may become quite complicated. One should bear in mind, however, that these patterns are similar to the equations of the interpreter specification. We claim, therefore, that the definition of animation features is an easy task for the specification writer.

In Section 8, we have outlined how our approach can be extended to the generation of source-level debuggers, by indicating how various debugger features can be defined for our example language. In Section 9, we have described prerequisites for a further extension to the generation of algorithmic debuggers.

What remains to be developed is a formalism to *specify* animation and debugging features. Such an animator/debugger specification would define both the functionality and the user-interface of the generated tools.

Acknowledgements

I am grateful to Paul Klint, Peter Fritzson, T.B. Dinesh, and the AADEBUG'93 referees for their comments on drafts of this paper.

References

1. Aho, A.V. , Sethi, R., Ullman, J.D.: *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. Bahlke, R., Moritz, B., Snelting, G.: A generator for language-specific debugging systems. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 92–101, 1987. Appeared as SIGPLAN Notices 22(7).
3. Bahlke R., Snelting, G.: The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.
4. Bergstra, J.A., Heering, J., Klint, P., Eds.: *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.
5. Berry, D.: *Generating Program Animators from Programming Language Semantics.* PhD thesis, University of Edinburgh, 1991.
6. Bertot, Y.: Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 327–337, 1991. Appeared as *SIGPLAN Notices* 26(6).
7. Borras, P., Clément, D., Despeyroux, Th., Incerpi, J., Lang, B., Pascual, V.: CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).
8. Deursen, A. van: Origin tracking in primitive recursive schemes. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1993. To appear.
9. Deursen, A. van, Klint, P., Tip, F.: Origin tracking. Report CS-R9230, Centrum voor Wiskunde en Informatica (CWI), 1992. To appear in *Journal of Symbolic Computation.*
10. Dinesh, T.B., Tip, F.: Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), 1992.
11. Fritzson, P., Gyimothy, T., Kamkar, M., Shahmehri, N.: Generalized algorithmic debugging and testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 317–326, 1991. Appeared as *SIGPLAN Notices* 26(6).
12. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
13. Kahn, G.: Natural semantics. In Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M., Eds.: *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
14. Klint, P.: A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
15. Klop, J.W.: Term rewriting systems. In Abramsky, S., Gabbay, D., Maibaum, T., Eds.: *Handbook of Logic in Computer Science, Vol II.* Oxford University Press, 1991. Also CWI report CS-R9073.
16. Müller, H., Winckler, J., Grzybek, S., Otte, M., Stoll, B., Equoy, F., Higilin, N.: The program animation system PASTIS. Bericht 20, Universität Freibürg, Institut für Informatik, 1990.
17. Shahmehri, N.: *Generalized Algorithmic Debugging.* PhD thesis, Linköping University, 1991.

18. Shapiro E.Y.: *Algorithmic Program Debugging.* MIT Press, 1982.
19. Stallman, R.M., Pesch, R.H.: *Using GDB, A guide to the GNU Source-Level Debugger.* Free Software Foundation/Cygnus Support, 1991. Version 4.0.

This article was processed using the LaTeX macro package with LLNCS style