Automatic Generation of Language-based Tools using the LISA system

Pedro Rangel Henriques[a,1] Maria João Varanda Pereira[b,2,*] Marjan Mernik[c,3] Mitja Lenič[c] Jeff Gray[d] Hui Wu[d]

[a] *University of Minho, Department of Informatics, Portugal*

[b] *Polytechnic Institute of Bragança, Portugal*

[c] *University of Maribor, Faculty of Electrical Engineering and Computer Science, Slovenia*

[d] *University of Alabama at Birmingham, Department of Computer and Information Sciences, USA*

Abstract

Many tools have been constructed using different formal methods to process various parts of a language specification (e.g., scanner generators, parser generators and compiler generators). The automatic generation of a complete compiler was the primary goal of such systems, but researchers recognized the possibility that many other language-based tools could be generated from formal language specifications. Such tools can be generated automatically whenever they can be described by a generic fixed part that traverses the appropriate data structures generated by a specific variable part, which can be systematically derivable from the language specifications. This paper identifies generic and specific parts for various language-based tools. Several language-based tools are presented in the paper, which are automatically generated using an attribute grammar-based compiler generator called LISA. The generated tools that are described in the paper include editors, inspectors, debuggers and visualizers/animators. Because of their complexity of construction, special emphasis is given to visualizers/animators, and the unique contribution of our approach toward generating such tools.

Key words: attribute grammars, compiler generation, language-based tools, visualizers/animators

1 Introduction

The advantages of formal specification of programming language semantics are well known. First, the meaning of a program is precisely and unambiguously defined; second, it offers a unique possibility for automatic generation of compilers or interpreters. Both of these factors contribute to the improvement of programming language design and development. The programming languages that have been designed with formal methods have improved syntax and semantics, less exceptions and are therefore easier to learn. Moreover, from formal language definitions many other language-based tools can be automatically generated, such as: pretty printers, syntax-directed editors, type checkers, data flow analyzers, partial evaluators, debuggers, profilers, test case generators, visualizers, animators, and documentation generators; for a more complete list see [1]. In most of these cases the core language definitions have to be augmented with tool-specific information (e.g., mapping information in debuggers). In other cases, a fragment of formal language definitions (e.g., regular definitions) is enough for automatic tool generation. It is also possible to extract implicit information from the formal language definition (e.g., dependencies among attributes in semantic functions) in order to automatically generate a tool. The are many benefits of automatically generated language-based tools. Building language-based tools from scratch is time consuming and error prone, which makes maintenance very costly. This is a serious problem in building language-based tools for domain-specific languages (DSLs). In the case of DSLs, a compiler/interpreter is usually developed without support for other language-based tools (e.g. debuggers), which are indispensable for programmers. The lack of appropriate tools might even cause newly developed DSLs to become obsolete.

Although previous efforts have explored the concept of automatic generation of language-based tools [2] [3] [4] [1] [5], this paper contributes a more general approach that identifies generic (fixed) and specific (variable) parts from which language-based tools can be generated automatically from language specifications. In many cases, the language specification must be extended,

* Corresponding author.
Email addresses: `prh@di.uminho.pt` (Pedro Rangel Henriques), `mjoao@ipb.pt` (Maria João Varanda Pereira), `marjan.mernik@uni-mb.si` (Marjan Mernik), `mitja.lenic@uni-mb.si` (Mitja Lenič), `gray@cis.uab.edu` (Jeff Gray), `wuh@cis.uab.edu` (Hui Wu).

[1] The paper is an extension of already published paper at Workshop on Language Description, Tools and Applications, ENTCS 65, No. 3, 2003

[2] The work of M. João is partially supported by the Portuguese program PRODEP, acção 5.2 da medida 5 – doutoramentos

[3] The project was supported by Slovenian and Portugal governments under the contract SLO-P-11/01-04

or appropriate information extracted, in order to be able to automatically generate a language-based tool. The paper discusses several tools where the language definition does not need to be extended, such as editors to help in writing sentences of the language and various inspectors (e.g., automata visualizers, syntax tree visualizers, and semantic evaluator animators) that are helpful for a better understanding of the language analysis process. Such example tools have all been incorporated in the compiler generator system called LISA [6]. The paper also presents several language tools that require extensions to a language definition in order to implement a new tool (e.g., debuggers, algorithm animators and program visualizers).

The main goal of the paper is to show how language-based tools can be automatically generated from an extended language definition in a systematic manner by identifying generic and specific parts. The approach is presented in detail for visualizers/animators. Program visualizers/animators are useful tools for deeper and clearer understanding of algorithms, and are valuable for both programmers and students. Algorithm animators and program visualizers are strongly language and algorithm-oriented, and usually are not developed in a systematic or automatic way. This paper introduces the architecture and implementation of the Alma system, which represents an approach to the automatic generation of animators from extended language definitions. The system has a specific front-end for each language and a generic back-end, and uses a decorated abstract syntax tree (DAST) as the intermediate representation. In the implementation of Alma the language development system LISA is used in two different applications. LISA generates the front-end for each new language, and some parts of it (Java classes) are reused to build the back-end.

The standard definitions about languages and context-free grammars that make automatic implementation of programming languages and language-based tools possible can be found in classical textbooks, such as [7]. To specify the semantics of programming languages, context-free grammars need to be extended. Attribute grammars [8] are a generalization of context-free grammars in which each symbol has an associated set of attributes that carry semantic information, and with each production a set of semantic rules with attribute computation is associated. Attribute grammars have proved to be very useful in specifying the semantics of programming languages, in automatic construction of compilers/interpreters, and in specifying and generating interactive programming environments [9]. The approach presented in this paper is strongly tied to the power provided by attribute grammars.

The organization of the paper is as follows. Related work is described in section 2. Language-based tools that are automatically generated by the LISA system are described in section 3. The design and implementation of the Alma system are described in section 4. A summary and concluding remarks are presented in section 5.

2 Related Work

The development of the first compilers in the late fifties without adequate tools was a very complicated and time consuming task. For instance, the implementation of the compiler for the programming language FORTRAN took about 18 human years [10]. Later on, formal methods, such as operational semantics, attribute grammars, denotational semantics, action semantics, algebraic semantics, and abstract state machines, were developed. They made the implementation of programming languages easier and finally contributed to the automatic generation of compilers/interpreters.

Many tools have been built in the past years, based on different formal methods to assist in processing different parts of language specification, such as: scanner generators, parser generators and compiler generators. The automatic generation of a complete compiler was the primary goal of such systems. However, researchers soon recognized the possibility that many other language-based tools could be generated from formal language specifications. Therefore, many tools not only automatically generate a compiler but also complete language-based environments. Such automatically generated language-based environments include editors, type checkers, debuggers, and various analyzers.

For example, FNC-2 [11] is an attribute grammar system that generates a scanner/parser, an incremental attribute evaluator, a pretty printer, and a dependency graph visualizer. The CENTAUR system [3] is a generic interactive environment which produces a language specific environment from formal specifications written in Natural Semantics, a kind of operational semantics. The generated environment includes a scanner/parser, a pretty printer, a syntax-directed editor, a type checker, an interpreter and other graphic tools. The SmartTools system [5], a successor of the CENTAUR system, is a development environment generator that provides a compiler/interpreter, a structured editor and other XML related tools. The ASF+SDF environment [12] generates a scanner/parser, a pretty printer, a syntax-directed editor, a type checker, an interpreter, and a debugger from algebraic specifications. In the Gem-Mex system [4], the formal language is specified with abstract state machines. The generated environment includes a scanner/parser, a type checker, an interpreter, and a debugger. Very similar to the Synthesizer Generator (`SGen`) [2], the LRC system [13] generates, from high-order attribute grammar specifications, an incremental scanner/parser and attribute evaluators, syntax-directed editor, multiple views of the abstract semantic tree (unparsing windows), and windows-based interfaces. From the above description of various well known compiler/interpreter generators can be noticed that editors, pretty printers, and type checkers are almost standard tools in such automatically generated environments. However, in those papers particular language-based tools are described from the user's point of view and not how these tools are actually generated. No systematic treatment of language-based tool generation has appeared in the literature. In this paper, a systematic approach is described with specific emphasis on the automatic generation of visualizers/animators.

To our knowledge, the only visualizer/animator to be automatically generated from formal specifications is `Jitan` [14] [15], a visualization environment for concurrent, object-oriented programming for Java. The `CENTAUR` system was used to implement `Jitan`, where the syntax was specified by the `METAL` formalism and the semantics defined by the `TYPOL` formalism. The authors of `Jitan` reported that only about ten semantic rules of language specifications needed to be equipped with simple extensions. This was possible because their two visualizer engines need to know the existence and status of available objects. In this case, the generic part of the system is enormous and the specific part is tailored to objects and threads. Even though their approach is specific approach to automatically generating language-based tools, the approach is quite different from that described in this paper with respect to generic and specific parts. The Alma system is much more general in this respect and desires to have the specific part as big as possible. For example, an Alma user has all of the power to define the appearance of the visualization (e.g., colors and lines) through rules.

3 Tools from language definitions generated by the LISA system

LISA is a compiler-compiler, or a system that generates automatically a compiler/interpreter from attribute grammar based language specifications. The syntax and semantics of LISA specifications and its special features (i.e., "templates" and "multiple attribute grammar inheritance") are described in more detail in [16]. The use of LISA in generating compilers for real programming languages (e.g., PLM, AspectCOOL and COOL, SODL) is reported in [17], [18], [19]. LISA is unique to other attribute-grammar based compiler generators because it employs the "attribute grammar=class" paradigm [9] to enable incremental language development to a greater extent than other approaches. LISA has been used in many projects in combination with newly developed technologies and frameworks, such as conversion of parts of LISA specifications to XML schema and XML schema evolution (e.g., metamodel inference [20]).

To illustrate LISA style, the specification of a simple Robot language is given in Fig. 1. A robot can move in four directions from the initial position (0, 0). After moving, it is stopped in an unknown location, which the user wants to compute. Often, it is desirable to extend languages like the Robot language with new features. For example, it may be desired to know when the robot will reach the final position. Another example of adding a new language feature is the possibility that the robot can move at a different speed. In that case new syntactic constructs have to be added to the language. The new language (RobotSpeed) is specified (Fig. 2) as an extension to the Robot language using multiple attribute grammar inheritance. From these descriptions LISA automatically generates an interpreter for the RobotSpeed Language.

Additionaly, LISA is capable of generating other language-based tools. In the following subsections four families of such tools are briefly described: *editors* to help the final users in the creation and maintenance of the sentences of the specified language; *inspectors* that are useful to understand the behavior, or to debug the generated language processor itself; *debuggers*, which are indispensable in the debugging process; and *visualizers/animators*, similar to inspectors, which are useful in understanding the meaning of the source program that is being processed.

It is important to notice that automatic generation is possible whenever a tool can be built from a generic (fixed) part and a specific (variable) part. An additional requirement is that the specific part, which is language dependent, has to be systematically derivable from the language specifications. That part has a well-defined internal representation that can be traversed by the algorithms of the generic part. For example, a lexical analyzer uses an algorithm that interprets an action table [21]. This algorithm is generic and the same for different languages. However, the action table represents the specific part, and is changed whenever a language specification is modified. Table 1 summarizes some of the language-based tools generated by the LISA system. It is not the aim of this paper to describe all of the algorithms (many of them are described in [21]). The algorithms for program visualization and animation are described in detail in section 4.

However, to show the differences in generic and specific parts and the differences in exploiting language definitions, other language-based tools are briefly introduced.

3.1 Editors

Two different LISA generated language oriented editors (i.e., editors that are sensitive to the language lexicon/syntax) are briefly described in this section.

```
language Robot {
 lexicon {
   Commands left | right | up | down
   ReservedWord  begin | end
   ignore [\0x0D\0x0A\ ]  // skip whitespaces
 }
 attributes int *.inx; int *.iny;
            int *.outx; int *.outy;
 rule start {
  START ::= begin COMMANDS end compute {
    START.outx = COMMANDS.outx;
    START.outy = COMMANDS.outy;
    // robot position in the beginning
    COMMANDS.inx = 0;
    COMMANDS.iny = 0;};
  }
 rule moves {
  COMMANDS ::= COMMAND COMMANDS compute {
    COMMANDS.outx = COMMANDS[1].outx; // propagation of position
    COMMANDS.outy = COMMANDS[1].outy; // to sub-commands
    COMMAND.inx = COMMANDS.inx;
    COMMAND.iny = COMMANDS.iny;
    COMMANDS[1].inx = COMMAND.outx;
    COMMANDS[1].iny = COMMAND.outy;
     }
  | epsilon compute {
    COMMANDS.outx = COMMANDS.inx;
    COMMANDS.outy = COMMANDS.iny; };
  }
 rule move {
  // each COMMAND changes one coordinate
  COMMAND ::= left compute {
    COMMAND.outx = COMMAND.inx-1;
    COMMAND.outy = COMMAND.iny; };
  COMMAND ::= right compute {
    COMMAND.outx = COMMAND.inx+1;
    COMMAND.outy = COMMAND.iny; };
  // COMMAND ::= up, COMMAND ::= down are omitted
 }
}
```

Fig. 1. Robot Language

3.1.1 Language Knowledgeable Editors

LISA generates a language knowledgeable editor, which is a compromise between text editors and syntax-directed editors, from formal language specifications. In this case, a language definition does not need to be extended because

```
language RobotSpeed extends Robot {
  lexicon {
    extends Commands speed
    Number [0-9]+
  }
  attributes int *.inspeed, *.outspeed;
  rule extends start {
   compute {
     // initial position is inherited
     START.time = COMMANDS.time;
     COMMANDS.inspeed = 1; // beginning speed
     START.outspeed = COMMANDS.outspeed; }
  }
  rule extends moves {
   COMMANDS ::= COMMAND COMMANDS compute {
     // total time is sum of times spent in sub-commands
     COMMANDS[0].time = COMMAND.time + COMMANDS[1].time;
     COMMAND.inspeed = COMMANDS[0].inspeed;   // speed propagation
     COMMANDS[1].inspeed = COMMAND.outspeed;  // to sub-commands
     COMMANDS[0].outspeed = COMMANDS[1].outspeed; }
   | epsilon compute {
     COMMANDS.time = 0;
     COMMANDS.outspeed = COMMANDS.inspeed; };
  }
  rule extends move {
  // these commands do not change speed
   COMMAND ::= left compute {
     COMMAND.time = 1.0/COMMAND.inspeed;
     COMMAND.outspeed = COMMAND.inspeed; };
   COMMAND ::= right compute {
     COMMAND.time = 1.0/COMMAND.inspeed;
     COMMAND.outspeed = COMMAND.inspeed; };
   // COMMAND ::= up, COMMAND ::= down are omitted
  }
  rule speed {
   COMMAND ::= speed #Number compute {
     COMMAND.time = 0;  // no time is spent for this command
     COMMAND.outspeed = Integer.valueOf(#Number.value()).intValue();
     // this command does not change the position
     COMMAND.outp = COMMAND.inp; };
  }
}
```

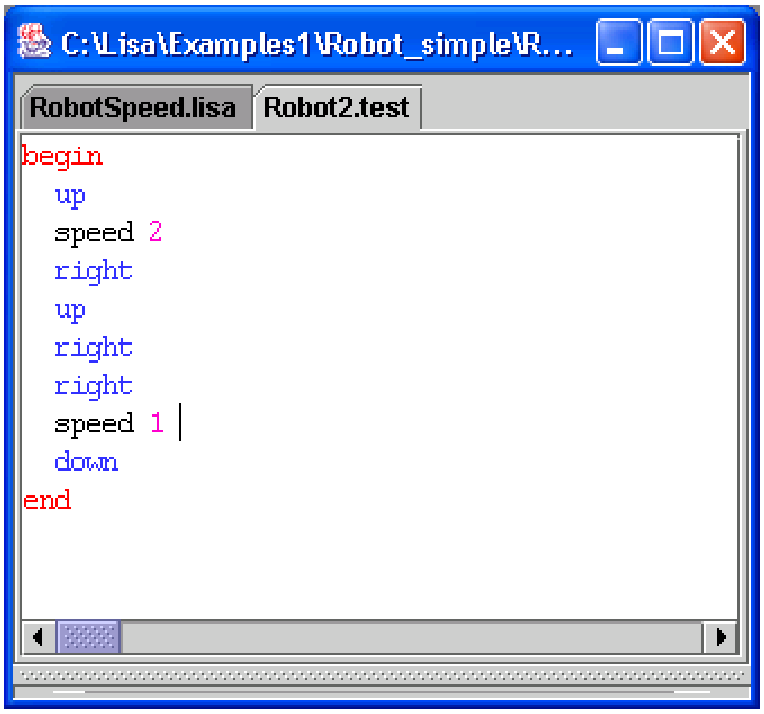
Fig. 2. RobotSpeed Language

Generated tool	**Formal specifications**	**Generic part**	**Specific part**
Lexer	regular definitions	algorithm which interprets action table	action table: $State \times \Sigma \rightarrow State$
Parser (LR)	BNF	algorithm which interprets action table and goto table	action table: $State \times T \rightarrow Action$ goto table: $State \times (T \cup N) \rightarrow State$
Evaluator	Attribute Grammars (AG)	tree walk algorithm	semantic functions
Language knowledgeable editor	regular definitions (extracted from AG)	matching algorithm	same as lexer
Structure editor	BNF (extracted from AG)	incremental parsing algorithm	same as parser
Finite state automata visualization	regular definitions (extracted from AG)	finite state automata layout algorithm	same as lexer
Syntax tree visualization	BNF (extracted from AG)	syntax tree layout algorithm	syntax tree
Dependency graph visualization	extracted from AG	DG layout algorithm	dependency graph
Semantic evaluator animation	extracted from AG	semantic tree layout algorithm	decorated syntax tree & semantic functions
Debugger	additional formal specifications	mapping algorithm	mapping component
Program visualization and animation (ALMA)	additional formal specifications	visualization, rewriting and animation algorithm	visual and rewrite rules and decorated abstract tree (DAST)

Table 1
Generic and specific parts of LISA generated language-based tools

the matching algorithm (i.e., the generic part) only needs information about regular definitions in the language.

The LISA generated language knowledgeable editor is aware of the regular definitions of the language lexicon (see table 1). Therefore, it can color the different parts of a program (comments, operators, and reserved words) to en-



```
begin
  up
  speed 2
  right
  up
  right
  right
  speed 1
  down
end
```

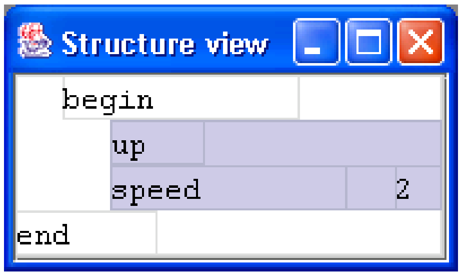
Fig. 3. Language knowledgeable editor

hance understandability and readability of programs. Improved understandability is important for programs written in DSLs where end-users are typically not programmers but application engineers. Developing such editors for every different DSL is time consuming and costly. The benefits of automatically generated editors are obvious.

In Figure 3 the reserved words, commands and integers in a RobotSpeed program are recognized by distinguishing each feature with a different color (e.g., in figure 3 the reserved words are in red).

3.1.2 Syntax-directed Editors

Syntax-directed editors help users to write syntactically correct programs before they are actually compiled, exhibiting the language structure by inserting directly the keywords at the correct places (the user only has to fulfill the variable parts of their text). Syntax-directed editors are aware of the language syntax of edited programs and can be automatically generated from a syntax language definition. LISA currently generates a structured editor from formal language specifications. A Structure Editor is a kind of syntax-directed editor, where the syntax structure of written programs are explicitly seen while editing the program (see Figures 4 and 5 where the selected text is a set of $COMMANDS$ in the RobotSpeed language). The language definition is not extended because the incremental parsing algorithm (i.e., the generic part) only needs the information about syntax definitions in the language.



```
begin
    up
    speed        2
end
```

Fig. 4. Structure editor

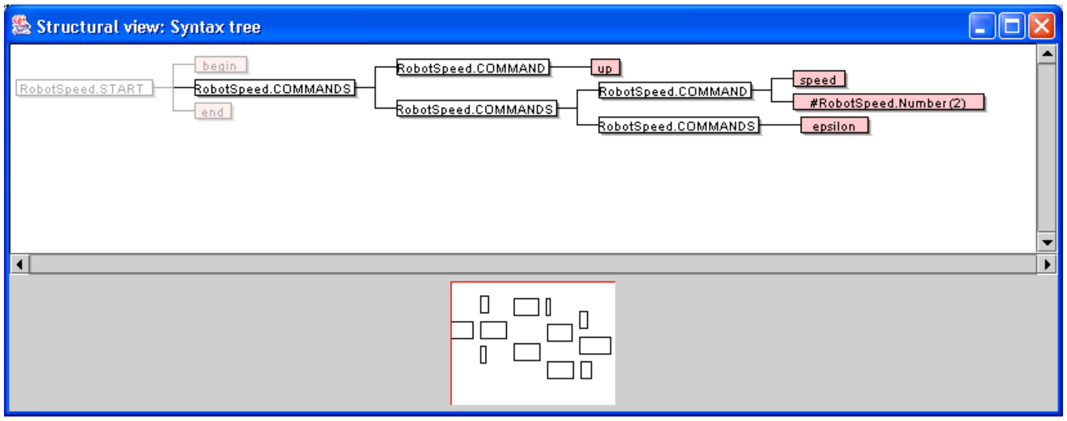


Fig. 5. Syntax tree view

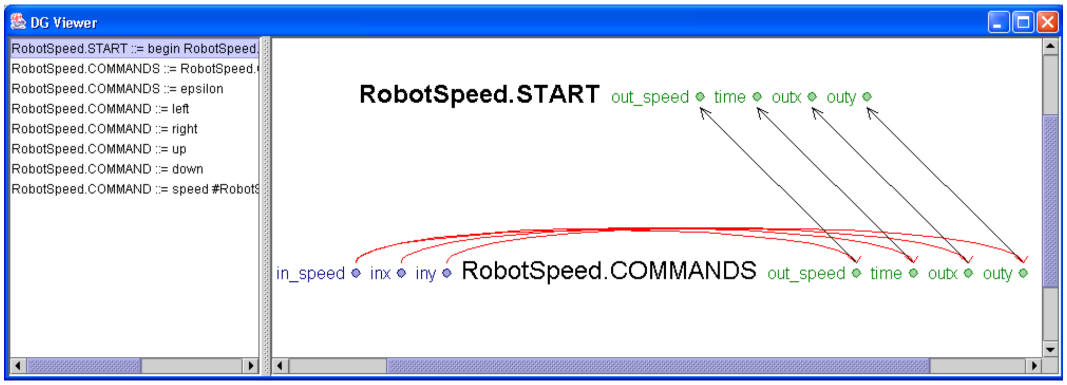
3.2 Inspectors for Language Processors

Inspectors are useful in better understanding how an automatically generated language compiler/interpreter works. LISA generates the following inspectors: finite state automaton visualizer (useful for better understanding how the lexical analyzer work), syntax tree visualizer (useful for better understanding how the syntax analyzer work), dependency graph visualizer and semantic tree visualizer. The last two inspectors are briefly introduced in this subsection.

3.2.1 Dependency Graph Visualization

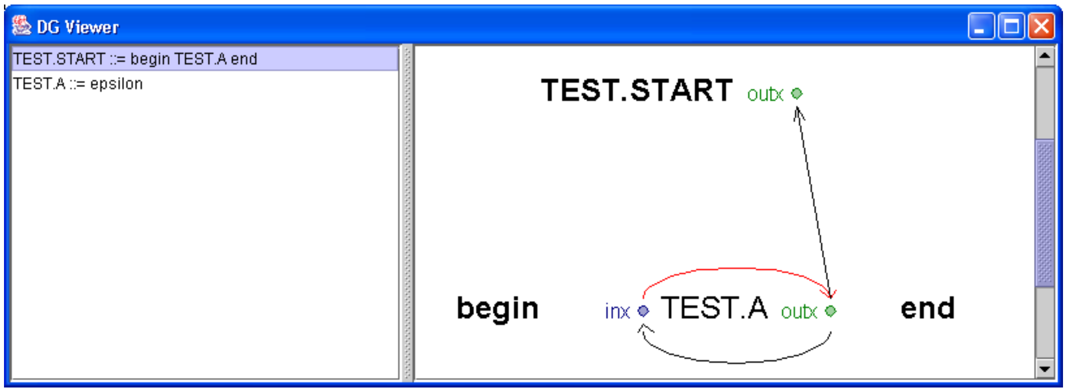
As attribute grammars are specified on the declarative level, the order of attribute evaluation is determined by the compiler construction tool. That sequence is also important for the language designer to understand the actual evaluation order. LISA generates this inspector from information extracted from language specifications, which is where the augmented dependency graph (i.e., the specific part) is computed. The augmented dependency graph is used in an algorithm (i.e., the generic part) for dependency graph layout.

Figure 6 presents an augmented dependency graph that is drawn by the LISA generated tool for the 1st RobotSpeed production. Direct dependen-


DG Viewer
RobotSpeed.START out_speed time outx outy
in_speed inx iny RobotSpeed.COMMANDS out_speed time outx outy

RobotSpeed.START ::= begin RobotSpeed.
RobotSpeed.COMMANDS ::= RobotSpeed.
RobotSpeed.COMMANDS ::= epsilon
RobotSpeed.COMMAND ::= left
RobotSpeed.COMMAND ::= right
RobotSpeed.COMMAND ::= up
RobotSpeed.COMMAND ::= down
RobotSpeed.COMMAND ::= speed #RobotS

Fig. 6. Dependency graph view



TEST.START ::= begin TEST.A end
TEST.A ::= epsilon

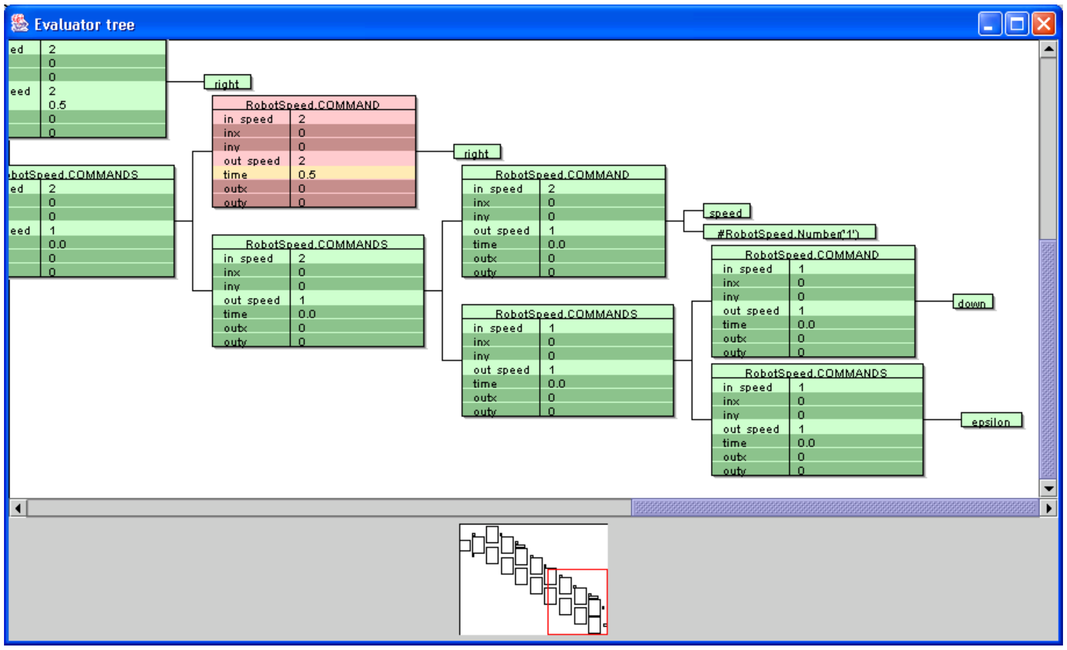
Fig. 7. Circular dependency graph

cies (e.g., `COMMANDS.time` → `START.time`) and indirect dependencies (e.g., `COMMANDS.inx` → `COMMANDS.outx`) are shown in different colors. With the aid of this inspector it is easier to discover why a particular attribute grammar is not absolutely non-circular. Circular attribute grammars are not supported by LISA and circularity is detected in Fig. 7.

3.2.2 Semantic Evaluator Animation

In attribute grammars a set of attributes carrying semantic information is associated with each nonterminal. For example, attributes `time` and `outx` are associated with nonterminal `COMMANDS` in the RobotSpeed language specifications. In the evaluation process the value of these attributes has to be computed. The semantic analysis is better understood by animating the visits to the nodes of the semantic tree, and the evaluation of attributes in these nodes. LISA generates this inspector from semantic functions associated to syntax rules. The semantic tree layout algorithm (i.e., the generic part) uses a decorated syntax tree and semantic functions, which constitutes the specific parts.

Figure 8 shows a snap-shot of the animation process. The animation of the



ed	2
	0
	0
eed	2
	0.5
	0
	0

botSpeed.COMMANDS	
ed	2
	0
	0
eed	1
	0.0
	0
	0

RobotSpeed.COMMAND	
in speed	2
inx	0
iny	0
out speed	2
time	0.5
outx	0
outy	0

RobotSpeed.COMMANDS	
in speed	2
inx	0
iny	0
out speed	1
time	0.0
outx	0
outy	0

RobotSpeed.COMMAND	
in speed	2
inx	0
iny	0
out speed	1
time	0.0
outx	0
outy	0

RobotSpeed.COMMANDS	
in speed	1
inx	0
iny	0
out speed	1
time	0.0
outx	0
outy	0

RobotSpeed.COMMAND	
in speed	1
inx	0
iny	0
out speed	1
time	0.0
outx	0
outy	0

RobotSpeed.COMMANDS	
in speed	1
inx	0
iny	0
out speed	1
time	0.0
outx	0
outy	0

Fig. 8. Semantic Evaluation view

evaluation process is also very helpful in debugging language specifications. Users can also control the execution by single-stepping and setting the break-points.

Notice the way scalability is addressed: as the tree grows it is impossible to show all of the picture in the main window. Therefore, a subtree is displayed in the main window and a general picture is shown below (see Fig. 8).

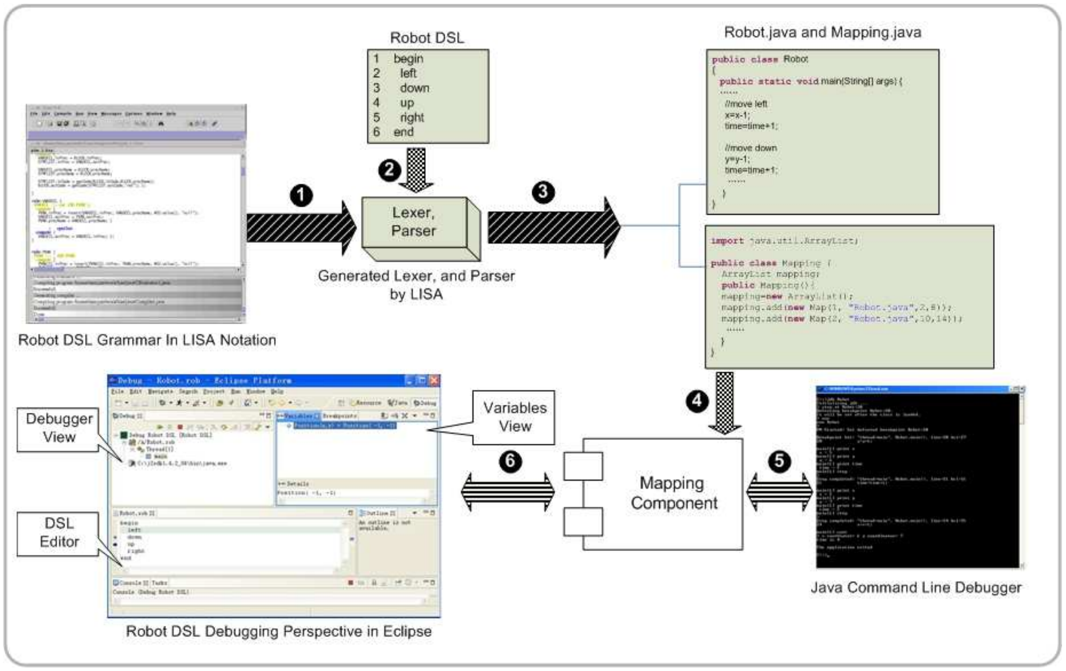
3.3 Debugging Support for DSLs Defined in LISA

Debuggers provide software engineers with an essential tool toward discovering the location of program errors. However, development can be difficult when it comes to the issue of debugging a program written in a DSL, which often requires both programming language development expertise and domain knowledge. This is due to the fact that a DSL is often translated into some other general purpose language (GPL) and then compiled using the tools available for the GPL. Even if the domain expert has knowledge about the underlying GPL, one line of DSL code may be translated into dozens of lines of GPL code, which requires knowledge of the code generator in order to understand the correspondence between the DSL and GPL. As an example, consider the challenges in debugging parsers generated by tools such as Yacc. In such a case, the benefits provided by the domain idioms are lost because the domain expert is forced to debug their intention at the GPL level, not at the higher abstraction level provided by the DSL. This sub-section describes the ability to generate debuggers for DSLs defined in LISA. From a DSL grammar, LISA can generate the mapping transformations needed by the DSL Debugger Framework (DDF) [22], which provides debugging support for DSLs in Eclipse. This allows an end-user or domain expert to debug their DSL program at the proper level of abstraction.

3.3.1 DSL Debugger Generation Processes Overview

An illustrative overview of the DSL debugger generation process is shown in Figure 9. The front-end of the process begins with the generation of a lexer and parser for the DSL. LISA automatically generates the lexer and parser from a DSL grammar definition, such as the Robot language grammar definition (shown in Figure 10 of the next section). In addition to the lexer and parser, a mapping generator is needed to link the DSL code to the generated GPL code. The mapping generator is specified as additional semantic actions in the DSL grammar definition. The lexer, parser, and mapping generator form the building blocks for the front-end of the DDF. The back-end of the DDF consists of the stand-alone command line GPL debugger and the Eclipse debugger perspective [23]. The Eclipse debugger perspective provides the graphical interface that is commonly expected in integrated development environments (IDE). Note that the choice of the GPL debugger depends on the kind of GPL code generated from the DSL. In the Robot language example, the generated GPL code is Java, which influenced the choice to use the Java command line debugger (jdb) [24]. Although this specific example represents a DSL that is translated to Java, the Eclipse debugger platform is independent of the GPL. Thus, LISA and the DDF can be used with any generated GPL provided a debugger exists for the GPL.

The semantic actions associated with the debugger use syntax-directed translation and additional semantic functions in the grammar specification to generate the mapping information. In Figure 9, with the mapping generator embedded inside the grammar, the lexer and parser generated by LISA (step 1) takes the Robot DSL as input (step 2). LISA not only translates the Robot DSL into the corresponding Robot.java, but also generates the Mapping.java file (step 3). The mapping file represents a data structure that records all of the mapping information about which line of the Robot DSL code is mapped to the corresponding segment of Robot.java code. It indicates the location of the Robot.java code segment. Interestingly, the mapping information crosscuts the grammar in such a way that an aspect emerges within the grammar definition [25]. The mapping component interacts and bridges the differences between the Eclipse debugger platform and the jdb (step 4). There are two round-trip mapping processes involved (step 5 and step 6) between the Robot DSL debugging perspective in Eclipse and jdb. A user issues debugging commands from the Eclipse that are interpreted into a series of jdb commands


Robot DSL
Robot.java and Mapping.java
Lexer, Parser
Generated Lexer, and Parser by LISA
Robot DSL Grammar In LISA Notation
Debugger View
Variables View
DSL Editor
Mapping Component
Java Command Line Debugger
Robot DSL Debugging Perspective in Eclipse
1
2
3
4
5
6

```
1  begin
2    left
3    down
4    up
5    right
6  end
```

```
public class Robot
{
 public static void main(String[] args) {
 ......
  //move left
  x=x-1;
  time=time+1;

  //move down
  y=y-1;
  time=time+1;
   ......
  }
}
```

```
import java.util.ArrayList;

public class Mapping {
  ArrayList mapping;
  public Mapping(){
  mapping=new ArrayList();
  mapping.add(new Map(1, "Robot.java",2,8));
  mapping.add(new Map(2, "Robot.java",10,14));
   ......
  }
}
```

Fig. 9. Debugger Generation Overview

against the Robot.java code. Based on the pre-defined debugging mapping knowledge, the mapping component determines the sequence of debugging commands that need to be issued to the jdb at the GPL level. The center piece of the DDF is the mapping component that bridges the generated code from the front-end to the execution engines of the back-end (e.g., the GPL command-line debugger and the Eclipse debugger perspective). The mapping component acts as an interpreter that knows two different languages (i.e., the DSL source and the generated GPL code). The DDF translates the user's debugging intentions from the Eclipse debugger perspective to the GPL debugger; it also translates the debugging outputs from the GPL debugger back to the user through the Eclipse debugger perspective at the DSL level.

The generic part in this case is a mapping algorithm that uses syntax-directed translation and additional semantic functions in the grammar specification to generate the mapping component, which constitute the specific part.

3.3.2 Robot DSL Debugger

Figure 10 represents a fragment of the Robot DSL grammar in LISA. Line 11 indicates the start of the grammar production to process a "right" command, with lines 12 through 18 providing the semantic actions needed to execute the intention of "right" in Java. Lines 12, 14, 16, and 17 represent the debug mapping information that contains the line number of the "right" command (attribute *dslline* in line 12) in the Robot DSL. The mapping contains the following information:

(1) the DSL line number (line 17),
(2) the translated Java file name (line 18),
(3) the line number of the first line of the corresponding code segment in Robot.java (attribute *gplbegline* on line 18),
(4) the line number of the last line of the corresponding code segment in Robot.java (atribute *gplendline* on line 18).

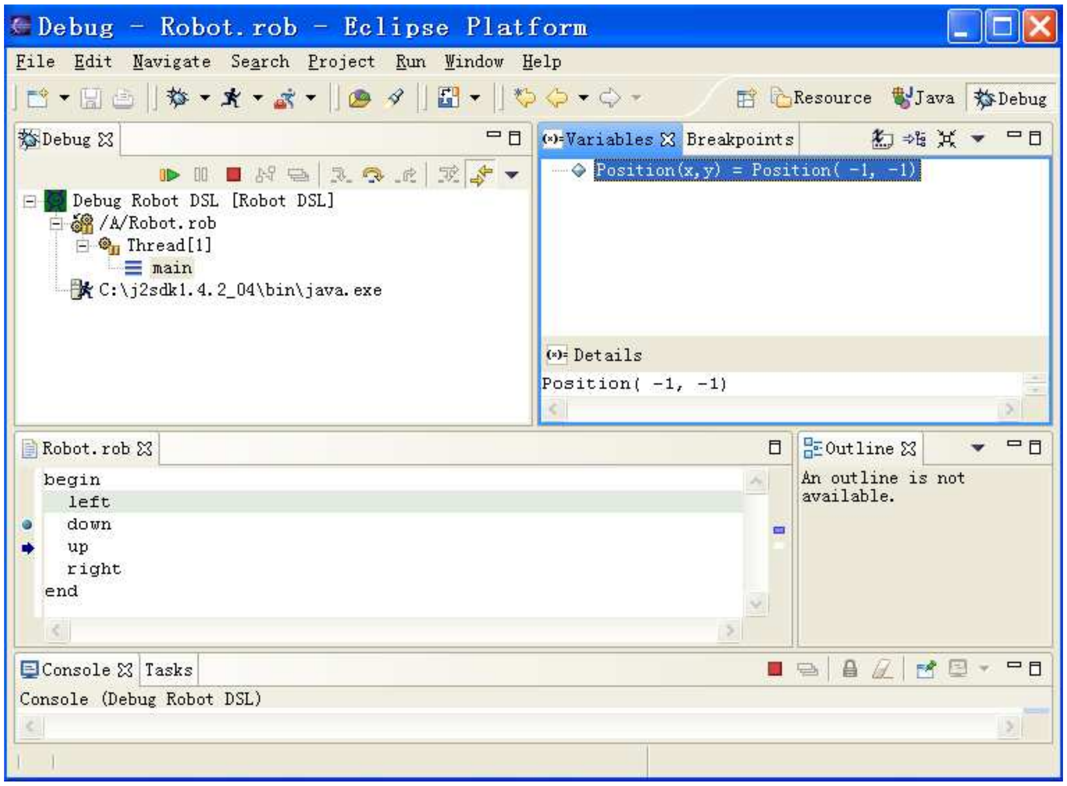
```
                                    ⋮
10 rule move {
11   COMMAND ::= right compute {
12     COMMAND.dslline = COMMAND.indslline + 1;
13     COMMAND.code1 =” x=x+1;// move right”;
14     COMMANDD.gplbegline = COMMAND.ingplbegline;
15     COMMAND.code = COMMAND.code1 + ” time=time+1;”;
16     COMMAND.gplendline = COMMAND.gplbegline + 2;
17     COMMAND.mapcode = ”mapping.add(newMap(” + COMMAND.dslline +
18         ”,\”Robot.java\”,”+ COMMAND.gplbegline + ”,” + COMMAND.gplendline + ”));”; };
19   COMMAND ::= left compute {
20     COMMAND.dslline = COMMAND.indslline + 1;
21     COMMAND.code1 = ” x=x-1;// move left”;
22     COMMANDD.gplbegline = COMMAND.ingplbegline;
23     COMMAND.code = COMMAND.code1 + ” time=time+1;”;
24     COMMAND.gplendline = COMMAND.gplbegline + 2;
25     COMMAND.mapcode = ”mapping.add(newMap(” + COMMAND.dslline +
26         ”,\”Robot.java\”,”+ COMMAND.gplbegline + ”,” + COMMAND.gplendline + ”));”; };

                                    ⋮
```

Fig. 10. Robot DSL Grammar in LISA Notation

The jdb responds to the debugger commands sent from the mapping component. The results from the jdb are sent back to a reverse-mapping component. Because the messages from the jdb are command line outputs, which know nothing of the Robot language and the Eclipse debug platform, it is necessary to remap the results back into the Eclipse debugging perspective. The Robot DSL's variable Position is displayed in the variables view (see upper right corner of Figure 11). The mapping component translates the messages back to the Robot DSL through the wrapper interface. The domain expert only interacts directly with the DSL editor and debugger view at the Robot language level (see left side of Figure 11).

This section demonstrated LISA's ability to generate programming language tools inside of the LISA programming environment. Additionally, integration with external Integrated Development Environments (IDEs), such as Eclipse, is also possible due to the power of language-based generation.


Debug - Robot.rob - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
Java
Debug
Variables
Breakpoints
Details
Outline
Console
Tasks

```
Debug Robot DSL [Robot DSL]
  /A/Robot.rob
    Thread[1]
      main
  C:\j2sdk1.4.2_04\bin\java.exe
```

```
Position(x,y) = Position( -1, -1)
```

```
Position( -1, -1)
```

```
begin
  left
  down
  up
  right
end
```

```
An outline is not
available.
```

```
Console (Debug Robot DSL)
```

Fig. 11. Robot DSL Debugger Perspective in Eclipse

3.4 Program Visualization and Animation

Another instance of tools that can be derived from formal language specifications are *program visualizers/animators*. The purpose of such a family of tools is to help the programmer to inspect the data and control flow of a source program—static view of the algorithms realized by the program (visualization) —and to understand its behavior—dynamic view of the algorithms' execution (animation). In this section the Alma system is briefly introduced. The *front-end*'s that are used by Alma can be constructed using any compiler generator tool, but in this discussion it will be used as a LISA addon.

For automatic generation of a program visualizer/animator, a language specification needs to be extended with additional information that defines how the input sentence is converted into the animator's internal representation (DAST), as shown in Figure 14. Below is an example of such an extension for the Robot language, where additional steps are added to each command.

```
language AlmaRobot extends Robot, AlmaBase {
    rule start {
      START ::= begin COMMANDS end compute {
        START.dast = new Alma.CRoot(COMMANDS.tree); };
    }
    rule moves {
      COMMANDS ::= COMMAND COMMANDS compute {
        COMMANDS[0].tree= new Alma.CStmtsNode(COMMAND.tree,
                                        COMMANDS[1].tree); }
      | epsilon compute {
        COMMANDS[0].tree = NULL; };
    }
    rule move {
      COMMAND ::= left #Number compute {
        COMMAND.tree = new Alma.CLstNode(new Alma.CConstNode("left"),
                              new Alma.CConstNode(#Number)); };
      | COMMAND ::= right #Number compute {
        COMMAND.tree = new Alma.CLstNode(new Alma.CConstNode("right"),
                              new Alma.CConstNode(#Number)); };
           ...
    }
}
```

The extension shown above illustrates the use of *multiple attribute grammar inheritance*, which is a standard LISA feature [16]. It is used to specify the attribute evaluation related to the DAST construction. From this specification, a parser and a translator are generated that converts each input text into an abstract representation used by the animator, common to all different source languages. That processor, which is the animator's *front-end*, is the language dependent component of the tool. In this case, its generic part is more complex (described in detail in section 4) than in the cases studied in previous subsections 3.1 and 3.2: it is not just a standard algorithm (we use three language independent algorithms), but it requires also two standard data structures (a visual rule base, and a rewriting rule base). Notice that the DAST is language paradigm dependent. Each node of the DAST is related to concepts defined in the source program. The visualization of these concepts will assist in understanding the program.

Consider the following source program in the Robot language:

```
DOWN 3
RIGHT 7
UP 2
LEFT 4
```

The animation algorithms can generate a visualization like the one that can be seen in Figure 12. The final layout can be modified by the Alma designer. Drawing procedures called by the visualizing rules can be changed easily.

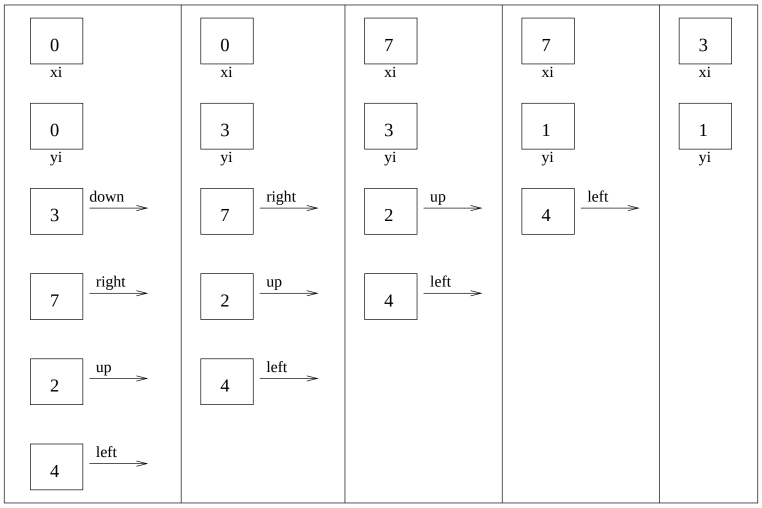


Fig. 12. Robot Operational Animation

Another possible visualization is shown in figure 13.

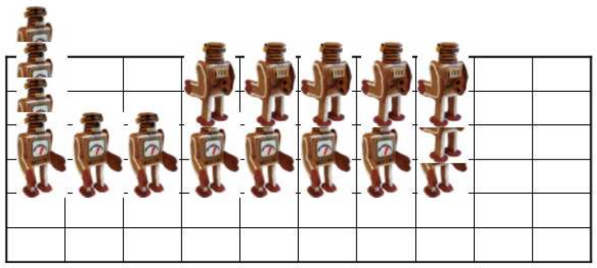


Fig. 13. Robot Animation

The figure above is more abstract and shows the effect produced by the program in the robot. For this kind of visualization, variable values are not shown like in the first visualization. Instead, the robot coordinates are used to evaluate each new position of the robot. The program variables are mapped to robot attributes in an interesting manner. In this case, it is clear that each x and y of the program will be the coordinates used to draw the robot. This is not as simple in other cases.

The approach allows the visualization of data structures and can handle procedures and objects. In these cases, the animation can be achieved with adequate visualization rules and drawing procedures. The implementation of the system, which is discussed in the next section, has a front-end specific for each language and a generic back-end. The implementation uses a decorated abstract syntax tree (DAST) for the intermediate representation between the front and back ends.

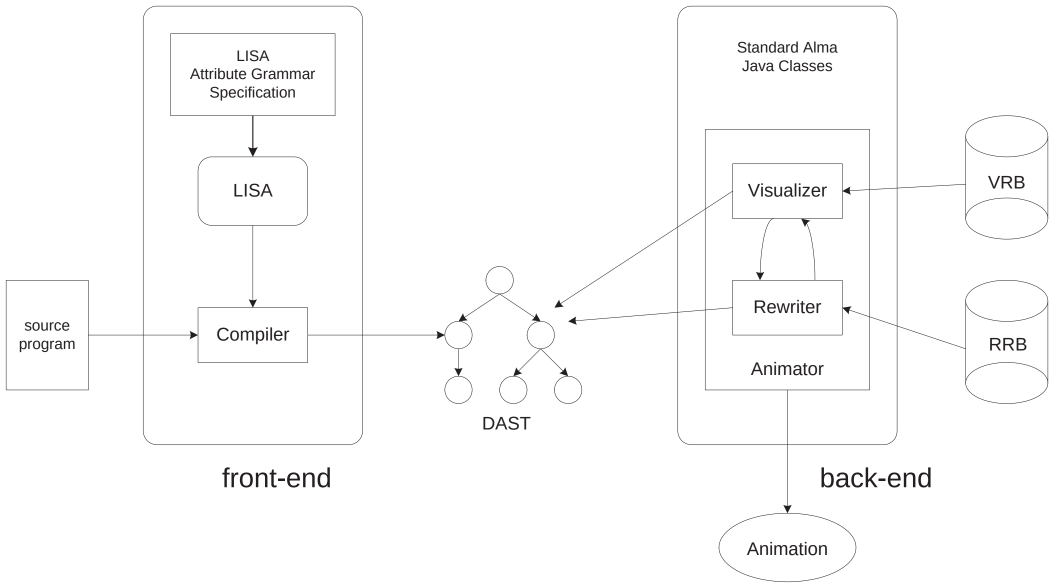


Fig. 14. Architecture of the Alma system

4 **Alma Implementation**

The Alma system was designed to become a new generic tool for program visualization and animation based on the internal representation of the input program in order to avoid any kind of annotation of the source code (with visual types or statements). The system was also designed to be able to handle different programming languages.

Alma was conceived as a tool to shield an end-user (a programming beginner, a student, a teacher) from the concerns of formal specification of the programming language. Visualization and Rewriting rules, which form the core of Alma, depend only on generic abstract concepts. The mapping of the concrete programming language constructs into abstract concepts is entirely embedded in the *front-end*, which is specific for each language and built just once by a compiler specialist. The *front-end* performs the translation task, from the concrete program to Alma's internal representation, and hides all details from the end-user.

4.1 Alma Architecture

To comply with the requirements above, we conceived the architecture shown in Figure 14. Alma also uses DAST as an internal representation for the meaning of the program that is to be visualized. All of the source language dependencies are isolated in the *front-end*, and the generic animation engine is in the *back-end*. The DAST is specified by an abstract grammar independent of the concrete source language. In some sense, it can be said that the abstract grammar models a virtual machine.

The DAST is intended to represent the program state in each moment, and not to reflect directly the source language syntax. In this way we rewrite the DAST to describe different program states, simulating its execution; notice that we deal with a semantic transformation process, not only a syntactic rewrite.

A *Tree Walk Visualizer* traverses the tree, creates a visual representation of nodes, and glues figures in order to get the program image at a specific moment. The DAST is rewritten (to obtain the next internal state) and redrawn to generate a set of images that will constitute the animation of the program. Different visualizations can be generated from the same DAST depending on the visualization rules.

4.1.1 Visualization in Alma

The visualization is achieved by applying visualization rules (VR) to DAST subtrees. The rules define a mapping between trees and figures and constitutes the specific part. When the partial figures corresponding to the nodes of a given tree are assembled together, a visual representation is obtained for the respective program.

Visualizing Rules

The VRB (Visualizing Rule Base) is a mapping that associates with each attributed tree, defined by a grammar rule (or production), a set of pairs

$$\text{VRB: DAST} \mapsto \text{set (cond} \times \text{dp)}$$

where each pair has a matching condition (`cond`) and a procedure (`dp`), which defines the tree visual representation. Each `cond` is a predicate, over attribute values associated with tree nodes, which constrains the use of the drawing procedure (`dp`); i.e., `cond` restricts the applicability of the visualizing rule.
The written form of each visualizing rule is as follows:

```
vis_rule(ProdId)= <tree-pattern>,
                  (condition),
                  {drawing procedure}


<tree-pattern> = <root, child_1, ..., child_n>
```

In this template, `condition` is a boolean expression (by default, evaluates to true) and `drawing procedure` is a sequence of one or more calls to elementary drawing procedures.
A visualization rule can be applied to all the trees that are instances of the production `ProdId`. A tree-pattern is specified using variables to represent each node. Each node has the attributes `value`, `name` and `type` that will be used on the rule specification, either to formulate the condition, or to pass to the drawing procedures as parameters.

Although each VRB associates to a production a set of pairs, for the sake of simplicity its written form only describes one pair. It is possible to have more than one rule for the same production. To illustrate the idea suppose that in Alma's abstract grammar a *relational operation*, `rel_oper`, is defined by the 13th production:

```
p13:   rel_oper : exp exp
```

where `exp` is defined as:

```
p14:   exp : CONST
p15:   exp : VAR
p16:   exp : oper
```

A visual representation for that relational operation is shown in figure 15.

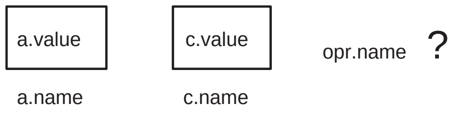


Fig. 15. Visualization of a relational operation

The visualization rules to specify that mapping are written below.

```
vis_rule(p13) =
       <opr,a,c>,
       ((a.type=exp) AND (c.type=exp)),
       {drawRect(a.name,a.value),drawRect(c.name,c.value),
        put(opr.name),put('?')}
```

This visualization rule is applied to a three-node tree, which consists of an operator and two operands. Each operand can be a `CONST`, `VAR` or `oper` (another operation). For each operand, a rectangle is constructed with its value inside and its name as a label of the rectangle. A visual representation of the operation is drawn and the image is finished with a `?` character to identify a relational expression.

Visualization rules are similar to pretty-printers or unparsing facilities in many compiler generators (e.g., PPML in Centaur [26]). Although unparsing produces text (in many case the text before parsing), visualization rules in Alma are also able to produce figures.

Visualization Algorithm

The visualization algorithm (i.e., the generic part) traverses the tree and applies the visualization rules to the sub-trees rooted in each node according to a bottom-up approach (post-fix traversal). Using the production identifier of the root node, it obtains the set of possible representations. A drawing procedure is selected depending on the first constraint condition that is true. The algorithm is presented below.

```
visualize(tree){
         If not(empty(tree))
            then forall t in children(tree) do visualize(t);
         rules <- VRB[prodId(tree)]; found <- false;
         While (not(empty(rules)) and not(found))
           do r <- choice(rules);
              rules <- rules - r;
              found <- match(tree,r)
         If (found) then draw(tree,r); }
```

A program animation is not the same as code visualization because it depends on the granularity of the visualization rules. The DAST is an abstract representation of the source code, which assists in applying a visualization rule to each node of the tree (getting a more detailed visualization, usually an operational view like a debugger). It is also possible to apply a visualization rule to a set of nodes, or even to the root. In this last case, the animation is more abstract from a debugger output. We assume that Alma suffers the same problems as other systems that use visual languages. Scalability is indeed a problem and care must be taken such that the drawings used in a visualization help program understanding.

4.1.2 Animation in Alma

Each rewriting rule (RR) specifies a state transition in the process of program execution and constitutes the specific part. The result of applying the rule is a new DAST obtained by a semantic (may be also a syntactic) change of a sub-tree. This systematic rewriting of the original DAST is interleaved with a sequence of visualizations producing an animation. A main function synchronizes the rewriting process with the visualization in a parameterized way, allowing for different views of the same source program.

Rewriting Rules

The RRB (Rewriting Rule Base) is a mapping that associates a set of tuples with each tree

$$\text{RRB: DAST} \mapsto \text{set}(\text{cond} \times \text{newtree} \times \text{atribsEval})$$

where each tuple has a matching condition (`cond`), a tree (`newtree`, which defines syntactic transformations), and an attribute evaluation procedure (`atribsEval`, which defines the changes in the attribute values).
The written form of each rewriting rule is as follows:

```
rule(ProdId)= <tree-pattern>,
              (condition),
                  <NewProdId: newtree>,
                  {attribute evaluation}


      <tree-pattern> = <root, child_1, ..., child_n>
      <newtree> = <root, child_1, ..., child_n>
```

In this template, `condition` is a boolean expression (by default, evaluates to true) and `attribute evaluation` is a set of statements that defines the new attribute values (by default, evaluates to skip).
A rewriting rule can be applied to all the trees that are instances of the production `ProdId`. A `tree-pattern` associates variables to nodes in order to be used in the other fields of the rule specification (i.e., the matching condition, the new tree and the attribute evaluation). When a variable appears in both the `tree-pattern` (the left side of the RR) and the `newtree` (the right side of the RR), it means that all the information contained in that node, including its attributes, will not be modified (i.e., the node is kept in the transformation unchanged).
Although each RRB associates to a production a set of tuples, its written form, introduced above, only describes one tuple. It is possible to have more than one rule for the same production. For instance, consider the following productions, belonging to Alma's abstract grammar, to define a conditional statement:

```
p8:     IF     : cond actions actions
p9:            | cond actions
```

The DAST will be modified using the following rules:

```
rule(p8) = <if,op,a,b>,
           (op.value=true),
           <p9:if,op,a>,
           { }


rule(p8) = <if,op,a,b>,
           (op.value=false),
           <p9:if,op,b>,
           { }
```

Rewriting Algorithm

The rewriting algorithm (i.e., the generic part) is also a tree-walker that traverses the tree until a rewriting rule can be applied, or no more rules match the tree nodes (in that case, the transformation process stops). For each node, the algorithm determines the set of possible RR using its production identifier (`ProdId`) and evaluates the contextual condition associated with those rules. The DAST will be modified removing the node that matches the left side of the selected RR and replacing it by the new tree defined by the right side of that RR. This transformation can be just a

semantic modification (only attribute values change), but it can also be a syntactic modification (some nodes disappear or are replaced).
The rewriting algorithm follows:

```
rewrite(tree){
  If not(empty(tree)) then  rules <- RRB[prodId(tree)];
  found <- false;
  While (not(empty(rules)) and not(found))
    do r <- choice(rules);
      rules <- rules - r;
      found <- match(tree,r)
  If (found)
    then tree <- change(tree,r)
    else a <- nextchild(tree)
  While (not(empty(a)) and not(rewritten(a)))
    do a <- nextchild(tree)
  If not(empty(a)) then tree <- rebuild(tree,a,rewrite(a))
  return(tree) }
```

Animation Algorithm

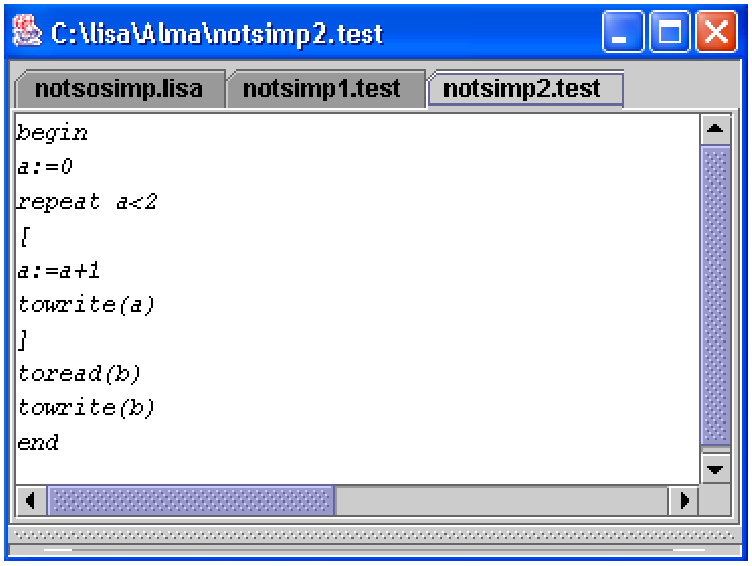
The main function defines the animation process, calling the visualization and the rewriting processes repeatedly. The simplest way consists in redrawing the tree after each rewriting, but the sequence of images obtained can be very long and may not be the most interesting. The granularity of the tree redrawing is controlled by a function, called `shownow()`, which after each tree's syntactic-semantic transformation decides if it is necessary to visualize again. The decision is made taking into account the internal state of the animator (that reflects the state of program execution) and the value of user-defined parameters.
The animation algorithm, which is the core of Alma's generic *back-end*, is as follows:

```
animate(tree){
        visualize(tree);
        Do    rewrite(tree);
              If shownow() then visualize(tree);
        until (tree==rewrite(tree)) }
```

When no more rules can be applied, the output and input of the rewrite function are the same.

4.2 Alma animation example

In this section, an Alma animation example is presented on a toy imperative language that consists of assignment, conditional, repetitive and I/O statements.



```
begin
a:=0
repeat a<2
[
a:=a+1
towrite(a)
]
toread(b)
towrite(b)
end
```

Fig. 16. Example source program

4.2.1 An example

The example presented in figure 16 has an assignment, a repetitive statement, a reading and a writing statement. Figures 17 and 18 show visualizations belonging to an Alma animation. The first one represents the initial state of the program and the second one shows the first iteration of the cycle. Notice that symbol `--->` represents an assignment or an operation (if an arithmetic symbol is under the arrow); the symbol `#` represents a conditional statement; the symbol `@` represents a repetitive statement; the symbol `=~=>>` represents a write and `<<=~=` a read statement.

We have adopted the original LISA approach to cope with the size of the tree to be drawn (as noticed in subsection 3.2.2). The approach displays the picture condensed in a small window below the main window with the circle part magnified.

The generality of the system can be a handicap to achieve output effects. The system would be more useful if it allows the addition of new rules to support new concepts or generate different outputs. Alma has a generic part (visualization/animation algorithms; tree, nodes, identifier table and rules structure; and rule base interpretation) and a specific part (visualization and rewriting rules, nodes). We conclude that the generic part gives generality and the specific part makes it possible to obtain more adequate visualizations.

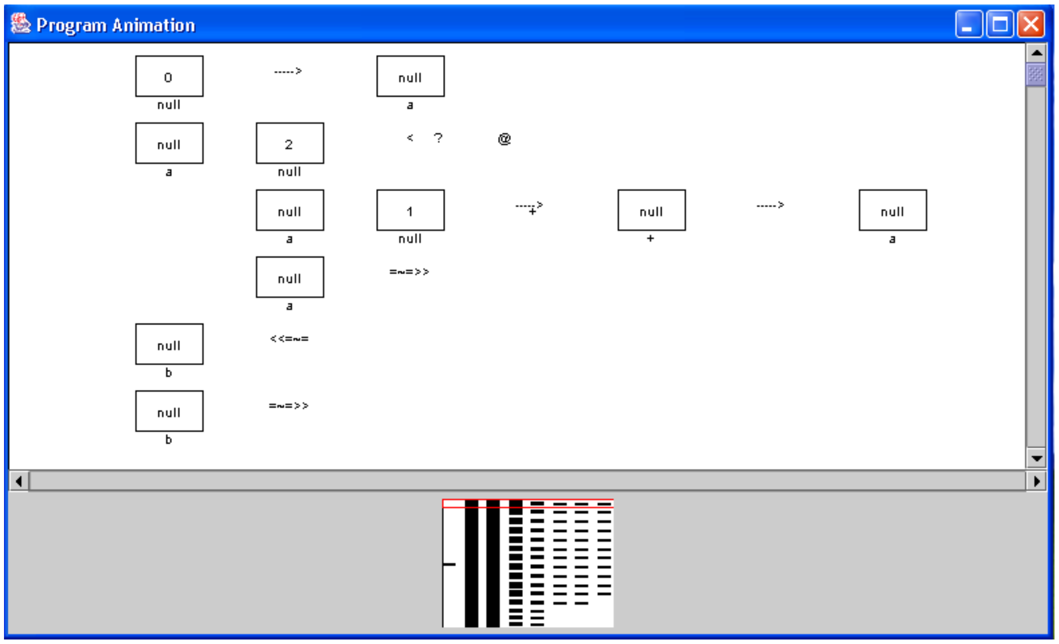


Fig. 17. Initial state of the program

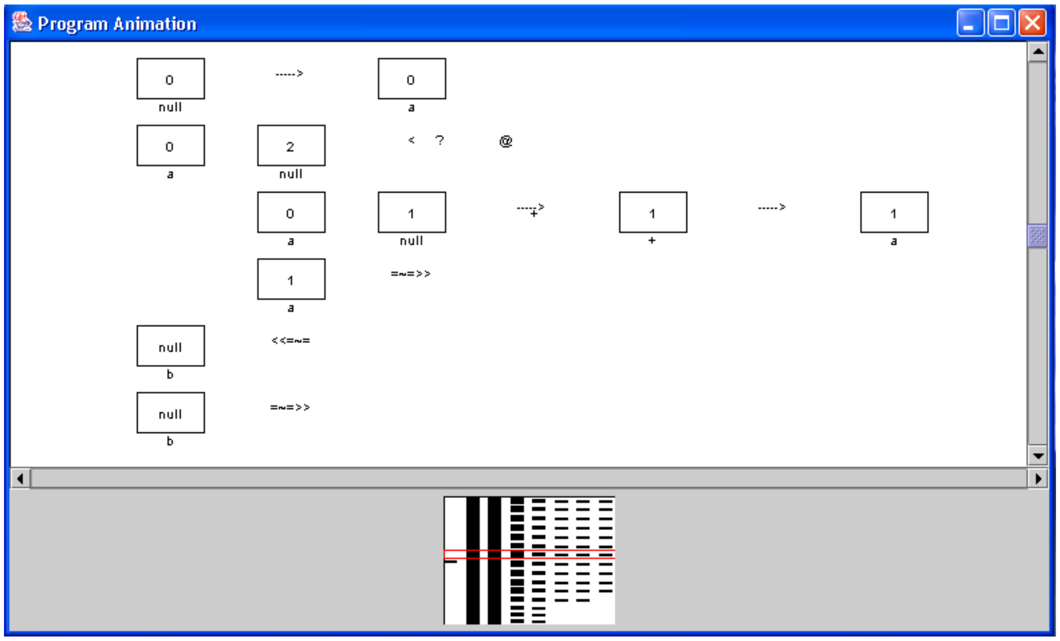


Fig. 18. The first iteration of the cycle

4.3 Other Alma features

Alma can also cope with different languages, different levels of animation detail, different types of visualizations and different types of paradigms.

Different Languages
If we want to apply the system to a different source language, we only have to construct a new *front-end* that defines the concrete syntax of the new language and maps its main concepts to Alma nodes. This *front-end* can be generated using LISA.

Different level of animation detail
It is also possible to modify the sampling frequency (number of state transformations before a visualization), or choose the set of nodes we want to visualize, in order to get a different level of animation detail. An animation can have more or less visualizations depending on the desired detail level. The most detailed animation implies the visualization of the tree after each rewriting. The synchronization between these processes depends on a function called *shownow*. This function counts the rewritings and returns 0 or 1 depending on the desired frequency.
The visualization is obtained by traversing a DAST that has the associated drawings. If we decide to show only some nodes we will get less detail in the visual representation. There are nodes that are more important than others and their visualization can explain all the functionality of the program. It is also possible to access an interface to choose the desired nodes and watch the results.
It is important to distinguish the animation detail level from the visualization detail level. For animation detail, the drawings do not have to change (it is concerned with process synchronization and the number of visualizations). In visualization detail, it is necessary to redefine the visualization rules in order to get different results.

Different types of visualizations
Alma has two bases of rules that can be improved with new semantics or new drawings. It may be desired to get different visualizations for the same language used before. Alternatively, it may be desired to animate a very different language, which would require the definition of new visualizations. There are several possibilities to change visualizations: varying the level of visualization detail using a different mapping between nodes and drawings; choosing different drawings; or both, in order to get a different abstraction level.
The generated visualizations are based on rules that map nodes to drawings. If we want to change the drawings in order to get a different visualization, we can modify rules or specify new ones. The same concepts can be represented with different drawings.
If we want to change the visualization detail, we must associate the drawings to another level of nodes. In some cases, the same drawings can be used, but when the concepts concerned at this level are different, it is necessary to define another drawing. By changing drawings and associated nodes, it is possible to modify the abstraction level of the visual results. The idea is to create new visualization rules in order to associate more abstract drawings to higher level nodes.

Different type of paradigms
If there is a very different source language from a different programming paradigm, it is necessary to verify which concepts are common and which are not. For the last

ones, new visualization rules must be defined, new DAST nodes must be created, and new semantics must be specified with rewriting rules. This section will briefly show an example in Prolog.
Consider the following input program:

```
mother(julie,susan).
mother(susan,john).
father(peter,paul).
father(peter,susan).
parents(M,P,E):-mother(M,E),father(P,E).
```

A *front-end* is needed to map the Prolog concepts to Alma nodes. An extended LISA grammar associates facts and rules to a PROCDEF node, because this node represents the definition of a code block that can be invoked from any place of the program. For each query an execution tree is created.

The animation of the execution tree (simulation of the proof process) uses the visualization rules already defined for other languages. In a similar way, the same rewriting rules are used to simulate procedure calls.
For example, consider the following query:

```
? - parents(M,P,susan).
```

The output can be seen in figure 19, which presents the less detailed version of the generated animation (the minimal number of steps are shown).
Figure 20 shows another kind of visualization for the same program. This visualization is obtained by using other visualization rules.
This example illustrates the possibility of reusing the visualization and rewriting rules, already defined in Alma for imperative languages, to animate declarative programs (proof processes).
These examples assume that Alma will be mainly used in small programs, or with DSLs for which there are no debugging tools or visualizers. Alma produces graphical representations that usually have problems of scalability and it's also very difficult to chose the appropriate drawings for better understanding. This discussion did not include details on output quality or the system performance. The focus of this section was the approach to visualize automatically different concepts and different languages using the same DAST-based approach.

5 Conclusion

Many applications today are written in well-understood domains. One trend in programming is to provide software tools designed specifically to handle the

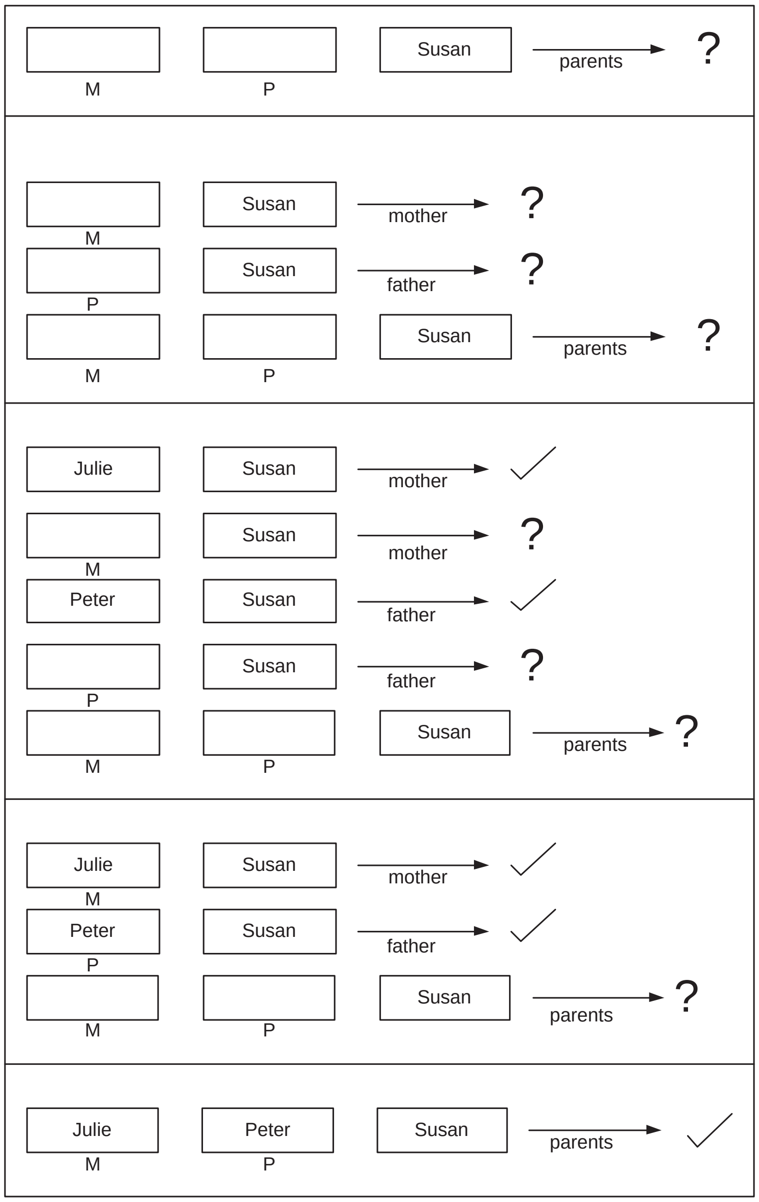


Fig. 19. Alma generated animation

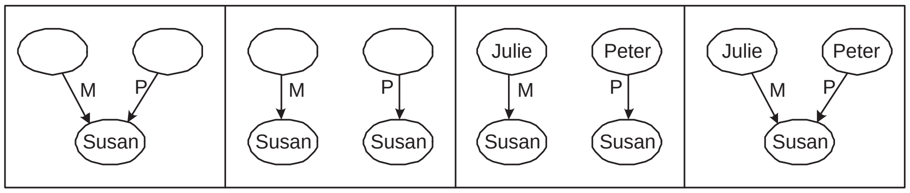


Fig. 20. Another Alma generated animation

development of domain-specific applications in order to greatly simplify their construction. These tools take a high-level description of the specific task and generate a complete application.

One such well established domain is compiler construction, because there is a long tradition of producing compilers by hand using an underlying theory that is well understood (supporting all the analysis phases, and even code generation and optimization processes). At present, there exist many generators that automatically produce language-based tools from programming language specifications. Although particular automatic generation of language-based tools was discussed before, in this paper a more general approach is taken by identifying generic and specific parts from which language-based tools can be generated automatically from language specifications. Previous generators varied widely in what constituted the generic and specific parts. Such classification can also be a base for comparing systems that automatically generate language-based tools. In order to generate automatically language-base tools, it is often the case that a language specifications needs to be extended or appropriate information needs to be extracted. Concrete examples of both types, produced by the generator system LISA, were introduced and discussed in the paper.

The benefits of automatically generated language-based tools should not be ignored. Building language-based tools from scratch (especially for DSLs) is time consuming and error prone, which makes tool maintenance very costly.

6 Acknowledgements

We would like to thank anonymous reviewers for useful comments on earlier versions.

References

[1] J. Heering, P. Klint, Semantics of programming languages: A tool-oriented approach, ACM Sigplan Notices 35 (3) (2000) 39–48.

[2] T. Reps, T. Teitelbaum, The Synthesizer Generator, ACM SIGPLAN Notices 19 (5) (1984) 42–48.

[3] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, CENTAUR: The system, ACM SIGPLAN Notices 24 (2) (1989) 14–24.

[4] M. Anlauff, P. Kutter, A. Pierantonio, Formal aspects and development environments for Montages, in: 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing, Springer/British Computer Society, 1997.

[5] I. Attali, C. Courbis, P. Degenne, A. Fau, D. Parigot, C. Pasquier, SmartTools: A generator of interactive environments tools, in: 10th International Conference on Compiler Construction, Vol. 2027, Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 355–360.

[6] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, LISA: An Interactive Environment for Programming Language Development, in: N. Horspool (Ed.), 11th International Conference on Compiler Construction, Vol. 2304, Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 1–4.

[7] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, MA, 1979.

[8] D. Knuth, Semantics of contex-free languages, Math. Syst. Theory 2 (2) (1968) 127–145.

[9] J. Paakki, Attribute grammar paradigms - a high-level methodology in language implementation, ACM Computing Surveys 27 (2) (1995) 196–255.

[10] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, R. Nutt, The fortran automatic coding system, in: Western Joint Computer Conference, 1957, pp. 188–198.

[11] M. Jourdan, D. Parigot, The FNC-2 system user's guide and reference manual, release 1.19, Tech. rep., INRIA Rocquencourt (1997).

[12] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Oliver, J. Scheerder, J. Vinju, E. Visser, J. Visser, The ASF+SDF Meta-environment: A component-based language development environment, in: 10th International Conference on Compiler Construction, Vol. 2027, Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 365–370.

[13] J. Saraiva, M. Kuiper, Lrc - a generator for incremental language-oriented tools, in: 7th International Conference on Compiler Construction, Vol. 1383, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[14] I. Attali, D. Caromel, S. O. Ehmety, S. Lippi, Semantic-based visualization for parallel object-oriented programming, ACM SIGPLAN Notices 31 (10) (1996) 421–440.

[15] I. Attali, D. Caromel, M. Russo, Graphical visualization of Java objects, threads, and locks, IEEE Distributed Systems Online 2 (1).
URL `http://dsonline.computer.org/0101/features/att0101_print.htm`

[16] M. Mernik, M. Lenič, Enis Avdičaušević, V. Žumer, Multiple Attribute Grammar Inheritance, Informatica 24 (3) (2000) 319–328.

[17] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, A reusable object-oriented approach to formal specifications of programming languages, L'Objet 4 (3) (1998) 273–306.

[18] E. Avdičaušević, M. Lenič, M. Mernik, V. Žumer, AspectCOOL: An experiment in design and implementation of aspect-oriented language, ACM SIGPLAN Notices 36 (12) (2001) 84–94.

[19] M. Mernik, U. Novak, E. Avdičaušević, M. Lenič, V. Žumer, Design and implementation of simple object description language, in: ACM Symposium on Applied Computing, SAC'2001, 2001, pp. 590–594.

[20] F. Javed, M. Mernik, J. Zhang, J. Gray, B. Bryant, Mars: A metamodel recovery system using grammar inference, Tech. rep., University of Alabama at Birmingham (2004).

[21] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[22] H. Wu, J. Gray, M. Mernik, Debugging domain-specific languages in eclipse, in: Eclipse Technology Exchange Poster Session at OOPSLA 2004, 2004.

[23] D. Wright, B. Freeman-Benson, How to write an eclipse debugger, in: Eclipse Corner, Fall 2004, http://www.eclipse.org/articles/index.html.

[24] jdb - the java debugger, available from http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html.

[25] H. Wu, J. Gray, S. Roychoudhury, M. Mernik, Weaving a debugging aspect into domain-specific language grammars, in: ACM Symposium for Applied Computing (SAC) - Programming for Separation of Concerns Track, To appear, 2005.

[26] A centaur tutorial, version 2.0, Tech. rep. (1994).