Tool Support for Language Design and Prototyping with Montages

Matthias Anlauff[1], Philipp W. Kutter[2], and Alfonso Pierantonio[2]

[1] GMD FIRST, Berlin, Germany
[2] Institute TIK, ETH Zürich, Switzerland

Abstract. In this paper, we describe the tool Gem-Mex, supporting Montages, a visual formalism for the specification of (imperative/object oriented) programming languages.

1 Introduction

Montages [KP97] form a graphical notation for expressing the syntax and semantics of imperative and object oriented languages. Every syntactic construct has its meaning specified by a Montage, which consist of a diagram defining graphically the control and data flow, and textually the semantic actions of the construct. The control-flow edges show how behavior of the construct is composed from the behavior of its components, and data-flow edges enable semantic actions to access the attributes of the construct's components.

The Montages formalism is tuned towards ease of writability, readability, maintenance, and reuse of specifications. The hope is that, like Wirth's 'syntax diagrams', a graphical notation may help in making formal language descriptions more accessible to the everyday user. It does not compete with existing compiler construction techniques which are designed for the flexible specification of efficient translations.

The notation is supported by the Gem-Mex (**G**raphical **E**ditor for **M**ontages and **M**ontages **EX**ecutable generator) tool[1] which guides the user through the process of designing and specifying a language. In other words, Gem-Mex forms a (meta) environment to enter and maintain the description of a language. In addition, it generates a specialized (programming) environment which allows to analyze, execute, and animate programs of the specified language. The language designer can use the generated environment as a semantics inspection tool to test and validate design decisions. Later the programmer can use the same environment for debugging programs. Further, in some situations the generated interpreter may be directly used as implementation of the language. Since language-documentation is based on the Montages descriptions, and the language-environment is deduced from this descriptions, it is guaranteed that design, documentation, debugging-support and implementation of the language are

[1] The tool runs on Unix platforms and Windows NT; it is freely available at `http://www.tik.ee.ethz.ch/∼montages`

always consistent. Typically, in a realistic scenario many versions of a language will evolve over time and a tool like Gem-Mex is useful to maintain them in a consistent way. The tool is currently used by different groups for both the specification of general purpose programming languages (Pascal, Java, Smalltalk) and the design of domain specific languages [KST98].

2 The Montage Formalism

consist of three parts

1. A canonical definition of abstract syntax trees (ASTs) from EBNF rules.
2. A simple attribution mechanism for these ASTs:
   – decoration of ASTs with guarded control-flow edges,
   – decoration the ASTs with data access-arrows (reversed data-flow arrows),
   – decoration the ASTs with dynamic-semantics-actions.

   The attribution rules are given by means of a formal visual language. The visual language relies heavily on the common intuition existing for the graphical notations control/data flow graphs, finite state machines, and flow charts.
3. A computational model based on the control/data flow graphs which result from the attribution process.

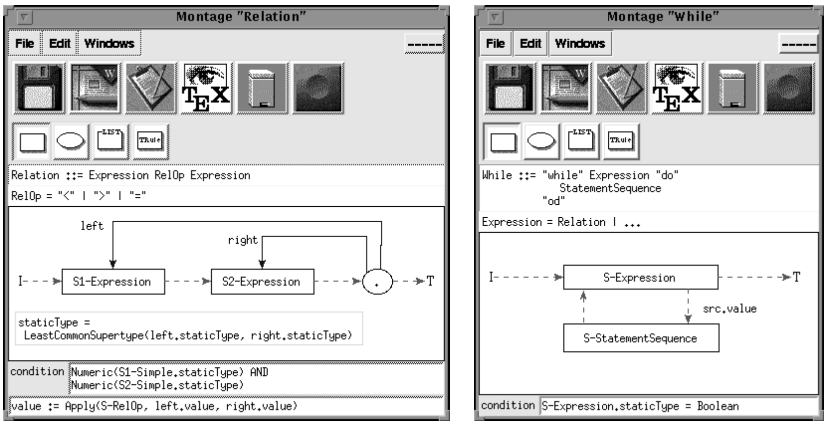
The attribution mechanism is based on standard techniques like attribute grammars [WG84] and graph grammars [REK97].

The dynamic semantic actions are given with Abstract State Machine (ASM) rules [Gur95]. ASMs are a state-based formalism in which a state is updated in discrete time steps. Unlike most state-based system, the state is given by an algebra, that is, a collection of functions and a set of objects. Both functions and sets may change their definitions by means of state transitions which are fired repeatedly.

ASMs have been already used to give the dynamic semantics of full-scaled programming languages (e.g. C, Occam, Java; see [BH98] for a commented bibliography) as well as for the systematic development of provably correct compilers [ZG97].

The computational model based on the control/data flow graphs is formalized with an ASM. In this model control flows through the graph, enabling the semantic actions attributed to the nodes. These actions use the data arrows to access attributes of other nodes, and redefine local attributes.

To summarize, a language specification consists of a collection of Montages, each specifying the meaning of one syntactic construct in the language. Such a Montage consists of subparts, containing the EBNF rule, the control and data flow attributions, the static semantics condition, and the dynamic semantics action. We show the Relation and the While Montage in Fig. 1. The dotted arrows define the control flow, the symbol I denotes the point where control enters from outside, and the T denotes the point where control exits. As expected the Relation Montage has a control flow, that first goes through the two subexpressions,



```
Relation ::= Expression RelOp Expression
RelOp = "<" | ">" | "="
staticType =
  LeastCommonSupertype(left.staticType, right.staticType)
condition Numeric(S1-Simple.staticType) AND
          Numeric(S2-Simple.staticType)
value := Apply(S-RelOp, left.value, right.value)
```

```
While ::= "while" Expression "do"
          StatementSequence
        "od"
Expression = Relation | ...
condition S-Expression.staticType = Boolean
```

Fig. 1. Montages examples

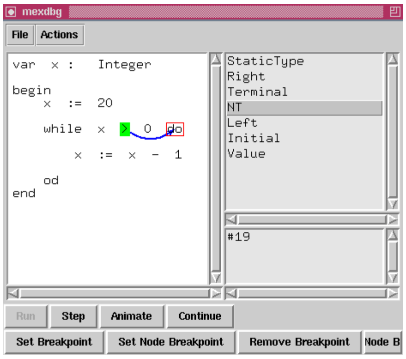
which are depicted with boxes. Then the control flow enters the oval, denoting a semantic action, which is given in the last subpart of the Montage. Finally the control leaves through the T, without any branches.

In contrast, the control flow of the While Montage contains a cycle. Whether to cycle or to leave the construct is dependent on the guard which is given as the label of the control arrow from the Expression to the StatementSequence. The *condition* part contains the context sensitive constrains formulated as a first-order logic predicate, e.g. in the Relation Montage the condition states that the types of the Expressions must be compatible. The *staticType* attribute holds the type of the expression evaluated during the static analysis.

3 Tool Architecture

In the introduction we sketched how the user is guided by Gem-Mex through the process of designing and specifying a language using Montages. Fig. 1 shows part of the user interface. The buttons in the first line are used to: save, open a Montage, generate html and tex output, delete, and exit. The buttons in the second line allow to choose specific graphical elements during the editing process of a Montage: boxes for simple components, ovals for semantic actions, special list boxes, additional textual attribution rules. As an example of the generated environment we see in Fig. 2 the user interface of the debugger. On the left side an example program of the specified language is shown. The arrow animates the flow of control through the program text. On the right side, attributes of the construct currently under execution are listed, and by clicking on them their value can inspected. Fig. 3 illustrates the architecture of the tool.

The implementation of Gem-Mex is based on Lex/Yacc, C, and Tcl/Tk. The core is a specifically developed ASM to C compiler called Aslan [Anl98] (ASM Language). Aslan provides support for structuring ASMs in an object oriented



```
var  x :  Integer

begin
    x  :=  20

    while  x  >  0  do

        x  :=  x  -  1

    od
end
```

```
StaticType
Right
Terminal
NT
Left
Initial
Value
```

```
#19
```

Fig. 2. The generated animation tool

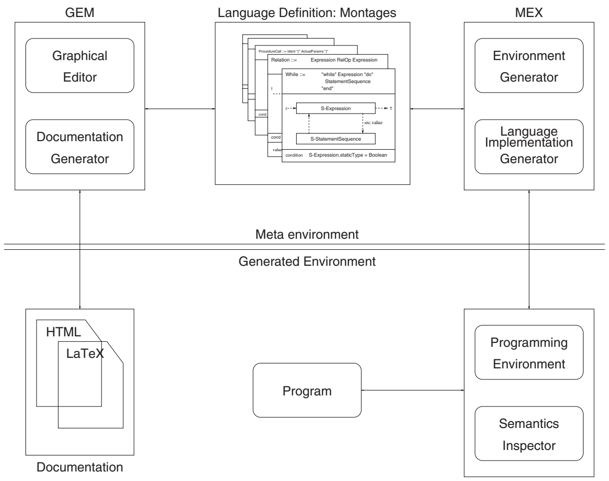


Fig. 3. Tool architecture

way. Gem-Mex generates for each Montage the ASM semantic in form of an Aslan class. Then this classes are compiled and executed.

References

Anl98. M. Anlauff. The aslan reference manual. Technical report, GMD FIRST, Berlin, Germany, 1998. 298

BH98. E. Börger and J. Huggins. Commented asm bibliography. In *EATCS Bulletin*, number 64, pages 105 – 127. 1998. http://www.eecs.umich.edu/gasm. 297

Gur95. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995. 297

KP97. P.W. Kutter and A. Pierantonio. Montages: Specification of Realistic Programming Languages. *J.UCS, Springer*, 3(5):416 – 442, 1997. 296

KST98. P.W. Kutter, D. Schweizer, and L. Thiele. Integrating formal domain specific language design in the software life cycle. In *Current Trends in Applied Formal Methods, Boppard, Germany*, LNCS. Springer, 1998. 297

REK97. G. Rozenberg, G. Engels, and H.J. Kreowski. *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific, 1997. 297

WG84. W.M. Waite and G. Goos. *Compiler Construction*. Springer, 1984. 297

ZG97. W. Zimmermann and T. Gaul. On the construction of correct compiler backends: An asm approach. *J.UCS, Springer*, 3(5):504 – 567, 1997. 297