Montages/Gem-Mex: a meta visual programming generator

Montages/Gem-Mex: a Meta Visual Programming Generator

M. Anlauff
GMD FIRST
Berlin, Germany
ma@first.gmd.de

P.W. Kutter
ETH Zürich
Zürich, Switzerland
kutter@tik.ee.ethz.ch

A. Pierantonio
Università di L'Aquila
L'Aquila, Italy
alfonso@univaq.it

Abstract

Last decade witnessed a disappointing lack in technology transfer from formal semantics to language design. Research in formal semantics has developed increasingly complex concepts and notation, at the expense of calculational clarity and applicability in the development of languages.

Montages is a visual domain-specific formalism for specifying all the aspects of a programming language. It is intelligible to a broad range of people involved in the language life cycle, from design to programming. Language descriptions are fed to a rapid prototyping tool, called Gem-Mex, which generates a visual programming environment for the given language.

Gem-Mex consists of a graphical front-end which allows a comfortable editing of the visual components of the specification. Starting from these visual descriptions the tool is able to generate in an automatic way high-quality documents, type-checkers, interpreters and a visual symbolic debugger. All these products form a powerful suite where the programmer can write, execute, animate and debug programs written in the specified language.

1 Introduction

Over the last two decades, formal semantics of programming languages followed considerably the trend towards higher levels of specialization, at the expense of computational clarity. This situation generated more experts than general users and hampered the technology transfer from semantics research to mainstream computing. It is striking how programming language designers make such little use of formal methods. Most of the languages have been specified informally. Even if a formal specification is undertaken, semanticists are expected to play the role of morticians in the community of programming languages, e.g. analyzing an already designed type-system. In other words, formal semantics have not played the same central role in language design, implementation and understanding as have formal-syntax techniques.

Of course, the situation is not uniform and often semantics research influenced the design of procedural and type-checking mechanism or security aspects in several projects. But still it is surprising and disappointing that language design methods of the 1970s and 1980s are still being used in the 1990s. Formal specifications of semantics are still uncommon and tend to appear years after the corresponding informal specifications. They are usually incomplete and/or inaccurate, and considered as not definitive. Language designers still rely in informal specifications which are open to incompleteness, inconsistency and ambiguity.

In order to make the language designer reconsider their attitude to formal methods, we proposed a formalism, called Montages [KP97b]. The main aim of such a mathematical framework is to adopt a "user-friendly" interface to the underlying mathematical machinery. By mean of visual descriptions the designer can map her/his intuitions to semantical definitions, which can be fed to a rapid prototyping tool, called Gem-Mex.

In this paper we describe the visual aspects of Montages and Gem-Mex. We show how the visual specification of control and data flow of language constructs are used to visualize the control and data-flow of programs. We highlight how a uniform visualization is used for language specification and for the generated program animation tools. This allows to take advantage of the visual specification during program visualization.

This paper is organized as follow. In the next section, we illustrate the Montages formalism and its semantics mainly informally and by means of a simple example. Sect. 3 the Gem-Mex tool is present by giving an overall picture of the system and how it can assist the user in editing the visual specifications and generating program animation and debugging tools.

Finally, in Sect. 4 we draw some conclusions.

2 Background: Montages

Montages [KP97b] constitute a specification formalism for describing all aspects of programming languages. Syntax, static analysis and semantics, and dynamic semantics are given in an unambiguous and coherent way by means of semi–visual descriptions. The static aspects of Montages resemble control and data flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. Montages are designed to provide a theoretical basis for a number of activities from initial language design to prototyping.

The mathematical semantics of Montages is given with Abstract State Machines (formally called Evolving Algebras) [Gur95, ASM95] In short, ASMs are a state–based formalism in which a state is updated in discrete time steps. Unlike most state based systems, the state is given by an algebra, that is, a collection of functions and universes. The state transitions are given by rules that update functions pointwise and extend universes with new elements.

ASMs have already been used to model the dynamic semantics of a number of programming languages, such as Occam [BDR94], C [GH93], C++ [Wal95] and Oberon [Kut97] to mention a few. At the risk of oversimplifying somewhat, one defines the *initial state* of the functions and specifies how they evolve by means of *transition rules*. The *initial state* is assumed to include the results of a static analysis. After this analysis the program's control and data flow is represented in the form of functions between parts of the program text. As usual the control flow functions specify the order in which statements are executed, and the data flow functions specify how values flow via variables through operations. The corresponding *transition rules* update the system state and let the control evolve through the control flow.

The existing case studies showed that it is possible to model with ASMs the dynamic semantics of realistic programming languages, but they have the disadvantage that they do not formalize the static aspects. Montages engineered the ASM's approach to programming language semantics showing how to model consistently not only the dynamic semantics, but the static analysis and semantics as well. In particular, we describe how to define intensionally the abstract syntax, i.e. the control and data flow, starting from the concrete one. This mapping is provided by means of graphs which confer to the specification a great intelligibility.

A language specification is given by a collection of Montages, which is hierarchically structured according to the rules of the corresponding context-free grammar given in EBNF [Wir77]. Each Montage is a "BNF-extension-to-semantics", that is a self contained description in which all the properties of a given construct are formally defined.

In Fig. 1 the Montage specification of a "While" construct is presented as it looks if edited with the Gem-Mex tool. The topmost part in the working area is the production rule defining the context–free syntax. Below is a graphical representation of a pattern in the parse tree, and of the control and data flow

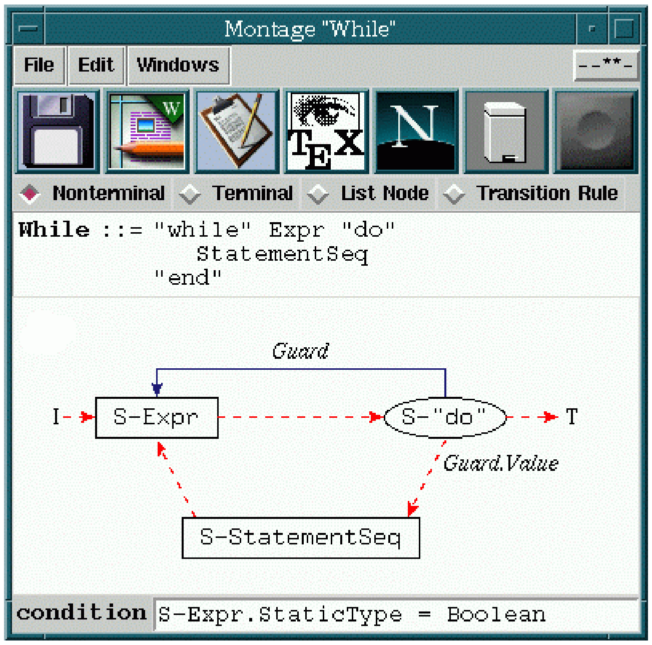
graph. Inner nodes of the parse tree are represented with boxes and leaves with ovals. Nested boxes are used to represent nodes on lover levels of the parse trees. The solid and dotted arrows denote the data and control flow, respectively. Control flow arrows are labeled by means of boolean predicates which determine through which edges the control flows from one state to the next, e.g. the predicate *Guard.Value* indicates that the control is passed from a "do"-token to the sequence of statements *StatementSeq* whenever the value of the expression *Expr*, which is retrieved by means of the data flow *Guard*, is evaluated to *true*. For the sake of simplicity, we allow to omit the label in certain situations:

- If all outgoing control arrows of a node have the label *true*, this label may be omitted.
- If two control arrows c_1 and c_2 leave a node, and c_1 is labeled with the negation of c_2's label, one of the labels may be omitted.
- If several control arrows $c_1, \ldots, c_n$ leave a node, the label of one arrow c_i may be omitted, if it is equal to

  $$\neg l_1 \wedge \ldots \wedge \neg l_{i-1} \wedge \neg l_{i+1} \wedge \ldots \wedge \neg l_n$$

  where l_j is the label of c_j. This is a generalization of the second case.

The control flow arrows I (initial) and T (terminal) are special arrows which serve to plug together the local flow-information to the global one. The third part of the While Montage contains the static semantics, that is, the type of the While-condition must be Boolean. The designer may make use of full first-order logic to express context sensitive constraints. In this example, the part containing explicit dynamic semantics rules is missing. This is usual for most of the control statements, but there are also cases in which an additional



```
While ::= "while" Expr "do"
             StatementSeq
          "end"
```

```
condition  S-Expr.StaticType = Boolean
```

Figure 1: The While Montage

transition rule is needed to define the complete dynamic semantics as illustrate, for instance, in Fig. 3. With respect to earlier works [KP97b, AKP97] the implicit control flow is a recent enhancement of the visual formalism of Montages.

The semantics of a Montages specification of a programming language is the following, given a program

- the context-free grammar obtained by collecting all the EBNF-rules in each Montage defines the concrete syntax;
- the visual part of the Montages is the mapping between the concrete and abstract syntax, i.e. it defines an inductive decoration of the parse tree; in other words the control and data flow arrows are mapped to pointwise definition of functions in the resulting ASM. This information is needed, in turn, by the dynamic semantics;
- the condition part is a first-order logic predicate which is evaluated while traversing the parse tree, i.e. for each internal node the corresponding predicate is checked. In general, different traversal strategies can be specified, predicates can be checked before or after the analysis of subtrees, or maybe the designer may prefer to define several passes, e.g. one additional pass to check declarations before the other fragments. A more leisured and detailed discussion can be found in [KP97b].
- the dynamic semantics part is a transition rules which is fired whenever the control reach that given construct. For instance, in fig 3 whenever the control reaches the node denoted by ":=" the following rule is enabled

  ```
  @``:='':
     Variable.Decl.Value := RHS.Value
  end
  ```

  which updates locally the value of the declaration of the left-hand-side variable to the value of the right-hand-side expression.

The logical and formal aspects of Montages are not supposed to be illustrated deeply in this work, a more detailed discussion can be found in [KP97b]. Nevertheless, it should be stressed how Montages represents a rigorous and formal instrument providing a mathematical framework which can be used by the designer to record unambiguously decisions about a particular language and obtaining new insight into the nature of the language developing description of it. In any case the long-term aim to generate complete, correct and efficient implementations automatically from language description rely heavily on a robust tool support and on the satisfaction of the general user.

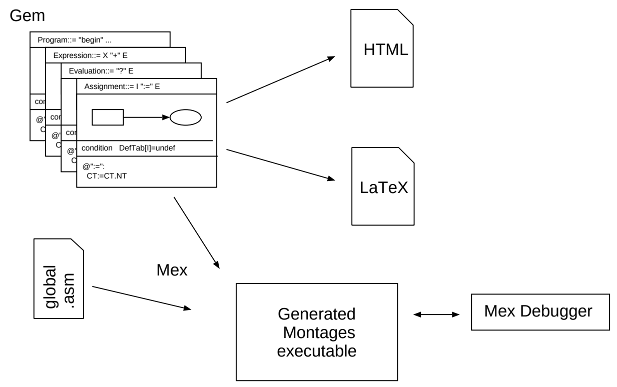


Figure 2: The General Structure of the Gem-Mex System

3 Gem-Mex: The Development Environment for Montages

The development environment for Montages is given by the Gem-Mex tool [Anl97], whose architecture is depicted in figure 2. It is a complex system which assists the designer in a number of activities related with the language life cycle, from early design to routine programmer usage.

It consists of a number of interconnected components

- the Graphical Editor for Montages (Gem) is a sophisticated graphical editor in which Montages can be entered; furthermore high-quality documentation can be generated;
- the Montages executable generator (Mex) which automatically generates correct and efficient implementations of the language;
- the generic animation and debugger tool visualizes the static and dynamical behavior of the specified language at a symbolic level; source programs written in the specified language can be animated and inspected in a visual environment.

A broad range of professionals may find interesting and convenient to use Gem-Mex. The whole development of a programming language can be supported with an effective impact on the productivity and robustness of the design. The designer can enter the specification, browse it and especially maintain it. Specifications may evolve in time even in a non-monotonic way since modifications can be localized within very neat boundaries. By doing so, different experimentation can take place with different versions of the syntax and semantics of the specified language in a very short time.

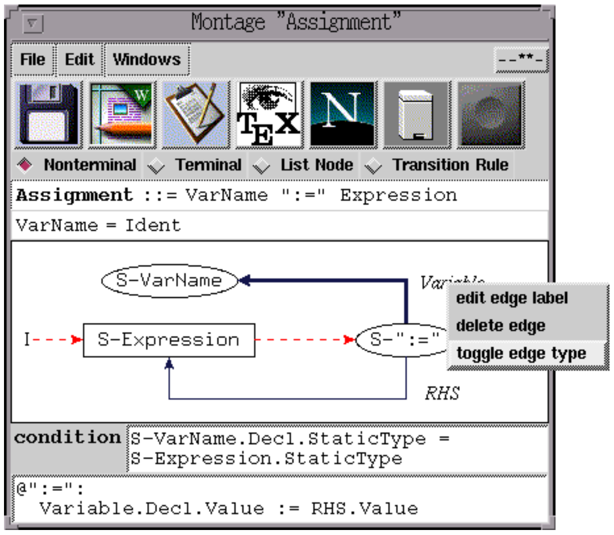
Besides the pure editing functionality, Gem can be used to generate documents suitable for specification presentation. Experience suggests how lack in documentation is a dangerous bottleneck for the consistency and coherence of a project. Both, paper and online presentation of the language specification are automatically generated by Gem:

- LaTeX documents illustrate the Montages and the grammar; such documents are easily customizable for the non-specialist user;
- HTML versions of the language specification allows to browse the specification and retrieve pieces of specification.

Moreover, intelligibility is enhanced by means of “literate specification” techniques directly supported by Gem. Formal parts of the specification can be substituted with textual elements by means of a “literate programming” tool integrated in the system. “Literate specification” means that the Montages text fields may contain references to other parts of the formalization specified outside of the Montages modules. Thus, the readability and comprehension of a Montages specification results very much similar to those of language manuals which are open to misunderstanding.

3.1 Visual Editing

As already mentioned the Graphical Editor for Montages (Gem) is a specialized drawing tool for Montages. The Gem user interface is divided into several sub-windows, corresponding to the four components of a Montage. The user can enter the several parts: writing the production rule; drawing the graphs in the drawing area; writing the first-order predicate for the context sensitive constraint; and finally writing the transition rule corresponding to the dynamic semantics.

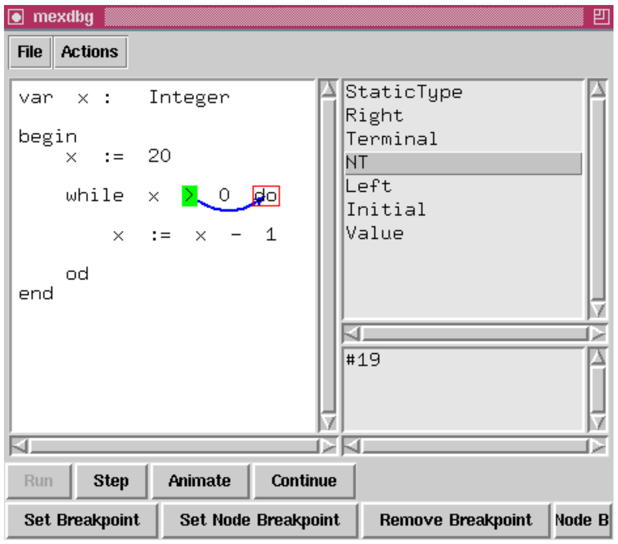


```
Assignment ::= VarName ":=" Expression
VarName = Ident
```

```
condition S-VarName.Decl.StaticType =
          S-Expression.StaticType
```

```
@":=":
  Variable.Decl.Value := RHS.Value
```

Figure 3: Editing the Assignment Montages



```
var  x :   Integer

begin
    x  :=  20

    while  x  >  0  do

        x  :=  x  -  1

    od
end
```

Figure 5: The Gem-Mex animation and debugging tool.

Not all the sub-windows must be present at the same time: Montages which have some missing components are not unusual. Thus different sub-windows can be hidden, and their size is automatically adjusted to the text respectively the drawing. The drawing area contains a visual editor, allowing to create nonterminal and terminal nodes, respectively. Eventually textual static analysis rules can also be added. Nodes and edges can be moved or modified.

The control and data flow arrows are entered by connecting nodes, using the mouse. For the convenience of the user, node labels can be placed at several positions, nested nodes can be moved as a unit, and the type of arrows can be toggled between (solid, blue) data and (red, dotted) control flow.

On top of the editor area some icons indicate shortcuts the user may use instead of the menu, such as

- storing, closing, and deleting the current Montage;
- opening another Gem with a Montages either selected in the grammar rule or in a list of all Montages in the working directory;
- regenerating the HTML and LaTeX document associated to the current Montage.

An example of the graphical user interface is shown in Figure 3.

3.2 The Derived Program Visualization

As described in Section 2, the control/data-flow graphs of the single constructs induce a global decoration of the parse tree. A larger example is given in Figure 4. This corresponds to a global control/data-flow graph which is used by the static and dynamic semantics. Thus, it can be used for animation and debugging purposes while execution. The main advantage of this technique is to use the local control/data-flow graph within each Montage to visualize the global flow. The user recognizes the abstract specification of the programming language construct graphically in a concrete program, since the same visual terminology is used. She/he can immediately understand the behavior of a construct by instantiating the rules in the current context.

Another advantage is given by the possibility to visualize non local arrows, e.g. an arrow connecting the occurrence of a variable within a block to its definition. Although such link must be given textually since it is global with respect to a single Montage, it can still be visualized in the global flow graph. The *Decl* arrow depicted in figure 4 illustrate such a situation

In order to enhance the improve the effectiveness, we did not base the animation on the abstract-syntax but Gem-Mex regenerate the program text by means of an internal pretty printer. The nodes of the graph are identified with the tokens of the program text, and the arrows are drawn between the tokens representing their source and target. In figure 5 the animation session based on the above example is shown.

4 Conclusions

In Section 2 we introduce a new way of specifying control flow, which enhances the visualization and simplifies existing Montages specifications considerably. If applied to the Montages specification of Oberon [KP97a], most dynamic rules are simplified to about three lines.

We showed how Montages can be used to generate program animation tools. In contrast to other work in program animation our tool is based on an abstract specification of the programming language, and can thus be considered as more reliable from a correctness point of view. Of course, we cannot compete with systems specially tailored for one specific language, let alone those tuned for a specific application area. But our approach invents to build an animation technique based on programming language concepts, rather than based on specific instances of this construct in different languages. The use of our animation tool, will thus deliver similar results, if applied to the same algorithm coded in different programming languages.

An interesting question is how the Montages approach can be extended from textual languages to visual ones. A problem is, that the intuitions and graphical components of a specified visual language may interfere with the intuitions and graphical components used in Montages. Like this, a main advantage of Montages may be lost, if applied to visual languages. Our style of animation applied to visual languages may lead to an overloaded and unintuitive use of graphics.

References

[AKP97] M. Anlauff, P.W. Kutter, and A. Pierantonio. Formal aspects and development environments of montages. 1997.

[Anl97] M. Anlauff. GemMex-Homepage, 1997. http://www.first.gmd.de/~ma/gemmex/.

[ASM95] ASM-Homepage. 1995. http://www.eecs.umich.edu/gasm/.

[BDR94] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: Simple Mathematical Interpreters. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.

[GH93] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[KP97a] P.W. Kutter and A. Pierantonio. The formal specification of oberon. *Journal of Universal Computer Science*, 3(5), 1997.

[KP97b] P.W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5), 1997.

[Kut97] P.W. Kutter. Dynamic semantics of the programming language oberon. Technical report, ETH Zürich, 1997.

[Wal95] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[Wir77] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11), 1977.

```
var x:Integer
begin
   while x > 0 do
      x := x - 1
   od
end
```

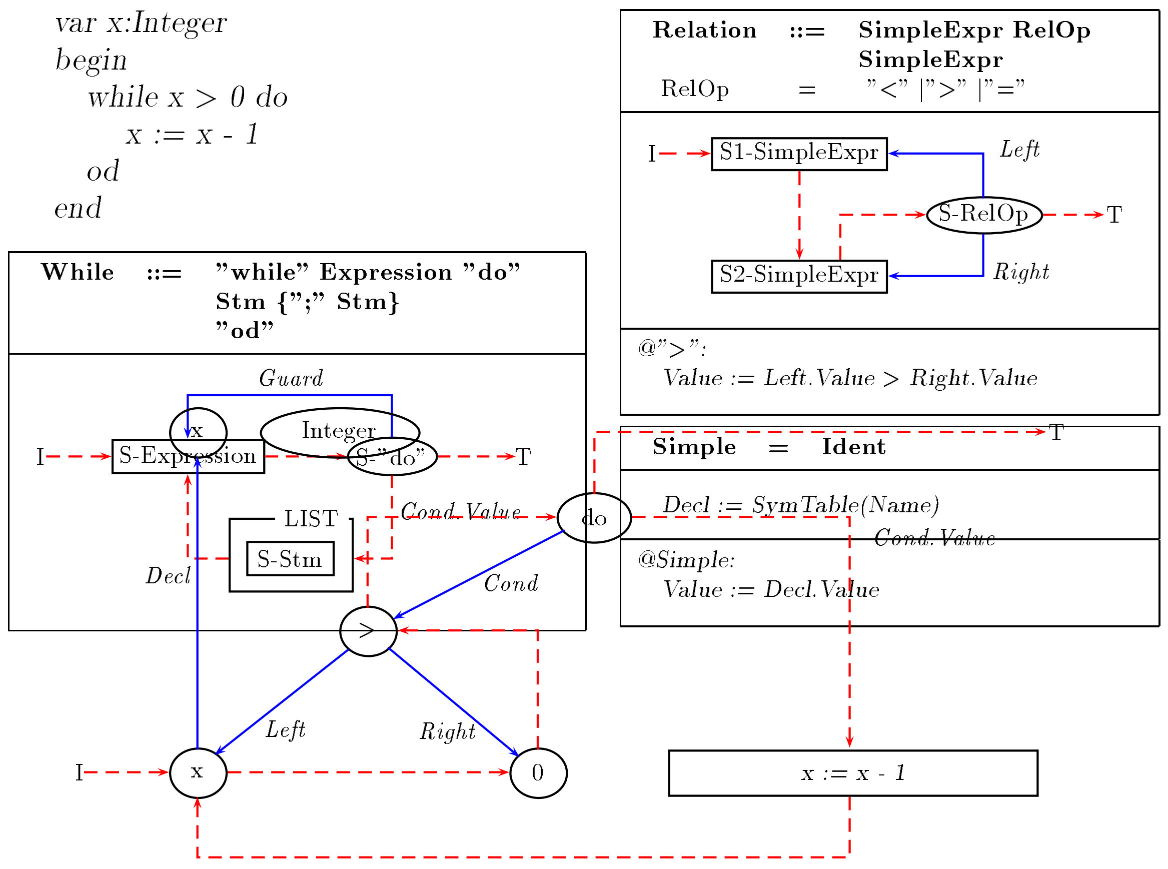


Figure 4: The local flow graphs of the Montages and the global flow graph of an example program.