Formal Aspects of and Development Environments for Montages

Matthias Anlauff
International Computer Science Institute
Berkeley, CA 94704, USA
anlauff@acm.org

Philipp W. Kutter
Eidgenössische Technische Hochschule
CH–8092 Zürich, Switzerland
kutter@tik.ee.ethz.ch

Alfonso Pierantonio
Università di L'Aquila
I–67100 L'Aquila, Italy
alfonso@univaq.it

August 31, 1997

Abstract

The specification of all aspects of a programming language requires adequate formal models and tool support. Montages specifications combine graphical and textual elements to yield language descriptions similar in structure, length, and complexity to those in common language manuals, but with a formal semantics. A broad range of people involved in programming language design and use may find convenient to use Montages in combination with the tool Gem–Mex. It allows the automatic generation of high–quality documents, type–checkers, interpreters and symbolic debuggers.

1 Introduction

Montages [19] are a specification formalism for describing all aspects of programming languages. Syntax, static analysis and semantics, and dynamic semantics are given in a unified 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. Thus, Montages are a formal instrument which can be equally well understood by language designers, compiler constructors, and programmers.

Based on Abstract State Machines (formally called Evolving Algebras) [12] Montages provide a theoretical basis for a number of activities from initial language design to prototyping. ASMs have been proposed by Y.Gurevich as a dynamic generalization of multi–sorted algebras, intended to provide a more versatile notion of Turing machine, "able to simulate arbitrary algorithms in a direct and essentially coding–free way" [12]. 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 programming languages such as Prolog [7], Occam [5], C [13], C++ [26], Oberon [17], and VHDL [6]. At the risk of oversimplifying somewhat, we can describe some of these models [13, 26, 17] as follows. Program execution is modeled by the evolution of two functions CT and S. The current task CT represents the part of the program text currently in execution and may be seen as an abstract program counter. S represents the current value of the store. Formally one define the *initial state* of the functions and specifies how they evolve by means of *transition rules*.

In the described models [13, 26, 17], the *initial state* is assumed to include the results of a static analysis, which is only described informally. This analysis provides a representation of the program's control and data flow 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 program counter and program state using the control flow and data flow functions.

Montages suggest how to use ASMs to model not only the dynamic semantics of a programming language, but the static analysis and semantics as well. In particular, we show how to generate the control and data flow, i.e. what for us correspond to the abstract syntax, starting from the concrete one. This mapping is provided by means of graphs which confer to the specification a certain intelligibility.

In this paper, we show some toy examples. The specification method scales–up to realistic languages, e.g. in [20] is the complete specification of the whole language Oberon can be found. Complex features as encapsulation, modularity, inheritance and pointers are covered in a surprisingly short and comprehensive manner. Montages have been used also in [2] and [10] for formalizing the object–oriented language Sather and the SQL direct (ISO9075), respectively.

The collection of Montages defining a language may be used used for generating automatically a number of tools, such as type–checkers, interpreters and symbolic debuggers. This is accomplished by means of the GEM–MEX language suite which provides a convenient and comfortable environment. These tools feature also the possibility to generate high quality documents suitable for presentations and reference manuals.

The paper is organized as follows. In section 2, we define some prerequisites. Montages are presentated in section 3. Then we illustrate the tool GEM–MEX. Finally, in the last two sections we provide a brief comparison with related work and draw conclusions.

2 Prerequisites

In the following sections we give some preliminary notions. In section 2.1 we recall the notion of context free grammars, look at the related derivation trees, and introduce *compact derivation trees*. The main point of the section is the observation, that the nodes of compact derivation trees are characterized by a specific subset of the symbols in the grammar. This subset, the so–called *characteristic symbols* is the base for the syntax driven modularity of Montages. We introduce the notion of *initial state* of ASMs, and specify how a compact derivation tree is mapped in such a state.

In section 2.2 we introduce the notion of *transition rule* of ASMs, and show some generic transition rules which can be used to traverse a compact derivation tree parallelly and sequentially. Although all used aspects of ASMs are explained during the sections, we have to be rather short. For a more complete treatment and motivations we refer to [12, 4].

2.1 Initial State and Tree Representation

Given a context free grammar of a language, the generation of a string S of that language can be described by means of a *derivation tree*. The root of the tree is labeled with the start symbol. We say as well, the root represents the start symbol. Every replacement of a nonterminal n by $s_1\ s_2\ \ldots\ s_m$ in the derivation of S is represented in the tree by appending from left to right nodes representing s_1 to s_m to the node representing n. The new nodes are labeled with the corresponding symbols s_1 to s_m. Such trees can be made more compact by putting multiple labels in the case of *synonym productions* [22]. Synonym productions are rules of the form $n ::= s_1|s_2|\ldots|s_m$, which give place to nodes with only one child. In such cases we simply do not append new nodes but we keep track of the synonym productions by adding a new label to the current node. The resulting trees are called *compact derivation trees* and we distinguish a synonym production $n ::= E$ by writing $n = E$. To exemplify the situation we give in fig. 1 the normal and the compact version of a term $a + b * (c + D) + e$ of a typical expression language.

According to the above definitions, each node is labeled with at least

- one terminal or
- one non-terminal, which is the left-hand-side of a non-synonym production.

Such symbols are called *characteristic symbols* since it can be shown that each node is labeled with exactly one of them. If a node is labeled with a characteristic symbol s we say as well that the node is characterized by s. Such a characterization partitions the set of nodes.

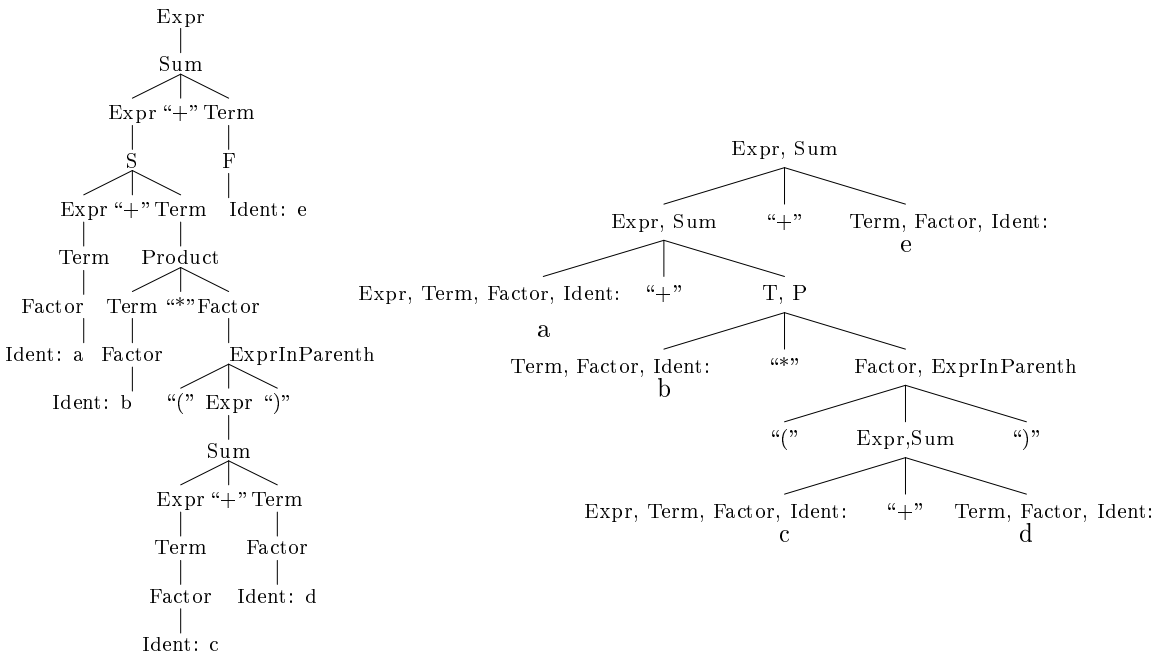


Figure 1: Normal and compact derivation trees

Given a program, its compact derivation tree is represented in the associated initial state. A state of an ASM is given by a set called the *superuniverse* and a collection of functions. The superuniverse has the distinguished elements *true*, *false*, and *undef*. Unary functions from the superuniverse to $\{true, false\}$ are used to represent sets or *universes*. The universe consisting of *true* and $false$ is called *Bool*. Our setting requires some specific universes. In particular, the nodes of the compact derivation tree constitute the universe *Node*. Moreover, each symbol s in $V_t \cup V_n$ is interpreted by a sub-universe of *Node* containing those nodes which are labeled by s.

Example The compact tree in fig. 1 states that universe *Term* contains six elements, namely all nodes with the label *Term*.

As mentioned in the last section, the universes interpreting the characteristic symbols partition the universe *Node*, i.e. each node is in exactly one such universe. But a node might be present in more than one universe if it has multiple labels, which is possible if its derivation includes synonym productions.

Example In fig. 1 the left– and bottom–most node of the compact tree is member of the universes *Expr*, *Term*, *Factor*, and *Ident*, where *Ident* is its unique characterization.

A number of so–called *selector functions* reflects the structure of compact derivation trees and allows us to retrieve the syntactical elements of the program text. Since descendants of a node are constructed by a production rule, we define the functions accordingly. Let x be a node whose descendants have been constructed by a replacement due to a production rule $n ::= E$, then

- If E is of the form "$s_1 s_2 ... s_m$" we access the new nodes in the universes $s_1, s_2 \cdots$, and s_m by unary functions

$$(\text{S-}s_i : Node \to s_i)_{i \in \{1,\ldots,m\}}$$

  If the same symbol s occurs more than once in "$s_1 s_2 \cdots s_m$", we enumerate the functions from left to right: S1-s maps x to the first s-descendant, S2-s to the second and so on.

- If E contains a symbol s in a { } part, then an element of a universe *ListNode* is created and serves as access point of the whole list. The details are given in section 3.2.

Apart from the selector functions, which reflect the structure of the tree, we need also an auxiliary function $Up : Node \rightarrow Node$, which links the descendants to their parents.

The automatic definition of initial state is implemented in the GEM-MEX tool which is described in section 4.

2.2 Transition Rules and Tree Traversal

Given an initial state a transition rule governs the behavior of an ASM. A transition rule is a description of how the current state evolves starting from the initial one. The transition rules are built over the following set of constructs: *update*, *block*, *conditional*, *vary* and *extend*. The update operator is a pointwise modification of a function. Its synopsis is

$$f(t_1, \ldots, t_n) := t_0$$

Such a rule first evaluates the terms $t_1, \ldots, t_n$ and t_0 over the current state to the elements $e_1, \ldots, e_n$ and e_0, and then modifies f on the point $(e_1, \ldots, e_n)$ to e_0. The update of f does not affect the definition of the function over the rest of the domain. A set of transition rules constitute a block and are executed in parallel. For convenience we speak henceforth about "transition rules", if we mean the block of them. A conditional rule consists of a condition C and another rule R.

if C **then**
  R
endif

The *then* branch R is triggered if the value of the guard expression C is evaluated to *true*. A conditional rule may have also an *else* branch, in such a case the corresponding transition rule is triggered if the guard is not evaluated to *true*. The operators *vary*, and *extend* are explained as they are encountered in the paper. The different states of an ASM are reached by iteratively triggering the transition rule until the ASM reaches a state which cannot evolve anymore.

Montages model the static semantics and analysis with an ASM that traverses the tree. In the next section we define what exactly happens during this traversal. In this section we define a general pattern of transition rules which can be used to define a traversal. In instance of the pattern traverses a compact derivation tree of a given grammar, and executes at each node an action, which depends on the characterization. In addition a general technique is introduced allowing to sequentialize the tree traversal in an arbitrary way.

Imperative versus declarative style In [18] we defined tree traversal in an imperative style. Here we use alternatively a definition in a declarative style. The declarative style presented here can be used for parallel traversal as well, which is needed for the derivation of parallel compilers [3]. The imperative version of [18] is a sequential refinement of the declarative version given here. The advantage of the imperative version is that it is easier to read for non-academic programmers. An advantage of the version given here, is that it facilitates extraction of an axiomatic description of the state reached after the graph traversal.

As noted the aim of the traversal is to execute for each node in the tree an action. We start with a rule executing the action for all nodes in parallel. Then we show how to specify an action depending on the characterization of the node. Finally we present a solution, how the execution of the actions can be sequentialized.

Parallel traversal In order to execute an action R for each node, we use the vary construct of ASMs. The rule

vary Self **over** Node (1)
  R
endvary

executes R for each element in Node simultaneously. The bound variable Self can be used in R to access the single elements. If for instance Node is a universe with two elements a and b, the above rule corresponds to a block of twice R, once with Self substituted by a and once with Self substituted by b.

Case distinction by characterization Using the fact that the nodes are partitioned by their characterization, we can execute a specialized rule R_n, the so called *action of n*, for a node characterized by n. Technically we can do this by replacing R in the above vary rule by a block of conditionals, one for each characterizing symbol n:

if n(Self) **then** (2)
  R_n
endif

Such a conditional triggers R_n only, if n(Self) evaluates to *true*, i.e. if Self is in the universe n. For convenience we say henceforth action of a node, if we mean the action of its characterization.

Up to now, for each node, its action is executed in parallel. The next task is to introduce the possibility to sequentialize the execution. Typically the actions should be executed for lower level nodes first, and in some order between the children of a node. The situation where actions of lower level nodes are executed first allows already for direct representation of structural induction: each node (representing a parsed term) can use the results of the actions (definitions) performed for the descendants (representing sub-terms). In addition we need often a certain sequentialisation between descendants, e.g. actions for declaration parts in programs must typically be performed before actions for the statement parts.

For the sequentialisation task, we need a boolean dynamic field

$$\textit{Visited: Node} \rightarrow \textit{Bool}$$

which is initialized with false for each node. This field indicates whether a node has been visited, i.e. whether its action has been executed. A relation

$$\textit{before: Node} \times \textit{Node} \rightarrow \textit{Bool}$$

relates nodes sequentially. The relation (*a before b*) indicates node a must be visited before node b. The relation *before* can be defined with a parallel tree traversal, as we will see in the next section.

Sequentialized traversal Using the above definitions, a sequentialized traversal is defined by the following rule

vary Self **over** Node (3)
  satisfying
    for all node **in** Node **holds**
      node before Self **implies** node.Visited
  R
  Self.Visited := true
endvary

where again R is refined to a case distinction by characterization (2). The final state or termination of a sequentialized tree traversal is reached if the root of the tree is visited.

3 Montages

Montages is a semi–visual formalism which defines for each characteristic symbol the related syntactic and semantic aspects of a language. We start by giving two examples for Montages of programming language constructs.

The first example, given in fig. 2, is a typical While loop. The topmost part is the production rule defining the context free syntax. Below is a graphical representation of the control and data flow graph. The NT (NextTask), and TrueTask arrows denote for instance sequential control flow, while the Condition arrow denotes the data flow. Control flow arrows are dotted and data flow arrows are solid. 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 boxes and circles are labeled with the selector functions accessing the corresponding

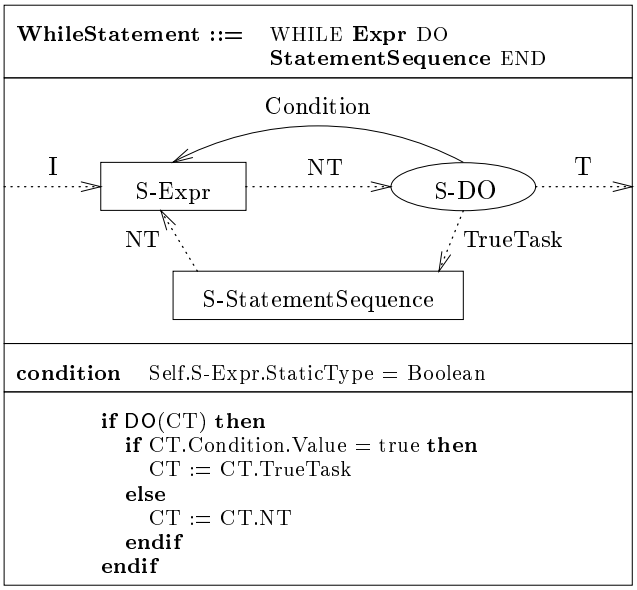


Figure 2: A Montage for a While statement

elements. A circle is used to indicate that the corresponding descendant is a token of the program text. The third part of the While Montage contains the static semantics, that is, the type of the While-condition must be Boolean. The last part contains the dynamic semantics rules. This rule is executed if the abstract program counter CT points to a DO-task. In this case, it checks whether the value of the condition is true. If it is true, the abstract program counter is set to the statement sequence (using the TrueTask arrow), else to the next task. The next task of the DO token is not defined directly by the graph, but it is defined through the mentioned plugging mechanism of the T arrow.

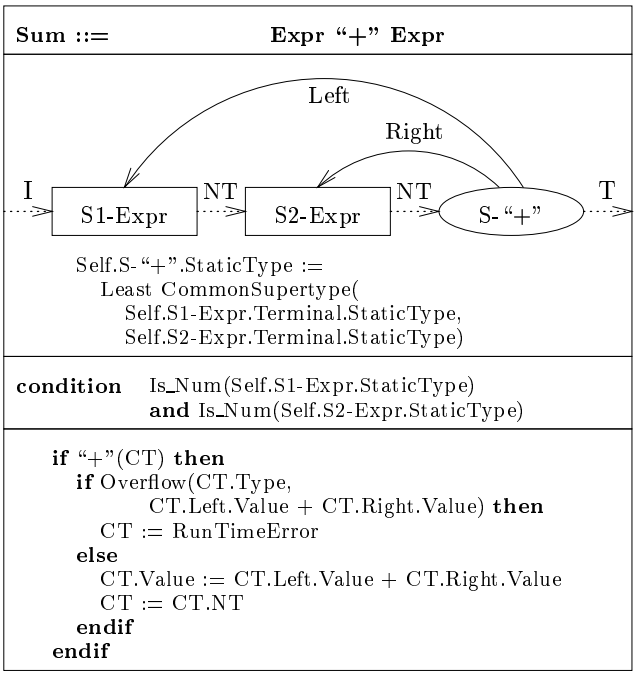


Figure 3: A Montage for a Sum expression

As a second example, we show a Sum Montage (fig.3). The static semantics of the sum expresses that both components must be of numeric type. In the definition we use a static function Is_Num(·) which maps all numeric types to true. The graph specifying control and data flow defines again NT control flow arrows, and two data flow arrows Left, Right, which are used to reference the left respectively right argument of the "+" token. In contrast to the first example, the second part of the Sum Montages contains a textual rule. This rule uses the static function LeastCommonSupertype in order to determine the type of the Sum from the types of the left and right arguments. The dynamic semantics rule of the "+"-token raises a runtime error, if the addition leads to an overflow, with respect to the type of the expression. Otherwise the value of the "+"-token is set to the result of the addition and control is passed to the next task.

It is remarkable how the understanding of a Montage does not require too much expertise as shown in the examples above. The formal semantics given below is an unambiguous arbiter between different ways of understanding and it makes clear how the interaction between the Montages works, e.g. how the I and T arrows plug the local flow information together to global control and data flow graphs. The resulting ASM semantics is exactly as compact as the visual Montages, i.e. each element in a Montages corresponds to an update in the ASM-semantics. In [19] we defined several notational shortcuts, which are not used in this text.

3.1 Basic Definitions

Formally speaking, the semantics of a Montages specification is an ASM $\mathcal{M}$ that for a given program checks the static semantics, initializes the control and data flow functions, and in a second phase executes the dynamic semantics. The transition rule of $\mathcal{M}$ consists thus of two rules, one modeling the first phase, called *statics rule*, and one modeling the second phase, called *dynamics rule*.

The statics rule is a sequentialized traversal as described in the last section. The actions executed by the statics rule check for each node the static semantics, and define control and data flow between leaf descendants. In the Montages framework, the defined flow information is stored as functions between parts of the program text, i.e. leaves of the derivation tree. This functions correspond to the arrows in the examples. As usual the final state of such a sequential traversal is reached if the root is visited. If the traversal is aborted, e.g. a function Abort is set to true, the checked program is not valid. Otherwise we can proceed with the dynamics rule.

The second phase uses the final state of the first as its initial state. The dynamics rule executed in the second phase corresponds to the transition rule of a traditional ASM for dynamic semantics, as described in the introduction. But in contrast to the traditional use of ASMs, where control and data flow are assumed to be given as functions between leaves, here these functions are defined by the first phase.

As exemplified above, Montages are modules containing for a characteristic symbol:

- A production rule, if the symbol is a non-terminal.
- A semi–visual specification of control and data flow functions. The graphical part consists of nodes representing the right-hand-side symbols of the production rule and arcs specifying flow function. The textual part is a transition rule.
- A first–order logic predicate that represents the static semantics constraint. This predicate is marked by the keyword **condition**.
- Transition rules that model the dynamic semantics of terminals generated by the production rule.

These four parts are used to define the statics and dynamics rules. The dynamics rule is simply the block of the rules given in the fourth part of the Montages. The statics rule is a sequentialized traversal (3). The second and third part of a Montage of a symbol s define the action of s in this traversal. The definition of the *before*-relation for the statics rule is done by a parallel traversal (1) with case distinction by characterization (2). The actions of this traversal are defined such that lower nodes in the tree must be visited before higher nodes, and that siblings are visited in the order corresponding to their left-before-right and top-before-bottom order in the graph of the Montages.

We can thus define the s-action in the parallel traversal defining the *before*-relation of the statics rule as follows:

before(Self.S_1, Self.S_2) := true
before(Self.S_2, Self.S_3) := true
...
before(Self.S_{n-1}, Self.S_n) := true
before(Self.S_n, Self) := true

where $S_1, S_2, \ldots, S_n$ are the selector functions accessing the descendants of an s-node in the left-before-right and top-before-bottom order defined by the Montage of s. Please note that the transitive closure of the defined relation is not necessary due to the way how the sequentialisation mechanism works.

To illustrate we give the corresponding actions for the While and Sum Montages.

Example The action for the While Montage (fig. 2) is

before(Self.S-Expr, Self) := true
before(Self.S-DO, Self) := true
before(Self.S-StatementSequence, Self) := true
before(Self.S-Expr, Self.S-DO) := true
before(Self.S-DO, Self.S-Expr) := true

and the action for the Sum Montage (fig. 3) is

before(Self.S1-Expr, Self) := true
before(Self.S2-Expr, Self) := true
before(Self.S-"+", Self) := true
before(Self.S1-Expr, Self.S2-Expr) := true
before(Self.S2-Expr, Self.S-"+") := true

The actions of the statics rule are explained step by step in the following. Lets assume for the discussion a fixed Montage for a symbol s. The action for this Montage is built up as block of updates. Each arrow in the control and data flow graph defines one update in the action. This update links not directly the graphically related nodes but two of their leaf-descendants. The definition which leaf descendants are linked relies heavily on the definition of two functions

$$\mathit{Initial} : \mathit{Node} \to \mathit{Node} \qquad \mathit{Terminal} : \mathit{Node} \to \mathit{Node}$$

which conceptually denote the first and the last leaf in the control flow between the leaf-descendants of a node. These functions are initialized with the identity on nodes, in order to be well defined for leaves, which serve as their own initial and terminal leaf. The definition on inner nodes is built up inductively during the tree traversal. A dotted arrow, labeled with I (respectively T) denotes the direct descendant, whose definition of Initial (respectively Terminal) has to be copied. In the While montage (fig. 2), for instance, the initial leaf of a WhileStatement-node is the initial leaf of the S-Expr-descendant, and the terminal leaf is the terminal leaf of the S-DO-descendant.

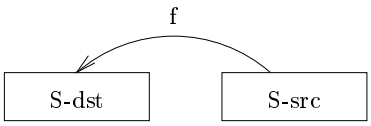
The arrows in the graph define three different kind of updates in the action, one for the above described Initial and Terminal functions, one for data flow arrows,and one for control flow arrows:

1. To define the functions *Initial* and *Terminal*, we specify which node in the graph contains the initial and terminal leaf, respectively. We call this nodes Inode and Tnode. Inode is defined as the target of a dotted arrow labeled with I, and Tnode is the source of a dotted arrow labeled with T. Formally these arrows define two selector functions I and T which link the Self with Inode and Tnode. The corresponding fragment of transition rule obtained by specifying Inode and Tnode graphically is the following block:

   Self.Initial := Self.I.Initial
   Self.Terminal := Self.T.Terminal

   We call this *fragment–1.*

2. Each solid edge as the following

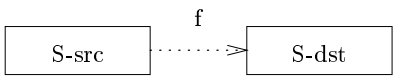


   defines the update

   Self.S-src.Terminal.f := Self.S-dst.Terminal

   The terminal leaf is chosen as target for the control flow. We call the block of these updates *fragment–2.*

3. Each dotted edge as the following



   defines the update

   Self.S-src.Terminal.f := Self.S-dst.Initial

   The defined control functions link the terminal leaf with the initial. We call the block of these updates *fragment–3.*

The action of the statics rules contains in addition to the updates corresponding to the arrows a rule which is given textually in the second part of the Montage. Please have a look at the textual rule in the sum Montage (fig. 3). Its structure resembles to that of updates generated by the second fragment, but it cannot be represented graphically. Theoretically all graphical defined updates can be given textually as well.

The only missing part is the check of the static semantics condition. This condition is checked before the updates of the action happen, and a nullary function *Abort* is set to true if the condition is false. In order to make the rule easier to read, we write the corresponding conditional rule at the beginning of all updates. We discussed now all parts of the action of a characteristic symbol, and can define it as follows:

The action of a characteristic symbol s in the sequentialized traversal being the statics rule of the Montages semantics is

if not Condition **then**
    Abort := true
endif
fragment–1.
fragment–2.
fragment–3.
TransRule

where Condition is the static semantics constraint of Montage s, TransRule is the textual rule in the second part of Montage s, and the *fragment–1.*, *fragment–2.*, and *fragment–3.* are the updates defined by the graph of Montage s.

Example We give for the While (fig. 2) and Sum (fig. 3) Montages the corresponding actions. The action for While is the following rule:

```
if not Self.S-Expr.StaticType = Boolean then
    Abort := true
endif
```

Self.Initial := Self.S-Expr.Initial Self.Terminal := Self.S-DO.Terminal	1
Self.S-DO.Terminal.Condition := Self.S-Expr.Terminal	2
Self.S-Expr.Terminal.NT := Self.S-DO.Initial Self.S-DO.Terminal.TrueTask := Self.S-StatementSequence.Initial Self.S-StatementSequence.Terminal.NT := Self.S-Expr.Initial	3

and the action for the Sum looks as follows:

```
if not (Is_Num(Self.S1-Expr.StaticType) and
    Is_Num(Self.S2-Expr.StaticType)) then
        Abort := true
endif
```

Self.Initial := Self.S1-Expr.Initial Self.Terminal := Self.S-"+".Terminal	1
Self.S-"+".Terminal.Left := Self.S1-Expr.Terminal Self.S-"+".Terminal.Right := Self.S2-Expr.Terminal	2
Self.S1-Expr.Terminal.NT := Self.S2-Expr.Initial Self.S2-Expr.Terminal.NT := Self.S2-"+".Initial	3

```
Self.StaticType :=
    LeastCommonSupertype(
        Self.S1-Expr.StaticType,
            Self.S1-Expr.StaticType)
```

3.2 List Processing

In many approaches a major part of a language specification is concerned with the processing of lists. Therefore we decided to include in Montages a simple, yet powerful list model together with graphical and textual specification elements that can be used to avoid all explicit list processing.

If the right–hand–side of a production rule contains a symbol in a { } part, a list of descendents is generated. As already mentioned we generate as well an additional node, a so called *list node*, that provides access to the elements and to all needed informations about the list. An attribute *ListLength* of the list node is set to the length of the generated list and a binary mix–fix function

$$_[_] : \mathit{ListNode} \times \mathit{Nat} \rightarrow \mathit{Node}$$

can be used to retrieve the elements of the list. The initial and terminal leaves of a list node are defined to be the initial leaf of the first element, respectively the terminal leaf of the last element in the list. If the list is empty, they point to the list node itself, which then serves as dummy element. The dynamic semantics of that dummy element corresponds to the skip command.

For convenience we assume that a number of patterns in the right-hand-side of production rules are recognized and treated as simple lists. These patterns are

{s} s{s} s {"t" s} [s {"t" s}]

where "t" is an arbitrary terminal and *s* a non-terminal. For all these patterns just one list node is generated, which can be accessed by the selector function S-s. The _[_] function can then be used to access all generated *s*-descendants from left to right, regardless by which *s* in the pattern the descendant was generated.

We have not yet defined the statics for list nodes. For simplicity we give just one of many possible solutions. We define the action in the *before*-definition such that the elements must be visited from left to right and that they are visited before the list node Self:

```
vary i over Nat
    satisfying i < Self.ListLength
        before(Self[i], Self[i + 1])
endvary
before(Self[Self.ListLength], Self) := true
```

In addition we chose the action in the statics rule of a list node such, that it sequentially links the elements by means of NT control arrows and sets the initial and terminal leaf to the corresponding leaves of the first and last element:

```
Self.Initial := Self[1].Initial                                  1
Self.Terminal := Self[Self.ListLength].Terminal
-------------------------------------------------------------------
vary i over Nat                                                  3
satisfying i < Self.ListLength
Self[i].Terminal.NT := Self[i + 1].Initial
endvary
```

These two definitions are a simple solution, which works fine for many examples. Other solutions can be defined with a generic Montage for lists.

In the graphs of Montages, a list node is represented by a box, which is marked in the right–top corner with the keyword LIST. A second box or circle within the LIST–box represents the single elements in the list. An arrow from a node within a LIST–box corresponds therefore to a family of arrows from all of the elements in that list, whereas arrows from or to the list-box itself have the same semantics as normal arrows.

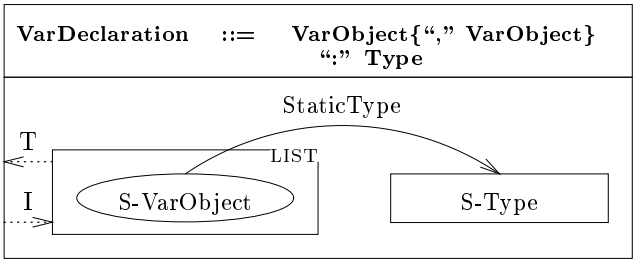


Figure 4: A variable declaration Montage

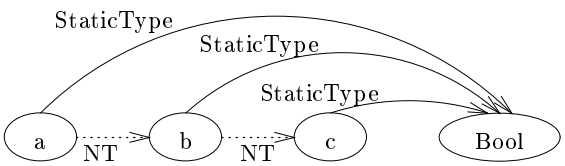


Figure 5: Control/data–flow graph of variable declaration "a, b, c: Bool"

As example we take a Montage for a variable declaration, as pictured in fig. 4. The production rule of that Montage generates a list of VarObject-descendants and a node labeled with Type. The single *StaticType*-arrow in the Montage specifies a family of data flow arrows, one from each variable object to the type-node. Please note that the action of the list node links all variable objects sequentially with an NT arrow. As example we show the generated control/data-flow graph of a variable declaration "a, b, c: Bool" (fig. 5).

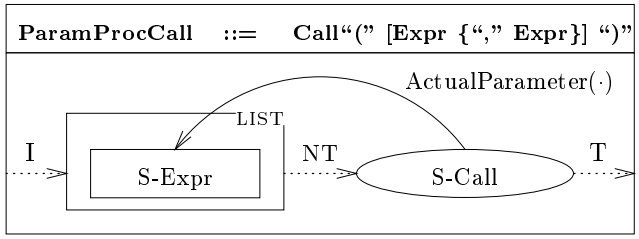


Figure 6: A Montage for a parameterized procedure call

Experience showed that often binary functions which link a leaf with all elements of a list are used. The second argument of such a function is the position in the list. We allow to define this type of binary function graphically by means of a data arrow going into a list box. A typical language construct where this is needed are the actual parameters of a procedure call (see fig. 6, we refer to KP97b for a complete version). The ActualParameter(·) arrow in the Montage defines a binary function ActualParameter which maps a call task c and a position n to the n-th actual parameter of c. As example we show the generated control/data-flow graph of a procedure call "P(x, y, z)" (fig. 7).

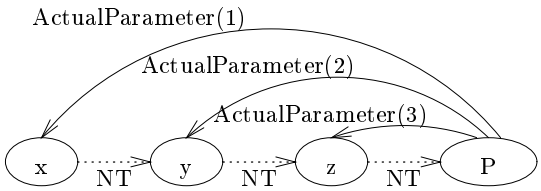


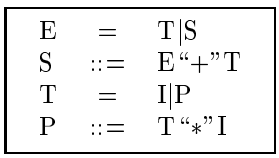
Figure 7: Control/data–flow graph of procedure call "P(x, y, z)"

4 Development Environment for Montages

The development environment for Montages is given by the GEM-MEX tool. It consists of a graphical editor (GEM) providing an easy means to edit the graphical and textual elements of a Montage, and an executable generator (MEX). GEM also contains functionality to generate documents suitable for presenting the Montages. Both, paper and online presentation of the specified Montages are supported by GEM: LaTeX as well as Html versions of the Montages specified can be generated. In order to increase the readability also of the parts of the formalization represented by the textual elements of the Montages, a "literate specification" style is supported by means of a literate programming tool integrated into the system. "Literate specification" means that the Montages text fields may contain references to other parts of the formalization specified out the Montages boxes. This makes the appearance of a Montages specification very much like that of an informal description yet being a formal one.

MEX is a type checker and executable generator for Montages. As mentioned earlier, in a first step MEX uses standard tools (lex and yacc) to construct the abstract syntax tree according to the syntax rules given by the Montages specification. The next step is the generation of code given by class description of the object-oriented programming language Sather. This code consists of the following parts:

- definition of a class hierarchie representing the grammar structure of the specification,



⇒

```
abstract class $E;
class S < $E is
  attr S_E   :$E;
  attr S_PLUS:PLUS;
  attr S_T   :$T
end;
abstract class $T < $E;
class P < $T is
  attr S_T  :$T;
  attr S_AST:AST;
  attr S_I  :I;
end;
```

Figure 8: Generated class structure

- code implementing the conditions and transition rules,
- code representing the ASM for the dynamic semantics given by the text in the bottom–most parts of the Montages, and
- code for debugging the generated executable.

For each grammar rule MEX generates either a concrete or an abstract class depending on whether it is a production rule or a synonym production. The first kind of rule are represented by concrete classes, the latter ones trigger the generation of abstract classes being parent classes of those (concrete or abstract) classes representing the alternatives of the synonym production. Nonterminal appearing on the right-hand-side of a production rule are modeled as attributes of the left-hand-side attribute. Figure 8 shows an example for the generation of classes corresponding to grammar rules.

The consistency check of the Montages is done during the construction of the abstract syntax tree. The data and control edges, the condition and the transition rules are evaluated during a left-to-right tree traversal constructing the initial state for the ASM defining the dynamic semantics.

MEX generates source code that represents the ASM rules of the dynamic semantics. This is done by simple data flow analysis of the updates and introducing auxiliary variables where necessary.

The generated code also contains debugging functionality in order to allow the user to interactively trace the run of the ASM given by the dynamic semantics rules. The debugging functionality includes the animation of nodes within the corresponding Montages in the GEM editor; token nodes and edge are highlighted when they are reached by the control flow.

5 Related Works

Denotational semantics has been regarded as the most promising approach for the semantic description of programming languages. But its problems with the pragmatical side of language design have been discovered already in case studies of the scale of Pascal and C (see for instance [24]). Information hiding, object orientedness and complex name analysis are not covered because of the global visibility of the definition of semantic domains throughout a denotation description. Moreover domain definitions often need to be changed when extending the language with unforeseen constructs, for instance a change from the direct style to the continuation style when adding *gotos* [21]. To cite Abramsky “... once languages with features beyond the purely functional are considered, the appropriateness of modeling programs by functions is increasingly open to question. Neither concurrency nor ‘advanced’ imperative features have been captured denotationally in a fully convincing fashion.” [1]

Other research has been carried out because of the above considerations about pragmatics. In particular, it is worth mentioning Action semantics [21], which is an *initial–algebra semantics* [11], based on Mosses’ unified algebras. Action semantics retained some denotational semantics features, i.e. context–free grammars for defining abstract–syntax trees, and the use of Horn clauses to give inductive definition of compositional

semantic functions. The main semantic entities are actions, which are specified by means of the action notation. To mention Mosses " ... the current structural operational semantics of action notation is not easy to modify; alternative forms of operational semantics, such as evolving–algebra semantics, might be preferable in that respect." [21]

Another universal meta–language for all aspects of programming languages is ASF+SDF [25]. It is an initial–algebra approach and specifies the static and dynamic semantics by means of positive conditional equations. As all the initial–algebra based formalisms they are forced to remain under the expressiveness of the logic of Horn clauses, i.e. positive conditional equations, otherwise the existence of the initial model is not guaranteed and the syntax cannot be mapped in an unambiguous way to the semantics because of non–existence of the universal homomorphism. In this respect, Montages are much more expressive since they make use of the full first–order logic for the static semantics predicates.

An approach with the same ambitious goal are Kahn's Natural Semantics [14] which are directly based on Natural Deduction. For somebody knowing mathematical logic, Natural Semantics are pretty intuitive and we used it for the dynamic semantics of Oberon [16]. Although we succeeded due to the excellent tool support by Centaur [9], the result was much longer and more complex then the Montages counterpart given in [20], since one has to carry around all the state information in the case of Natural Semantics.

Although attribute grammars [15] are not designed to specify all aspects of languages, it's worth noting that the the solution for the static aspects of our approach has some similarities with attribute grammars. Although in certain cases they may be executed very efficiently, we preferred not to use them for the following reasons: using ASMs we have the same formalism for all parts of the specification; and as shown in [22] attribute grammars tend to be very long if applied to real programming languages.

Using ASMs for dynamic semantics, the work in [23] defines a framework comparable to ours. Although it has different aims, namely efficient execution. For the static part, it proposes occurrence algebras which integrate term algebras and context free grammars by providing terms for all nodes of all possible derivation trees. This allows such an approach to define all static aspects of the language in a functional algebraic system, which is supported by the MAX tool. In any case, the additional mathematical machinery must be hidden from the user and Montages might be well suited for that task.

6 Conclusions

In this paper we presented a novel approach to cope with specifications of all aspects of programming languages. Expressive yet intelligible descriptions of language constructs and of complex features together with ease of maintenance were sought. In this respect, the well known ASMs have already attracted attention.

The classical use of ASMs abstracts from the static semantics and assumes the result of a static analysis in order to define the dynamic semantics. The main criticisms against such an approach is that the static analysis is not formalized. An exception is the work on Occam [5]. Unfortunately the solution presented there allows not for the definition of the static semantics. Montages solves the problem using control and data flow graphs and at the same time allows one to give a very compact definition of static semantics as full first–order predicates. At the same time Montages retain the advantages of ASMs.

Experience in scaling up both basic ASMs and Montages for large case studies such as the specification of SQL [10] and Oberon [17, 20] showed some important advantages of Montages with respect to basic ASMs:

- the readability and comprehension of specifications improved drastically since the specification is arranged in capsules of behavior according to the rules of the context–free grammar,
- the maintenance of the whole specification is much easier since the behavior can be easily localized and eventually modified according to requirement changes,
- starting from the static semantics and analysis allows one to have a better comprehension of the dynamic semantics,
- the specification process requires less time since
  - the designer is driven by the structure of the Montages,

– Montages represents a sort of factorization of behavior which can be reused in the definition of other languages in a off–the–shelf fashion or via refinement.

As pointed out in [4] ASMs are deliberately not imposing any particular calculus. The ease of abstraction makes ASMs very suitable for different application domains and for each of them it makes sense to have a different verification system with its own assumptions. Moreover, one is able to do complex mathematical proofs directly in the ASM framework, as shown in [4, 5, 8]. The translation of an ASM model in other formalisms is interesting if tools are available with powerful verification capabilities.

Montages are formal descriptions of programming languages with an higher intelligibility than usual semantics descriptions. This is mainly due to the use of visual elements and the elaborated structuring mechanism.

Future case studies will show whether Montages are not only suited for the specification of programming languages, but for other application domains as well, such as databases, hardware languages and communication protocols.

Acknowledgments We gratefully acknowledge Yuri Gurevich, Egon Börger, Wolf Zimmermann, Jim Huggins, David Espinosa, and Daniel Schweizer for their constructive comments on early drafts of the paper. Thanks goes to Richard Waldinger and Chuck Wallace who helped us with the writing; Richard proposed the name Montages.

References

[1] S. Abramsky. Semantics of Interaction. In *Trees in Algebra and Programming – CAAP'96, 21st Int. Coll.*, volume 1059 of *LNCS*, page 1. Springer Verlag, 1996.

[2] M. Anlauff. The semantics of the object–oriented programming language sather. Technical report, International Computer Science Institute, Berkeley, 1997. In preparation.

[3] M. Anlauff, P.W. Kutter, A. Pierantonio, D. Rosenzweit, L. Thiele, and W. Zimmermann. Compiler construction with montages. submitted for publication, 1997.

[4] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering. In *SOFTSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236 – 271. Springer Verlag, 1995.

[5] E. Börger and I. Durdanović. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, 39(1):52 – 92, 1996.

[6] E. Börger, U. Gläser, and W. Mueller. Formal Definition of an Abstract VHDL'93 Simulator by EA–machines. In *Semantics of VHDL*, volume 307 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, 1995.

[7] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. *Science of Computer Programming*, 1994.

[8] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness*, chapter 2, pages 20 – 90. Series in Computer Science and Artificial Intelligence. Elsevier Science B.V.North Holland, 1995.

[9] P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. Technical Report 777, INRIA, Sophia Antipolis, 1987.

[10] B. DiFranco. Semantica Statica e Dinamica di SQL mediante i Montaggi. Master's thesis, Universitá di L'Aquila, 1997. in italian.

[11] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebras semantics and continuous algebras. *J.ACM*, (24):68 – 95, 1977.

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

[13] Y. Gurevich and J.K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274 – 308. Springer Verlag, 1993.

[14] G. Kahn. Natural Semantics. In *Proceedings of the Symp. on Theoretical Aspects of Computer Science, Passau, Germany*, 1987.

[15] D.E. Knuth. Semantics of Context–Free Languages. *Math. Systems Theory*, 2(2):127 – 146, 1968.

[16] P.W. Kutter. Executable Specification of Oberon Using Natural Semantics. Term Work, ETH Zürich, implementation on the Centaur System [9], 1996.

[17] P.W. Kutter. Dynamic Semantics of the Programming Language Oberon. TIK-Report 27, ETH Zürich, 1997.

[18] P.W. Kutter and A. Pierantonio. Montages: Unified static and dynamic semantics of programming languages. Technical Report 118, Dip. Matematica Pura ed Applicata, Universitá di L'Aquila, July 1996.

[19] P.W. Kutter and A. Pierantonio. Montages Specifications of Realistic Programming Languages. *Springer Journal of Universal Computer Science*, 3(5):416–442, 1997.

[20] P.W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Springer Journal of Universal Computer Science*, 3(5):443–503, 1997.

[21] P. Mosses. Theory and Practice of Action Semantics. In *MFCS'96, 21st International Symposium*, volume 1113 of *LNCS*, pages 37 – 61. Springer Verlag, 1996.

[22] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, ETH Zürich, 1989.

[23] A. Poetzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 233 – 247. Springer–Verlag, 1994.

[24] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, 1986.

[25] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping – An Algebraic Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

[26] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131 – 164. Oxford University Press, 1994.