Generating an Action Notation Environment from Montages Descriptions

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

[1] GMD FIRST, D-10000 Berlin
ma@first.gmd.de
[2] Federal Institute of Technology, CH-8092 Zürich
{kutter, alfonso, thiele}@tik.ee.ethz.ch

Abstract. In the present paper, a methodology is presented which enables the implementation of the Action Notation formalism based on a formal and modular specification. As a result, an interpreter and debugger is automatically generated which allows the visualization of an Action Notation program execution and the inspection of all semantic identities in terms of the given formal specification.
These results are based on several new concepts. At first, a formal description of Action Notation is provided by means of Montages. Montages are a semi-visual formalism for the specification of syntax and semantics of programming languages. Moreover, the structuring of Action Notation via facets is refined and used to define a new specification architecture that ensures the required modularity. The tool support for Montages (Gem-Mex) automatically generates a prototypical implementation from the language's Montages specification.

1 Introduction

Action Semantics [Mos92,Wat91] allows the description of large, realistic programming languages like Standard ML [Wat99] or Java [BW99]. The main reason is that the addition of new constructs to a described language does not require reformulation of the already-given description. According to [Mos98b,Mos98a] we call this property *modularity*. Nevertheless, experience showed that despite of modularity and other pragmatic qualities of Action Semantics, tool support is crucial when large languages are specified.

In Action Semantics, as in denotational semantics, semantic functions map the abstract syntax of the described language to semantic entities. Here, however, the semantic entities are so-called actions, which are specified in Action Notation (AN), rather than higher-order functions expressed in lambda-notation. In this paper, we describe how to obtain an environment for the execution and inspection of AN descriptions. Implementing such an environment by hand presents some disadvantages. For example, it is very difficult to maintain the consistency of such a system if the definition of AN evolves over time. Moreover, the implementation is not accessible for inspection to the user as there is no direct link (neither formally nor visually) between the specification of AN and the execution of a AN program.

Generating the environment from an executable specification solves the mentioned problems. For instance, the ASD tool [vDM96] is generated from an ASF+SDF description of AN. The resulting environment consists of a structural editor, an interpreter, and semantics inspection support on the abstraction level of term rewriting. ASF+SDF, as well as Denotational Semantics, SOS, and Natural Semantics present the disadvantage of not being inherently modular [Mos98b,Mos98a].

If AN is extended, as for instance in [MM93], a modular semantics is necessary. Otherwise, for each extension of AN re-proving all the laws of AN is required. An approach to achieve modularity for this purpose has been presented in [Wan97].

Independent of modular proofs, it is useful to have a modular tool support that allows to quickly adopt the newest theoretical results and extend the existing implementations accordingly. In addition, it may be useful to test the practicability of an extension before doing all the work of re-proving the laws of AN. Another advantage of basing a tool on a modular description of AN is the possibility to develop, test, and validate the specification in small pieces. Once their correct behavior is assessed in isolation, one is able to investigate the interaction between modules, while putting everything together. Such a development process structures the resulting implementation and makes it more transparent and accessible for the user.

This work illustrates the first tool environment for AN based on and automatically generated from a modular specification. AN descriptions can be visualized, debugged, and interpreted in terms of the specification. This corresponds to *origin tracking*, i.e. to the direct and consistent relation of the implementation with the underlying specification.

The described results are based on the following techniques:

- We use a a semi-visual formalism for the specification of syntax and semantics of programming languages, called Montages [KP97a,AKP98]. Similar to Action Semantics, Montages aim at being a pragmatic framework for language engineering. Montages are based on context-free grammars (EBNF), finite state machines for visual control flow and Abstract State Machines (ASMs) [Gur88,Gur95] for the dynamic semantics. These concepts are informally described in Section 2.
- In Section 3, we introduce an architecture which is based on a refined partition of AN in facets. The imperative and parts of the basic facet are formalized, and as an example of the modularity it is shown how they can be combined. The given parts show and explain the techniques used in the complete Montages of Action Notation [AKP97c], covering the remaining parts of the basic facet as well as the declarative, and reflective facets. The communicative facet has not yet been incorporated.

– Finally, Section 4 gives an impression of the generated AN environment and shows how it serves as a platform for conducting empirical validation of the design decisions using origin tracking.

2 Montages

In this section, the methodology is introduced on which the results of the paper are based. Montages is a formalism for the specification of programming languages. We will describe informally only those features which are relevant for the specification of Action Notation. The complete specification of Montages is available in [KP97a,AKP98].

The aim of Montages is to document formally the decisions taken during the design process of realistic programming languages. Syntax, static and dynamic semantics are given in a uniform and coherent way by means of semi-visual descriptions. The static aspects of a language are diagrammatic descriptions of control flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. The intended use of the tool Gem-Mex is, on one hand to allow the designer to 'debug' her/his semantics descriptions by empirical testing of whether the intended decisions have been properly formalized; on the other hand, to automatically generate a correct (prototype) implementation of programming languages from the description, including visualization and debugging facilities.

The departure point for our work has been the formal specification of the C language [GH93][1], which showed how the state-based formalism Abstract State Machines [Gur88,Gur95,Hug] (ASMs), formerly called Evolving Algebras, is well-suited for the formal description of the dynamic behavior of full-blown practical languages. In essence, ASMs constitute a 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. The model presented in [GH93] describes the dynamic semantics of the C language by presuming on an explicit representation of *control and data flow as a graph* (CDG). This represents a major limitation for such a model, since the control and data flow graph is a crucial part of the specification. Therefore, we developed Montages which extend the approach in [GH93] by introducing a mapping which describes how to obtain the control and data flow graph starting from the abstract syntax tree.

The formulation of Montages [KP97a] was strongly influenced by some case studies where the Oberon language [KH95,KP97b] has been specified. Oberon is an object-oriented language that is used for the implementation of compilers, operating systems [WG92], various applications, and teaching [RW92]. Montages have been used also in other case studies, such as the specification of the

[1] Historically the C case-study was preceded and paralleled by work on Pascal [Gur88], Modula2, Prolog, and Occam, see [BH98] for a commented bibliography on ASM case studies.

Java [Wal97] language, the front-end for correct compiler construction [HLT98], and the design and prototyping of a domain-specific language in an industrial context [KST98]. The logical/algebraic characterization of an extended Montages formalism has been presented in [AKP98]. The tool Gem-Mex has been completely re-implemented with respect to the initial prototype [AKP97a]. Complete references, documentation and tools can be obtained via http://www.tik.ee.ethz.ch/∼montages/.

The experience showed that the underlying model for the dynamic semantics, namely the specification of a control flow graph including conditional control flow and data flow arrows and its close relationship to the well known concept of *Finite State Machines*, shortens the learning curve considerably. It confers to the formalism enhanced pragmatic qualities, such as writability, extensibility, readability, and, in general, ease of maintenance.

In our formalism, the specification of a language consists of several components. As depicted in Fig. 1, the language specification is partitioned into three parts.

1. The EBNF production rules are used for the context-free syntax of the specified language L, and they allow to generate a parser for programs of L. Furthermore, the rules define in a canonical way the signature of abstract syntax trees (ASTs) and how the parsed programs are mapped into an AST. Section 2.1 contains the details of this mapping. In Fig. 1 the dotted arrow from the EBNF rules visualizes that this information is provided from the Montage language specification.
2. The next part of the specification is given using the *Montage Visual Language* (MVL). MVL has been explicitly devised to extend EBNF rules to finite state machines (FSM). A MVL description associated to an EBNF rule defines basically a *local* finite state machine and contains information how this FSM is plugged into the *global* FSM via an inductive decoration of the abstract syntax trees. To this end, each node is decorated with a copy of the finite state machine fragment given by its Montage. The reference to descendents in the AST defines an inductive construction of a global structured FSM. In Section 2.2 we define how this construction works exactly.
3. Finally, any node in the FSM may be associated with an Abstract State Machine (ASM) rule. This *action rule* is fired when the node becomes the current state of the FSM. As shown in Fig. 1, the specification of these rules is the third part of a Montages specification. The underlying abstract state machine formalism is shortly described in Section 2.3.

The complete language specification is structured in specification modules, called Montages. Each Montage is a "BNF-extension-to-semantics" in the sense that it specifies the context-free grammar rule (by means of EBNF), the (local) finite state machine (by means of MVL), and the dynamic semantics of the construct (by means of ASMs). The special form of EBNF rules allowed in a specification and the definition of Montages lead to the fact that each node in the abstract syntax tree belongs exactly to one Montage.

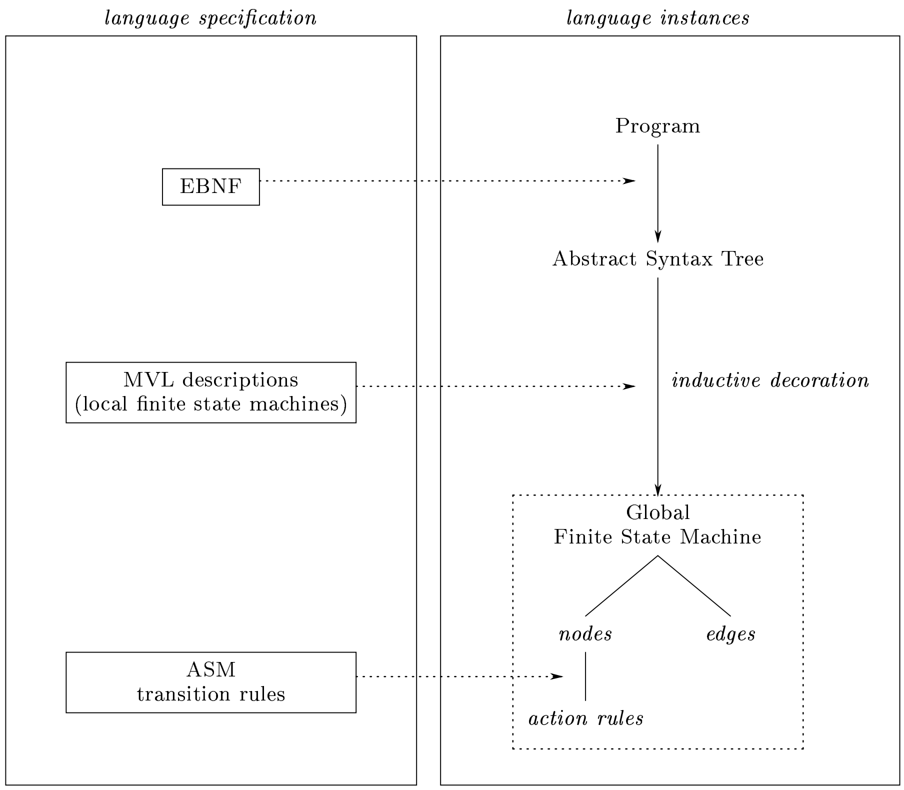


Fig. 1. Relationship between language specification and instances.

As an example the Montage for a nonterminal with name Sum is shown in Fig. 2. The topmost parts of this Montages is the production rule defining the context-free syntax. The remaining part defines static aspects of the construct given by means of an MVL description. Additionally, the Montage contains an action rule, which is evaluated after the two operands, i.e. when the control reaches the sum node.

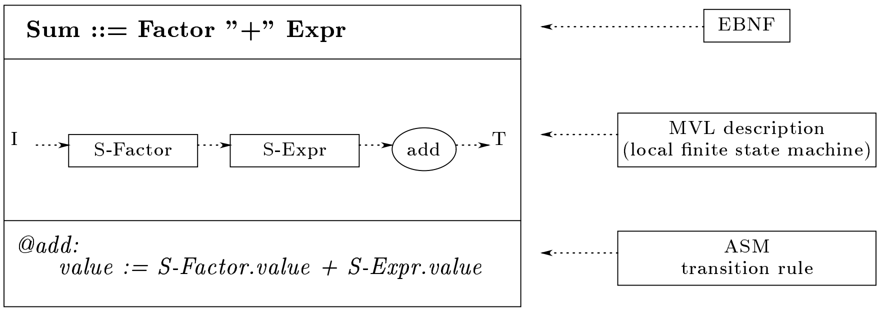


Fig. 2. Montage components.

The definition of Montages usually contains a fourth section which is devoted to the specification of the static semantics. As we are not using this property in the current paper, it will not be described. Future work will use this feature to formalize the elaborated static-semantics of AN, following the work in [Ørb94].

2.1 From Syntax to AST

In this section, the first step in Fig. 1 is described. As a result of this step we get the abstract syntax tree of the specified program. But we also compose the Montages corresponding to the different constructs of the language. This composition of the partial specifications is done based on the structure of the AST.

EBNF rules The syntax of the specified language is given by the collection of all EBNF rules. Without loss of generality, we assume that the rules are given in one of the two following forms:

$$A ::= B\ C\ D \tag{1}$$

$$E \ = \ F \mid G \mid H \tag{2}$$

The first form defines that A has the components B, C, and D whereas the second form defines that E is one of the alternatives F, G, or H. Rules of the first form are called *characteristic productions* and rules of the second form are called *synonym productions*. We guarantee that each non-terminal symbol appears in exactly one rule as the left-hand-side. Non-terminal symbols appearing on the left of the first form of rules are called *characteristic symbols* and those appearing on the left of synonym productions are called *synonym symbols*.

Composition of Montages Each characteristic symbol and certain terminal symbols define a *Montage*. A Montage is considered to be a *class*[2] whose instances are associated to the corresponding nodes in the abstract syntax tree. Symbols in the right-hand side of a characteristic EBNF rule are called *(direct) components* of the Montage, and symbols which are reachable as components of components are called *indirect components*. In order to access descendants of a given node in the abstract syntax tree, statically defined attributes are provided. Such attributes are called *selectors* and they are unambiguously defined by the EBNF rule. In the above given rule, the B, C, and D components of an A instance can be retrieved by the selectors S-B, S-C, and S-D. In Fig. 3 a possible representation of the A-Montage as class and an abstract syntax tree (AST) with two instances of A and their components are depicted.

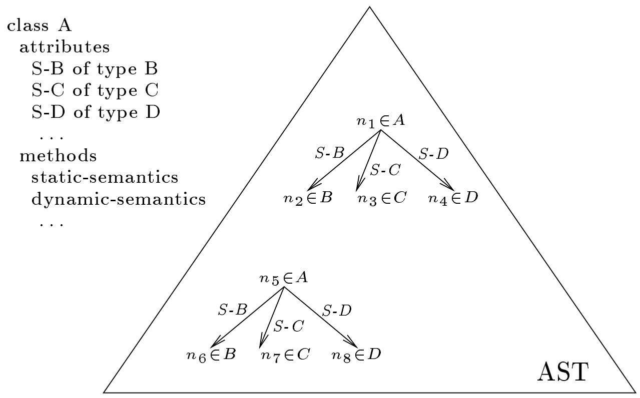


Fig. 3. Montage class A, instances in the AST, selectors S-B, S-C, S-D

Synonym rules introduce *synonym classes* and define subtype relations. The symbols on the right-hand-side of a synonym rule can be further synonym classes or Montage classes. Each class on the right-hand-side is a subtype of the introduced synonym class. Thus, each instance of one of the classes on the right-hand side is an instance of the synonym class on the left-hand-side, e.g. in the given example, all F-, G-, and H-instances are E-instances as well. In the AST, each

[2] In this context we consider class to be a special kind of abstract data type, having attributes and methods (actions) and, most important for us, where the notion of sub-typing and inheritance are predefined in the usual way.

inner node is an an instance of arbitrarily many (possibly zero) synonym classes and of exactly one Montage.

Terminals, e.g. identifiers or numbers, do not correspond to Montages. The micro-syntax can be accessed using an attribute *Name* from the corresponding leaf node. The described treatment of characteristic and synonym productions allows for an automatic generation of AST from the concrete syntax given by EBNF, see also the work in [Ode89].

Induced structures Inside a Montage class, the term *self* denotes the current instance of the class. Using the selectors, and knowledge about the AST, we can build paths w.r.t. to self. For instance, the path *self.S-B.S-H.S-J* denotes a node of class J, which can be reached by following the selectors S-B, S-H, and then S-J, see Fig. 4. The use of such a path in a Montage definition imposes a number of constraints on the other EBNF rules of the language. The example *self.S-B.S-H.S-J* requires that there is a B component in the Montage containing the path. Further, every subtype of B must have an H component, and every subtype of H must have an J component. In other words, the path *self.S-B.S-H.S-J* must exist in all possible ASTs.

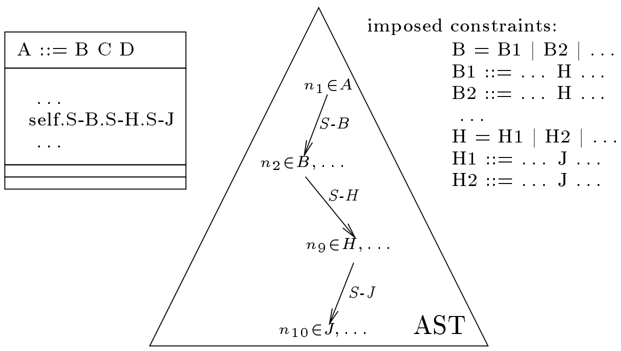


Fig. 4. Montage A using path self.S-B.S-H.S-J, situation in AST, and constraints on EBNF rules of B, H.

Example As a running example we give a small language $\mathcal{S}$. The expressions in this language potentially have side effects and must be evaluated from left to right. The atomic factors are integer constants and variables of type integer. The start symbol of the EBNF is Expr, and the remaining rules are

Expr	=	Sum | Factor
Sum	::=	Factor "+" Expr
Factor	=	Variable | Constant
Variable	::=	Ident

Constant ::= Digits

The following term is an $\mathcal{S}$-program:

```
2 + x + 1
```

As a result of the generation of the AST and the composition of the individual Montages shown in Fig. 2 and Fig. 5 we obtain the structure represented in Fig. 6.

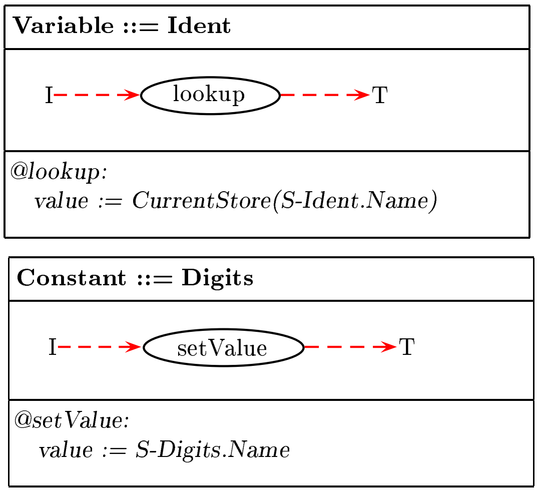


Fig. 5. The Montages for the language $\mathcal{S}$.

In particular, the nodes from 1 to 8 represent instances of the Montage classes and the edges point to the successors of a particular node. The edges are labeled with the selector functions which can be used in the Montage corresponding to the source node to access the Montage corresponding to the target node. The nodes themselves show the class hierarchy starting from the synonym class and ending with the Montage class. The leaf nodes contain the definition of the attribute Name, i.e. the micro-syntax.

2.2 From AST to Control Flow Graphs

According to Fig. 1, the next step in building the data structure for the dynamic execution is the inductive decoration of the AST with a number of finite state machines. Again, this process is described rather informally here.

As we have seen in Fig. 2 and Fig. 5, the second part of a Montage contains the necessary specifications given in form of the *Montage Visual Language* (MVL). Two kinds of information are represented here: (a) the local state machine to be associated to the node of the AST and (b) information on the embedding of this

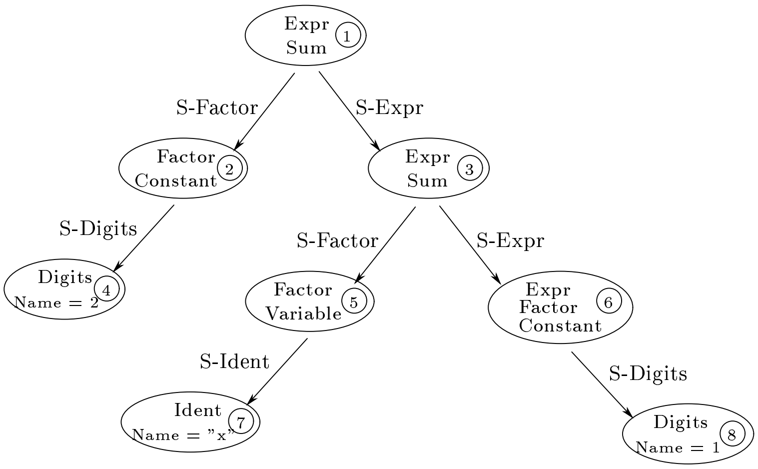


Fig. 6. The abstract syntax tree and composition of Montages for `2 + x + 1`

local state machine. Using our running example, Fig. 7 just represents the MVL sections of the Montages as they are associated to the corresponding nodes of the abstract syntax tree. The hierarchical state transition graph resulting from

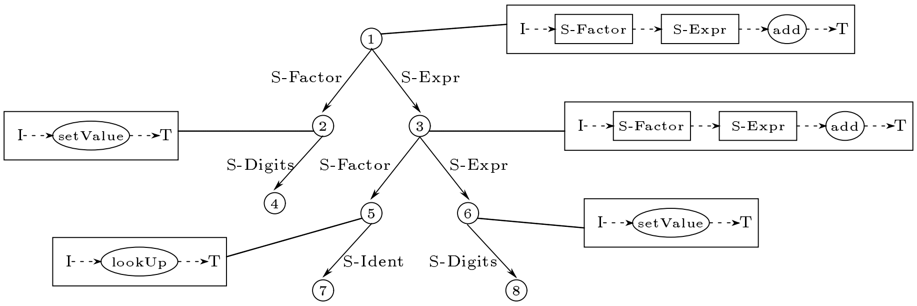


Fig. 7. The finite state machines belonging to the nodes.

the inductive decoration is shown in Fig. 8 for the running example.

Montage Visual Language Now, the elements of the MVL and their semantics can be described as follows:

– There are two kinds of nodes. The oval nodes represent states in the generated finite state machine. These states are associated to the AST node corresponding to the Montages. The oval nodes are labeled with an attribute. It

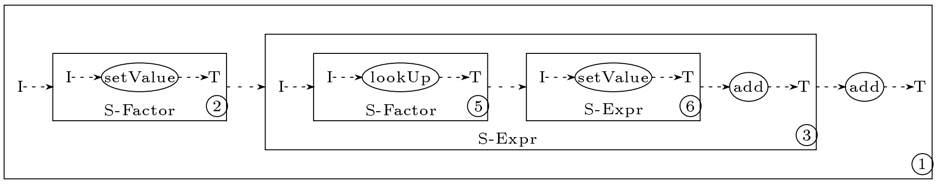


Fig. 8. The constructed hierarchical finite state machine.

serves to identify the state, for example if it is the target of a state transition or if it points to a dynamic action rule.

– The rectangular nodes or boxes represent symbols in the right hand side of the EBNF rule and are called direct components of a Montages, see Section 2.1. They are labeled with the corresponding selector function. Boxes may contain other boxes which represent indirect components. This way, paths in the AST are represented graphically.
– The dotted arrows are called control arrows. They correspond to edges in the hierarchical state transition graph of the generated finite state machine. Their source or target can be any box or oval. In addition, their source or target can be either the symbol I (I stands for initial) or T (T stands for terminal), respectively. In a Montage, at most one symbol of each, I and T, is allowed. If the I symbol is omitted, the states of the Montage can only be reached using a jump, if the T symbol is omitted, the Montages can only be left using a jump.
– As in other state machine formalisms (such as Harel's StateCharts), predicates can be associated to control arrows. They are simply terms in the underlying ASM formalism and are evaluated after executing the action rule associated to the source node. Predicates must not be associated to control arrows with source I.
– There are additional notations not used in this paper — for example data flow edges representing the mutual access of data between Montages and box structures representing lists in an effective way. Moreover, in this section of a Montage, one may specify further action rules to be performed in the static analysis phase, for example building up data structures necessary for the static and dynamic semantics.

It remains to show how the hierarchical finite state machine, for example Fig. 8 is built and how its dynamic semantic is defined.

Hierarchical FSM Building the hierarchical FSM is particularly simple. The boxes in the MVL are references to the corresponding local state transition graphs. Remember that nested boxes correspond to paths in the AST. Therefore, there are references to children only, i.e. to other state transition graphs along the edges of the AST. After resolving the references, a representation as in Fig. 8 is obtained.

Dynamic Semantics After the static analysis phase action rules are executed which define the dynamic semantics of the language.

- States of the finite state machines are visited sequentially.
- The action rule associated to a visited state is executed. The specification of these actions is based on the ASM formalism and specified in Section 2.3.
- The control is passed to the next state along a control arrow whose predicate evaluates to true. The control predicate, i.e. a term in the ASM formalism, is evaluated after executing the action associated to the source node.
  If there is more than one possible next state, the system behaves like a nondeterministic FSM. Up to now we did not use nondeterministic FSMs.
- If the target of a control arrow is a T, then a control arrow leaving the corresponding box in the enclosing parent state machine is followed. The term parent refers to the partial ordering of local state machines as imposed by the AST.
- If the target of a control arrow is a box, the corresponding local state machine corresponding to it is entered via the symbol I.

More formally, the arrows from I and to the T symbols define two unary functions, *Initial* and *Terminal* denoting for each node in the AST the first, respectively last state that is visited. According to the above description, the inductive definition of these functions is given as follows.

For each state s in the finite state machines,

$$s.Initial = s \tag{3}$$

$$s.Terminal = s \tag{4}$$

and for each instance n of a Montage N whose MVL-graph has an edge from I to a component denoted by path *tgt*,

$$n.Initial = n.tgt.Initial$$

and for each instance m of a Montage M whose MVL-graph has an edge from a component denoted by path *src* to T,

$$m.Terminal = m.src.Terminal$$

Using these definitions, the structured finite state machine can be flattened. The arrows of the flat finite state machine are given by the following equations defining the relation *ControlArrow*. For each instance n of a Montage N and each edge e in the MVL-graph of N,

$$ControlArrow(n.src.Terminal, n.tgt.Initial)) = true$$

where *src* is the path of the source of e and *tgt* is the path of the target of e.

Applying these definitions to the running example results in the flat state machine of Fig. 9. In the same figure the dotted lines denote the relation of a state to its corresponding Montage, which is accessible as *self*. Using the Montages

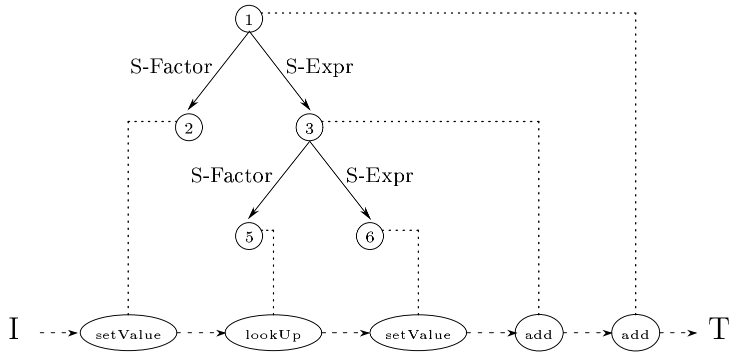


Fig. 9. The flat finite state machine and its relation to the AST.

shown in Figs. 2 and 5 and their action rules, we can track how the ASM rule associated with the *add* states can access the AST-nodes of its left and right arguments as *self.S-Factor* and *self.S-Expr*. The results of calculations performed by the actions are stored in the additional attributes *value*. The *add* action accesses the values of its arguments using the selectors, and defines its own *value* field to be the sum of the arguments. Assuming that *CurrentStore* maps x to 4, the execution of the flat or structured finite state machine sets the value of node two to the constant 2, sets the value of node five to the current store at x, sets the value of node six to 1, sets the value of node three to the sum of 4 and 1, and finally sets the value of node one to the sum of 2 and 5.

Please note that *all* the informally described concepts have been formalized using ASM notation. Even in the implementation, a Montages specification is at first transformed into static functions and ASM rules which are then executed by an ASM simulation engine. Therefore, the specification of control may also be provided in the dynamic semantics of a Montages. We use this possibility in the specification of AN in Section 3 via the function *JumpTo*. The underlying ASM formalism is described in the next section.

2.3 Dynamic Semantics by means of ASM rules

Basic ASM formalism The fundamental concept in ASMs is the *object*, which we consider an atomic entity. We call the set of all objects the *super-universe* $\mathfrak{U}$. In any ASM, $\mathfrak{U}$ contains the distinct objects *true, false* and *undef*. Additional examples for objects in the Montages context are the nodes of the abstract syntax tree, and the states of the finite state machines.

The *state* λ of the system is given as the mapping of a number of function symbols, the *signature* Σ, to actual functions. For short we write f_λ for the function interpreting the symbol f in state λ. We write $\mathfrak{S}(\Sigma)$ for the set of all Σ–states.

The functions interpreting the symbols need *not* be strict with respect to *undef.* In particular, the equality symbol is defined such that *undef* = *undef* evaluates to *true*. In our framework Σ contains a unary function for each attribute of a Montage. Examples are the selector-attributes, the attributes denoting the states, and specific to our running example the attribute *value*. Subsets[3] of $\mathfrak{U}$ are modeled by functions from $\mathfrak{U}$ to $\{true, false\}$. Such a function delivers *true* for all members of a set (instances of a type), and *false* otherwise. The set or type consisting of *true* and *false* is called *Boolean*. For each montage M, Σ contains the set M of M-instances. In the examples we have seen the sets *Expr*, *Factor*, *Sum*, *Variable*, *Constant*, *Ident*, and *Digits*. These sets can be used to characterize the types of functions, for instance

$$\mathit{value} : \mathit{Sum} \cup \mathit{Variable} \cup \mathit{Constant} \rightarrow \mathit{Integer}$$

A state transition changes these functions pointwise, by so-called *updates*. An update is a triple

$$(f, (o_1, \ldots, o_n), o_0)$$

where f is an n-ary function symbol in Σ, and $o_0, \ldots, o_n \in \mathfrak{U}$. Intuitively, *firing* this update at a state λ changes the function associated with f in λ at the point $(o_1, \ldots, o_n)$ to the value o_0, leaving the rest of the function unchanged.

Examples for rules are the actions *add*, *lookUp*, and *setValue* in Figs. 2 and 5. For instance the rule in Fig. 2

value := S-Factor.value + S-Expr.value

abbreviates

self.value := self.S-Factor.value + self.S-Expr.value

The meaning is that the attribute *value* of *self* is updated to the sum of the values of the left and right argument, which are accessed by means of the selector attributes *S-Factor* and *S-Expr*. Not only the semantics of the action rules, but the complete semantics of the finite state machine plus their construction is given by an ASM rule.

The formal semantics of a rule R in a state λ is given by a deterministic denotation *Upd(R, λ)* being a set α of updates [Gur97]. Given an update set α and a state λ, firing α at λ results in the successor state $\lambda' = \alpha(\lambda)$. We then have the following relation between f_λ and the new functions $f_{\lambda'}$:

$$f_{\lambda'}(o_1, \ldots, o_n) \longmapsto \begin{cases} o_0 & if \quad (f, (o_1, \ldots, o_n), o_0) \in \alpha \\ f_{\lambda'}(o_1, \ldots, o_n) & otherwise \end{cases} \tag{5}$$

[3] traditionally ASM literature speaks about universes

Update sets are defined by transition rules. The basic update denotes one update of a function at some point. The new values and the point are given by terms over the signature. Rules can be composed in a parallel fashion, such that all updates are executed at once. Conditional execution of a rule fires only in certain cases. The do-forall rule allows to fire the same rule for all objects satisfying some condition. Finally the extend-rule allows to introduce reserve-objects, that have not been used before. The last rule is typically used to create new objects of some class.

Formally, a transition rule R is built up recursively by the following constructions. The corresponding denotations are given. Let $eval_\lambda$ be the usual term evaluation over the state λ.

(Upd 1: basic update) if $R = \boxed{f(t_1, \ldots, t_n) := t_0}$
where $t_0, \ldots, t_n$ are terms over Σ,
then $Upd(R, \lambda) = \{(f, (eval_\lambda(t_1), \ldots, eval_\lambda(t_n)), eval_\lambda(t_0))\}$

(Upd 2: parallel composition) if $R = \boxed{R_1 \ldots R_m}$

then $Upd(R, \lambda) = \bigcup_{i \in \{1,\ldots,m\}} Upd(R_i, \lambda)$

(Upd 3: conditional rules) if $R = \boxed{\textbf{if } t \textbf{ then } R_{true} \textbf{ else } R_{false} \textbf{ endif}}$

then $Upd(R, \lambda) = \begin{cases} Upd(R_{true}, \lambda) & if \quad eval_\lambda(t) = true \\ Upd(R_{false}, \lambda) & otherwise \end{cases}$

(Upd 4: do forall) if $R = \boxed{\textbf{do forall } e : t \quad R' \textbf{ enddo}}$

then $Upd(R, \lambda) = \bigcup_{o : eval_{(\lambda \cup e \mapsto o)}(t)} Upd(R, \lambda \cup e \mapsto o)$

(Upd 5: extend) if $R = \boxed{\textbf{extend } E \textbf{ with } e \quad R' \textbf{ endextend}}$

then $Upd(R, \lambda) = Upd(R, \lambda \cup e \mapsto o)$,
where o is a completely new allocated element.[4]

In addition to the existing ASM concepts we use a number of structuring concepts well known from object oriented and functional programming. The formal semantics of these concepts are given in terms of basic ASMs.

Classes and Methods In Section 2.1 we introduced classes whose instances are the nodes in the AST. As already noted the instances of a class S are modeled by a universe S in the signature and attributes of the class are unary functions, whose domain are the instances of the class. In Section 3 this technique will be used to present several ADTs which encapsulate basic concepts of Action

[4] See [Gur97] for a formalization of "completely new".

Notation. Experience showed that in this way, the semantics and tools support of Montages can be freely extended in new areas.

In addition, classes allow multiple inheritance, and recursive, dynamically bound methods. The sub-typing of classes and synonym classes mentioned in Section 2 is an application of inheritance. The method calls have a value parameter semantics, and are used at several places in Section 3.

Constructors The concept of terms built up by constructors can be mapped to the ASM approach as follows: each of the function names may be marked as *constructive*, expressing that constructor functions are 1-1 and total.

Let $\Sigma_c \subseteq \Sigma$ be the set of all constructive function symbols. If $f \in \Sigma_c$, with arity n, then the following conditions hold for all states A of the ASM:

(i) $\forall t_1, \ldots, t_n, eval_A(t_i) \neq undef, 1 \leq i \leq n \quad : \quad f(t_1, \ldots, t_n) \neq undef$
(ii) $\forall g \in \Sigma_c$, arity of g is m; $\;t_1, \ldots, t_n, eval_A(t_i) \neq undef \quad :$
$$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m) \Leftrightarrow$$
$$f = g \wedge n = m$$
$$\wedge eval_A(t_i) = eval_A(s_i), 1 \leq i \leq n$$

where $eval_A(t)$ stands for the evaluation of term t in state A of the ASM. Informally speaking that means that each constructive function is (i) total with respect to $\mathfrak{U}$ and (ii) injective. If $f \in \Sigma_c$, then f is called a *constructor*, and the terms $f(t_1, \ldots, t_n)$ are called *constructor terms*. In the following, we use the constructor term t as a synonym for its unique value $eval_A(t)$.

For instance, the stack constructors *empty* and *push* can now be defined as follows:

```
constructor empty, push(_,_)
```

In addition, it is possible to define universes that are built up by constructor terms. For example, defining a universe *Stack* as

```
universe Stack = { empty, push(_,_) }
```

introduces the constructive functions *empty* and *push*, $eval_A(empty) \in Stack$ and $eval_A(push(t_1, t_2)) \in Stack$, for all terms t_1, t_2.

If a constructor definition is syntactically contained in a class definition C, then the constructor terms are put into the corresponding universe C. For example, the following constructor definitions are equivalent to the previous ones:

```
class Stack is
  constructors empty, push(_,_)
   ...
```

Pattern Matching Based on the concept of constructor terms, pattern matching functionality is provided. A *pattern matching equation* is a conditional term of the form $t_1 =\tilde{}\, t_2$. The *pattern term* t_2 may contain any numbers of *pattern variables* of the form "&x". This kind of equations perform the pattern matching operation well-known from the functional programming context.

Consider the following equational specification

$$\begin{aligned} x.push(y).pop &= x \\ x.push(y).top &= y \\ empty.top &= undef \\ empty.pop &= empty \end{aligned}$$

then the ASM translation is given by the following.

```
class Stack is
  constructor empty
  constructor push(_,_)

  method top is
    if self =~ push(&s, &d) then
      top_result := &d
    else
      top_result := undef

  method pop is
    if self =~ push(&s, &d) then
      pop_result := &s
    else
      pop_result := empty
```

In Section 3 the class Stack is used to simulate structure-based control flow concepts, which do not correspond to the finite state machine based model in Montages.

The ADT Stack exemplified a functional specification style which in turn can be freely mixed with the typical imperative ASM techniques based on updates of functions, as proposed in [Ode98]. In Section 3 many examples for this technique are shown, including a refinement from a functional specification of an ADT Map into a more imperative specification. In [Ode98] it is show how such refinements can be proved to be correct.

The tool support of Montages is based on the ASM compiler Aslan [Anl]. Aslan is a conservative and faithful implementation of ASM as defined by Gurevich in [Gur95] and [Gur97]. Furthermore Aslan presents some extensions, including classes and constructors, whose semantics has been formalized.

3 Action Notation Specification

In the Montages specification of the Action Notation-formalism (MAN) the exploitation of state-based features along with structure-based ones allows an architecture specification to reduce the interdependence between modules. This architecture follows and refines the partition of Action Notation (AN) into facets. In Section 3.1 an overview on this architecture is given. Section 3.2 shows how we implemented data notation(DN), and how terms are accessed and evaluated. The structure based features are illustrated in the description of the basic facet (Section 3.3). The full power of the state based features are shown in Section 3.4 where we give the specification of the main aspects of the imperative facet.

3.1 Architecture

The formal description of the AN consists of an interconnection of specification modules. Each module contains some local information, which possibly makes use of the behavior described in some other module. The specification architecture is illustrated in Fig. 10. The decomposition was done to separate the different

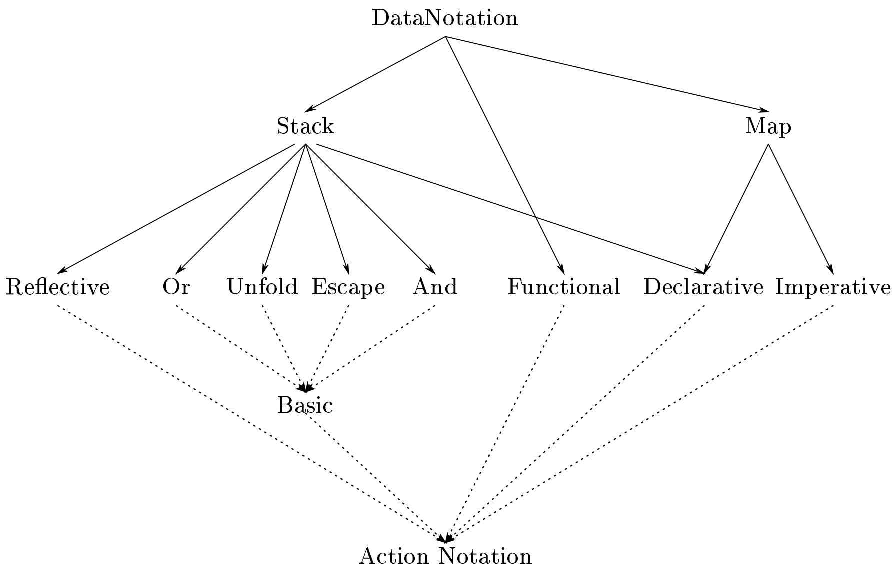


Fig. 10. Specification architecture

concerns involved in the design. Solid arrows in the graph denote import relation. The module *DataNotation*, described in Section 3.2, is used in all other modules.

The module *Stack* consists of the ADT Stack together with some functions described in Section 3.3. Stack is used both by the module *Reflective* and the four modules *Or*, *Unfold*, *Escape*, and *And* which are partitioning the actions of the basic facet into four modules. Each of these modules works in isolation, e.g. it refers to no part of other facets. In Section 3.3 the Or and Unfold modules are introduced in detail.

The module *Map* consists mainly of the definition of ADT Map. This ADT is used both in the *Imperative* facet and the *Declarative* facet. In Section 3.4 Map is informally introduced and used to explain the imperative facet. Later in Section 3.5 a simple and a refined specification of Map are shown.

The *Declarative* facet combines the techniques introduced in the description of the basic and the imperative facet. The declarative as well as the reflective facet are not further described in this text. The *communicative* facet is not yet included in our tool.

The described modules can be arbitrarily combined. The composition along the dotted arrows can always be reduced to expressions of the following form as depicted in figure 11.

$$M = M_1 \oplus_{M_0} M_2 \tag{6}$$

This means that the module M is obtained by the union of M_1 and M_2, which shares the module M_0. We call it, *composition* of M_1 and M_2 via M_0.[5]

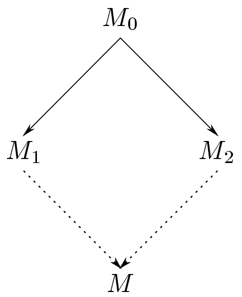


Fig. 11. A simple module decomposition.

For instance, if we consider the specification *Basic*, it is obtained by

$$\mathit{Basic} = \mathit{Or} \oplus_{\mathit{Stack}} \mathit{Unfold} \oplus_{\mathit{Stack}} \mathit{Escape} \oplus_{\mathit{Stack}} \mathit{And} \tag{7}$$

i.e. the composition of *Or*, *Unfold*, *Escape*, and *And* via *Stack*.

The complete specification of Action Notation is obtained by combining all modules of the architecture. Among the possible combinations of a smaller number of modules, we would like to mention the binary combination of the *functional* facet with one of the other modules. Using these combinations, useful examples of the single facets can be tested in isolation.

[5] In frameworks which have been categorically characterized, e.g. many-sorted algebraic specification, this operator usually corresponds to the *push-out* construction [EGRW98,EM85].

3.2 Data Notation and Yielders

Part of the AN is the Data Notation (DN), which is a collection of abstract data types given in terms of *algebraic specifications*. The predefined parts of DN can be implemented and executed, by interpreting the equations defining them as a term rewriting system. In MAN we include DN as a part of the underlying algebra. For each constructor in DN we define a constructor which is part of our ASM model. Formally the set *DN* contains all terms built up with these constructors.

The function *Eval* is used to evaluate data notation:

$$Eval : DN \rightarrow \mathfrak{U}$$

The concrete syntax rules of DN are included in the EBNF of MAN. The concrete syntax of DN-term t is parsed and transformed in the abstract syntax tree for t. The root of the tree (and of all subtrees) is in turn reconnected with the correct DN constructor term by means of the attribute

$$data : DN\text{-}parse\text{-}tree \rightarrow DN$$

Fig. 12 illustrates this process. A part of the EBNF of DN is given together with the corresponding canonical definition of constructors. The term "if true then sum(3,5) else 2" is an example of the provided DN fragment in concrete syntax. The corresponding abstract syntax tree is sketched, and each node is related to the corresponding term in *DN*, by means of the attribute *data* which is depicted by dotted arrows.

So far *DN* is completely static and corresponds to algebraic specifications. So-called *yielders* extend DN with constructs whose evaluation depends from the state or current information. As an example we shall see in the next subsection yielders that depend on the current storage. The semantics of a yielder is given by defining how it is evaluated by means of *Eval*.

3.3 The Basic Facet

In the basic facet four different forms of control flow are supported. These forms differ considerably from the control flow induced by the finite state machines in Montages. The main difference is that these forms depend on the surrounding structure of the AST in a dynamically recursive way, while the FSM support of Montages determines a static next-state relation. Where no static next-state relation exists, the control flow has to be modeled by explicit jumps. To determine the jump targets, we introduce a stack simulating the recursion.[6] The current value of the stack is given by a 0-ary dynamic function ranging over Stack. Since we use that function to simulate a situation similar to the one in structural approaches, we call it structural control-flow stack (SCS).

[6] Unfortunately we realized too late that recursive ASM calls following the structure of the ASTs could have been used to express structural control flow more directly.

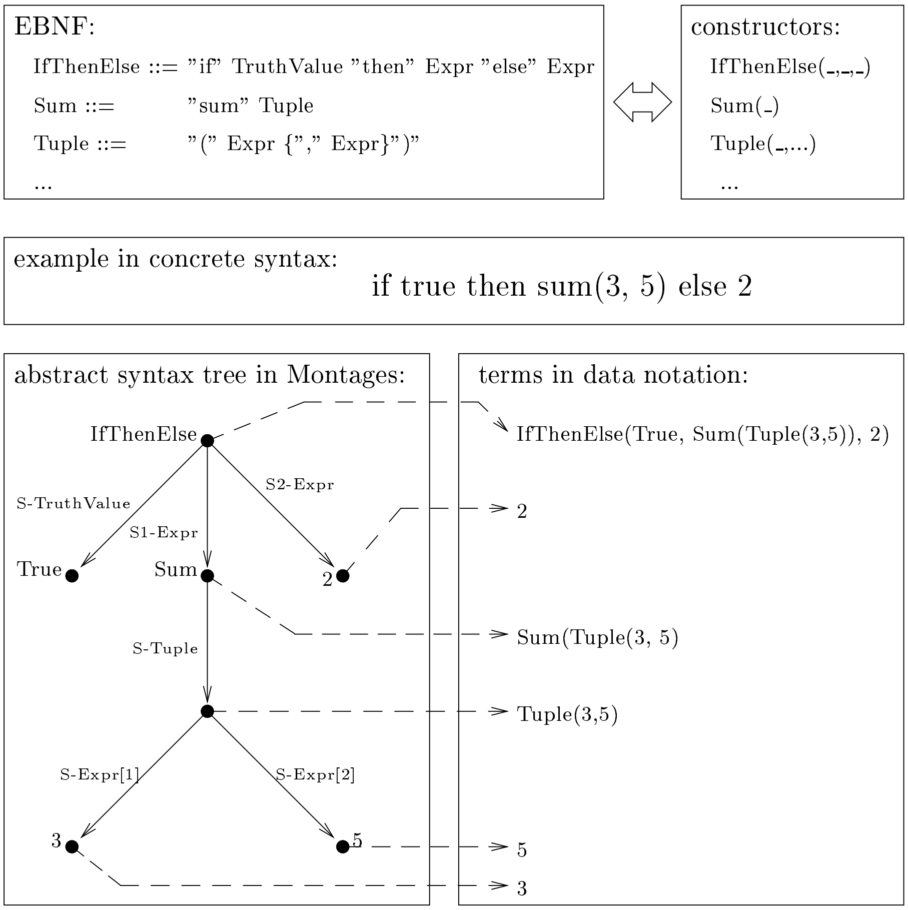


Fig. 12. The mapping from DN trees to DN terms

$$SCS :\rightarrow Stack$$

The ADT Stack has been introduced in Section 2.

Current Information Several actions in the basic facet are manipulating the so-called *current information*. Current information has several components which are introduced in the facets. The problem is to allow a description of the actions in the basic facet which is orthogonal to the other facets. As a solution we use the functions

$$\begin{aligned} \mathit{GetInformation} &: \ \rightarrow \mathit{Information} \\ \mathit{SetInformation} &: \ \mathit{Information} \rightarrow \\ \mathit{CombineInformation} &: \ \mathit{Information}, \mathit{Information} \rightarrow \mathit{Information} \end{aligned}$$

GetInformation is returning the current information; *SetInformation(i)* sets the current information to i, and *CombineInformation(i_1,i_2)* returns the information resulting out of combining i_1 and i_2. The concrete definition of these functions are refined in each facet. In the basic facet, no type of information is introduced, and the three functions correspond to skip rules. In the functional facet the transient information is introduced and in the declarative facet the scoped information is introduced. The state of the current store as used in the imperative facet is considered to be stable information. AN is designed such that stable information is not manipulated by above operations.

Unfolding and Unfold The composed action *Unfolding* and the primitive action *Unfold* are used mainly for iterative constructs. The meaning of *Unfold* is the execution of its syntactically least enclosing *Unfolding*. In the abstract syntax trees, the relation of Unfold instances to their least enclosing Unfolding instance is given by the lastUnfolding attribute.

$$\mathit{lastUnfolding} : \mathit{Unfold} \rightarrow \mathit{Unfolding}$$

In a functional AN model, one can regard *unfolding A* as an abbreviation for an action, generally infinite, formed by continually substituting A for *unfold*. In MAN we model *Unfold* as a call to the enclosing Unfolding. The *resumePoint* of the Unfold is put as return address on the stack, and control is passed to the lastUnfolding.

```
SCS := SCS.push(return(resumePoint))
JumpTo(lastUnfolding)
```

The Unfold montage in Fig. 13 thus has two action nodes, an unlabeled one, executing the above rule and the other serving as the resume point after a completed *Unfold*.
In the Unfolding montage in Fig. 13 first the Action component is executed. If after this execution there is a return Address $return(\&r)$ on the SCS, then control is passed back to &r, otherwise the Unfolding terminates.

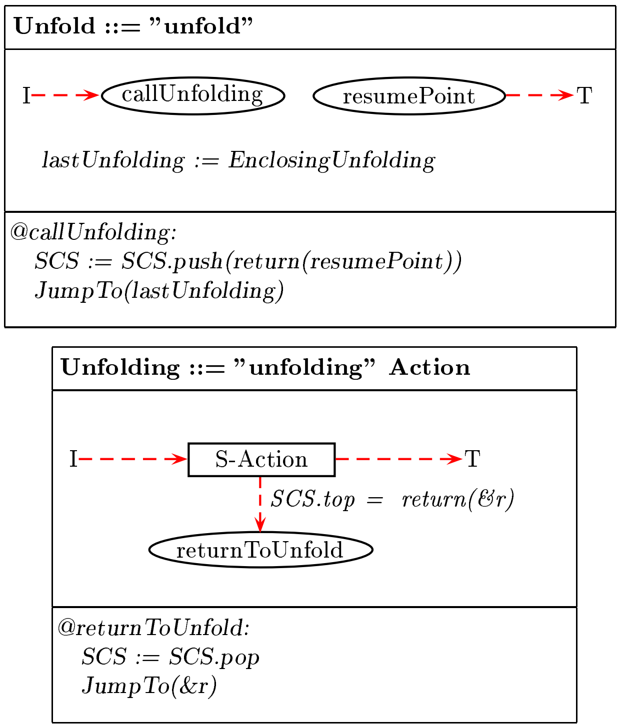


Fig. 13. The Unfold and Unfolding Montages

Example 1. The following example calculates 2 to the power of 3. We use an auxiliary action "action if _ then _else _ endif" which is not part of AN. The correct formulation of "action if c then a1 else a2 endif" in AN is "(check c then a1) or (check not c then a2)".

```
give 3
unfolding
  action if it is larger then 0 then
    give it - 1 then
    give unfold * 2
  else
    give 1
  endif
```

And then The action *A1 and then A2* executes *A1*, *A2* sequentially. In contrast to Then (see Fig. 16) the current information before the execution is passed in parallel to both, *A1* and *A2*. The current information of *A1 and then A2* is obtained by combining the current information after the execution of *A1* with the current information after the execution of *A2*. As mentioned the stable information is not altered by the manipulations of current information.

The control graph of the AndThen montage is a simple sequence. The first action node *pushAction* pushes the current information on the stack. Then the control is passed to the first action, and then to the action node *swapAction*. The swapAction pops the information *&inf* from the stack, pushes the current information on the stack, and finally sets the current information to *&inf*.

Or, Fail, and Commit The action *A1 or A2* represents implementation-dependent choice between alternative actions. If *A1* or *A2* fails, the other alternative is tried.

The first action node of Or chooses nondeterministically d among *left* and *right*. If d is equal to

left then the alternative right branch *S2-Action.Initial* is pushed together with the current information as

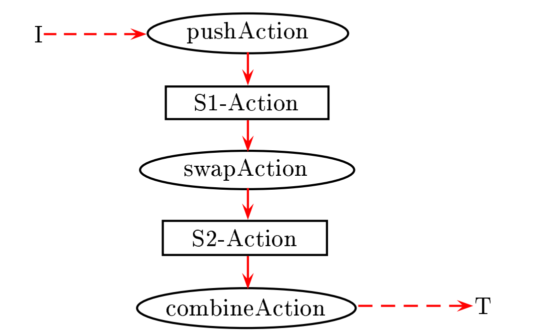
$$alternative(S2\text{-}Action.Initial, GetInformation)$$

where *alternative(_, _)* is a constructor. Further control is passed to the left branch *S1-Action.Initial*, using the *JumpTo(_)* operation which allows for non local and dynamically redirected arrows in the finite state machine.[7]

right the left branch is pushed, and control passed to the right one.

[7] These jumps can be defined graphically in MVL, but the needed graphical elements are not described here.

AndThen ::= Action "and" "then" Action



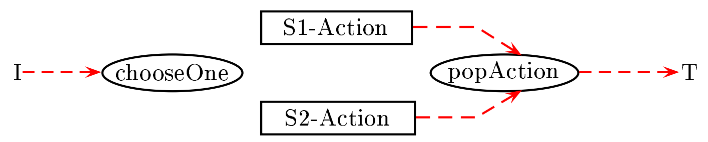

```
@pushAction:
  SCS := SCS.push(GetInformation)

@swapAction:
  if SCS = &l.push(&inf) then
     SCS := &l.push(GetInformation)
     !SetInformation(&inf)

@combineAction
  if SCS = &l.push(&inf) then
    SCS := &l
    !CombineInformation(&inf, GetInformation)
```

Fig. 14. The AndThen montage.

Or ::= Action "or" Action




```
@chooseOne:
  choose d in {left, right}
    if d = left then
      SCS :=
        SCS.push(alternative(S2-Action.Initial, GetInformation))
      JumpTo(S1-Action.Initial)
    else
      SCS :=
        SCS.push(alternative(S1-Action.Initial, GetInformation))
      JumpTo(S2-Action.Initial)

@popAction:
  SCS := SCS.pop
```

Fig. 15. The Or montage.

The primitive action Fail is modeled by an action node with the rule *!FailSemantics(SCS)*. The definition of FailSemantics is

```
method FailSemantics(x) is
  if  x =~ &s.push(alternative(&a, &i)) then
    !SetInformation(&i)
    SCS := &s.push(failed)
    JumpTo(&a)
  elseif x =~ &s.push(&any) then
    &s.!FailSemantics
  else
    _println("the program failed..")
```

In the case of an *alternative(&a, &i)* on top of the SCS, the current information is set to *&i*, *alternative(&a, &i)* is replaced with *failed*, and control is passed to *&a*. Otherwise *!FailSemantics* is called with the rest of the stack as argument. Like this the first alternative on the stack is searched recursively. If there is no alternative at all, the whole action failed, and execution is aborted.

The primitive action Commit replaces all *alternatives* on the stack with *failed* such that no backtracking is possible anymore.

3.4 The Imperative Facet

The imperative facet is the most natural part of AN to be specified with Montages because of the state-based nature of the framework. The simplest model of imperative behavior is a unary, dynamic function *CurrentStore* as used in the example of Section 2. A basic update *CurrentStore(x) := y* is used to update the store at position x to y. In the model of the imperative facet of AN, a more refined solution is needed.

In this section we give the specification of the imperative facet using an abstract data type *Map*. In contrast to the example in Section 2, the CurrentStore is a 0-ary function, ranging over the instances of Map.

$$\mathit{CurrentStore} :\to \mathit{Map}$$

The different operations are explained informally where needed, and in the Section 3.5 two alternative formal definitions of Map are given. One is a simple, abstract solution, and the other contains certain implementation-oriented decisions which overcome the problem of handling the store while dealing with large AN descriptions.

Reading the store The following yielder

YieldTheStoredAt ::= “the” Sort “stored” “at” Yielder is used to read the store at the cell denoted by the *Yielder*-component. If the result is of the specified sort, it is returned, otherwise the distinct element *nothing* is returned.

Using the definitions of selector attributes, the sort and yielder components can be accessed as *S-Sort* and *S-Yielder*, respectively.

For the integration in DN, a constructor *theStoredAt(_, _)* is introduced, and the field *data* is defined according to Section 3.2:

$$y.data = theStoredAt(y.S\text{-}Store.data, y.S\text{-}Yielder.data)$$

where y ranges over the instances of *YieldTheStoredAt*.

The definition of *Eval* is extended with the following equation:

Eval(theStoredAt(s, y)) =
  let r = CurrentStore.lookUp(y.Eval) in
    if r of type Eval(s) then r else *nothing*

where *_.lookUp(_)* is an operation of Map, used to read the store.

In AN one is only allowed to write and read in instances of the universe *Cell*. Cells are allocated using the primitive actions *Reserve*, *Unstore*, and *Unreserve*. For the ease of presentation, we do not use these, but instead we assume the existence of some yielders

$$cell_0, cell_1, \ldots$$

evaluating to instances of type *Cell*.

Getting a snapshot of the store A primitive yielder

YieldCurrentStorage ::= "current" "storage"

is introduced to get a snapshot of the store. The corresponding constructor is *currentStorage*, the definition of *data* is

$$y.data = currentStorage$$

where y ranges over the instances of *YieldCurrentStorage*. The extension of the definition of *Eval* is

Eval(currentStorage) = CurrentStore.getCopy

where *_.getCopy* is an operation of Map returning a snapshot of the current storage.

Writing the store The primitive action

StoreIn ::= "store" Yielder "in" Yielder is used to write the datum *S1-Yielder.data* in the cell *S2-Yielder.data*.

The StoreIn-montage consists of an action node associated with the following transition rule.

CurrentStore.upDate(S2-Yielder.data.Eval, S1-Yielder.data.Eval)

where *_.upDate(_, _)* is an operation of Map used to update the store.

In the definition of the imperative facet we have seen that *Map* is an abstract data type having the following operations.

$$\begin{aligned} upDate &: \; Map, Datum, Datum \rightarrow \\ lookUp &: \; Map, Datum \rightarrow \; Value \\ getCopy &: \; Map \rightarrow Map \end{aligned}$$

The first operation has an imperative behavior, whereas the second and third operations have a functional behavior. In the next section we introduce a direct implementation of the described behavior, and then a refined version where both the first and the third operations use imperative techniques.

Example 2. The following example makes use of the yielder $cell_0$, as explained above.

```
store 1 in cell0 then
store 2 in cell0
```

First 1 is stored in $cell_0$, then the content of $cell_0$ is overwritten by 2.

Example 3. The following example illustrates the use of *YieldCurrentStorage*.

```
store 5 in cell0 then
store current storage in cell0 then
store current storage in cell0
```

This AN description is executed in three steps. After the first step the store maps $cell_0$ to 5. After the second step, $cell_0$ is mapped to a snapshot of the store after the first step. After the third step, the store maps $cell_0$ to a copy of the store after the first two steps.

Then action combinator In the presented examples we made use of the *Then* action combinator, although conceptually the then-action does not belong to the imperative facet. The definition of *Then* by means of Montages is given in Fig. 16. The control graph of the *Then*-montage defines graphically

- the left action (accessible with S1-Action) to be the *initial* in the control flow of *Then*

– the right action (accessible with S2-Action) to follow the left sequentially and
– the right action to be the *terminal* in the control flow of *Then*.

The *Then*-combinator does not alter any kind of information thus no additional action node is required for it. Applying this montage to the above example results in a simple sequence of two *StoreIn* actions.

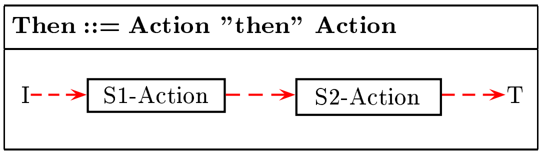


Fig. 16. The montage for the Then constructor

3.5 The ADT Map

The missing part for our model of the imperative facet is the definition of Map. We start by a simple definition that illustrates the use of ASMs. Later on, a more refined version is presented. The more refined version illustrates how our approach allows to solve efficiency bottle-necks on the specification level.

Simple Definition of Map Both for the simple and the refined definition, the ADT *Map* has an attribute *map*, being a unary, dynamic function from *Datum* to *Datum*. In the imperative facet this attribute is used to map cells to values.[8] In the object oriented style of our ASM interpreter the signature of *Map* is given as

```
class Map  is
  funattr map(_)
  method lookUp(x)
  method upDate(x,y)
  method getCopy
```

Initially the attribute *map* is undefined everywhere. No typing has to be provided, only the arity of attributes is needed. The *lookUp* and *upDate* operations are defined as:

```
 method lookUp(x) is
   lookUp_result:= map(x)
```

[8] In the declarative facet *map* is used to map tokens to bindable values.

```
method upDate(x,y) is
  map(x) := y
  upDate_result := self
```

The first operation is having a functional behavior, returning the result of reading the map-attribute, while the second method has an imperative behavior, having the side effect to update the map-attribute and returns the given instance of Map.

Consider the execution of *Example 1* starting with an empty store. After the first store action, $CurrentStore(cell_0)$ is equal to 1. After the second store, $CurrentStore(cell_0)$ is equal to 2.

The operation *getCopy* can be added to the simple definition as follows. A new Map is allocated and then the complete definition of the map is copied:

```
method getCopy is
  extend Map with newMap
    do forall x in dom map
      newMap.map(x) := map(x)
    enddo
    getCopy_result := newMap
```

This simple and abstract solution makes the tool unusable for large AN descriptions using the imperative facet. Thus in the next paragraph we propose an alternative refined definition of *Map*. The unusual property of this refinement is, that we mix functional and imperative specification parts.

Refined Definition of Map A more effective solution is to use a linked list of maps. The link from one map to the last is given by an attribute

$$lastMap : Map \to Map$$

To *lookUp* a value in such a list, all maps starting from the head are searched recursively. An *upDate* to a list of maps is performed on the head of the list only, the rest can be built up by snapshots. If a snapshot is taken using *getCopy*, the map is not cloned as above, but it is only marked by setting a flag *copyTaken*. When then next *upDate* operation is done, first a new map is allocated, and the *lastMap* attribute of the new map is set to the old map. Then the update is done only on the new map, guaranteeing that the old map is not altered and can be used as a valid snapshot.

The signature of Map extended with lastMap and copyTaken is

```
class Map  is
  funattr map(_)
  attr    lastMap
  relattr copyTaken

  method lookUp(x)
```

```
  method getCopy
  method upDate(x,y)
```

In the *lookUp* method, first the attribute *map* is searched, and then *lookUp* is called recursively on *lastMap*:

```
method lookUp(x) is
  if map(x) != undef then
    lookUp_result := map(x)
  elseif lastMap != undef then
    lookUp_result := lastMap.lookUp(x)
  else
    lookUp_result := undef
```

The attribute *copyTaken* is declared as a 0-ary relation which is initialized as *false*. It is used as a flag, to remember whether a copy has been taken. The method *upDate* is functionally the identity function, and as a side effect it sets the *copyTaken* flag:

```
method getCopy is
  copyTaken := true
  getCopy_result := self
```

The update of such a map takes into account whether a copy has been taken. If yes, then a new map is allocated, and linked to the old one via *lastMap*. The new value is updated in the new map, and this map is returned.

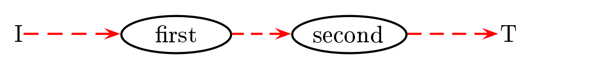
If no copy has been taken, then no new map is needed and the update is made on the old map, which is returned as result.

```
  method upDate(x,y) is
   if not copyTaken then
     map(x):= y
     upDate_result := self
   else
     extend Map with m
       m.lastMap := self
       m.map(x) := y
       upDate_result := m
```

Assume we execute *Example 2* using the refined definition. At the end of the execution three instances of *Map* exist, reflecting the state of the store after step one, two and three. All of them are linked by *lastMap*.

A consequence of our proposed solution is that the evaluation of the components of StoreIn and the execution of upDate cannot be done simultaneously. In Fig. 17 the control flow of StoreIn consists thus of two action nodes, the first evaluating the components and storing the results in the attributes tmpValue and tmpCell, and the second executing the upDate operation.

StoreIn ::= "store" Yielder "in" Yielder



@first:
tmpValue := Eval(S1-Yielder.data)
tmpCell := Eval(S2-Yielder.data)

@second:
CurrentStorage.upDate(tmpCell, tmpValue)

Fig. 17. The Montage for the StoreIn action

4 The Generated Environment

The development environment for Montages is given by the Gem-Mex tool [AKP97a,AKP97b]. It is a complex system which assists the designer in a number of activities related with the language design process. 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 documentation can be generated automatically; Fig. 18 shows the editor opened for the Or Montage;
- 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 dynamic behavior of the specified language at a symbolic level; source programs written in the specified language and user-defined data structures can be animated and inspected in a visual environment.

In our case, the generated environment for AN has different purposes. In particular, while developing or extending the AN specification, it served to test empirically and validate the intended semantics of AN. At the same time, the same support can be used to visualize, debug, and explain the behavior of AN descriptions in terms of the specification (origin tracking).

4.1 Generation of Language Interpreters

Using the formal semantics description given by the set of Montages and the auxiliary ASM classes, the Gem-Mex system generates an interpreter for the specified language. No additional implementation details are requested to be provided by the designer. The core of the Gem-Mex system is *Aslan*, which stands for *A*bstract *S*tate Machine *Lan*guage and provides a fully-fledged implementation of the ASM approach. Aslan can also be used as a stand-alone,

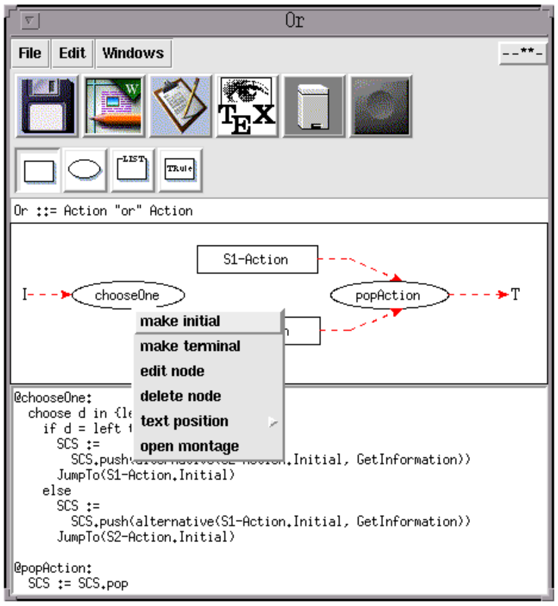


Fig. 18. The graphical editor of the Gem-Mex tool

general purpose ASM implementation. The process of generating an executable interpreter consists of two phases:

1. The Montages containing the language definition are transformed to an intermediate format and then translated to an ASM formalization ("`montages2asm`" in Figure 19).
2. The resulting ASM formalization is processed by the Aslan compiler generating an executable version of the formalization, which represents an interpreter implementing the formal semantics description of the specified language.

Using Aslan as the core of the Gem-Mex system provides the user the possibility to exploit the full power of the ASM framework to enrich the graphical ASM macros provided by Montages with additional formalization code.

4.2 Generation of Visual Programming Environments

Besides pure language interpreters, the Gem-Mex system is able to generate visual programming environments for the generated ASM formalization of the programming language semantics.[9] This is done by providing a generic debugging

[9] This feature is again available to all kind of ASM formalizations implemented in Aslan not only to those generated from a Montages language specification.

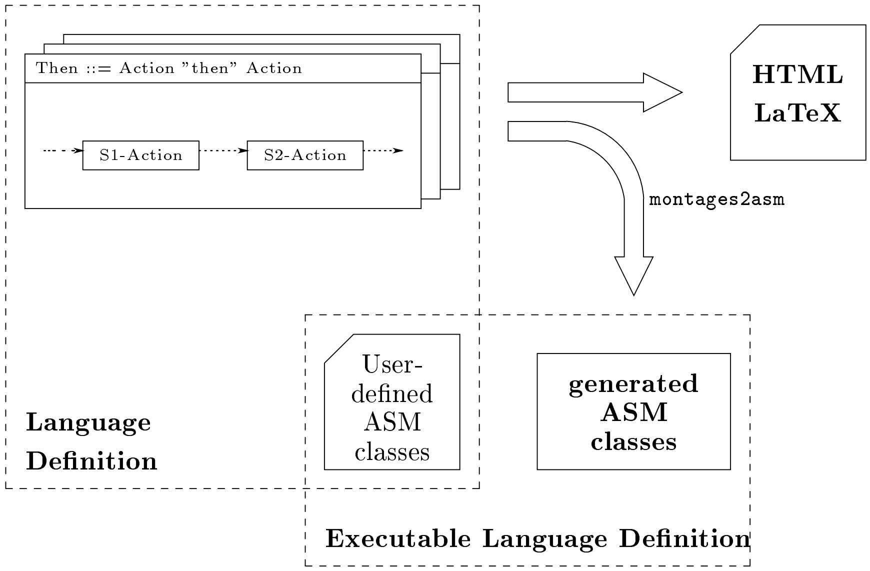


Fig. 19. The architecture of the Gem-Mex system

and animation component which can be accessed by the generated executable. During the translation process of the Montages/ASM code special instructions are inserted that provide the information being necessary to visualize the execution of the formalization. In particular, the visual environment can be used to debug the specification, animate the execution of it, and generate documents representing snapshots of the visualization of data structures during the execution. The debugging features include stepwise execution, textual representation of ASM data structures, definition of break points, interactive term evaluation, and re-play of executions. Fig. 20 shows an example of this kind of visualization, where the AN description of Example 1 is illustrated in the topmost window (*view source code*). In particular, for that particular description the universe *Unfolding* is containing the node denoted by #148[10] (window *Unfolding*) and the selector function *S-ActionFactor* links the unfolding node occurrence with its action argument (node #147), which is highlighted in the text (window *S-ActionFactor*).

Figure 21 shows an example of the graphical animation facility of the Gem-Mex system. On the right-hand-side of the window the AN program of Example 1 is visualized and the position information generated during the compilation process of the Montages is displayed. This position information is used, for example, to highlight certain parts of the source code that correspond to values of data structures contained in the language formalization. In Figure 21, the change of the value of the "current-task" function *CT* is animated by drawing

[10] The node #46 corresponds to the Aslan class definition of *Unfolding*.

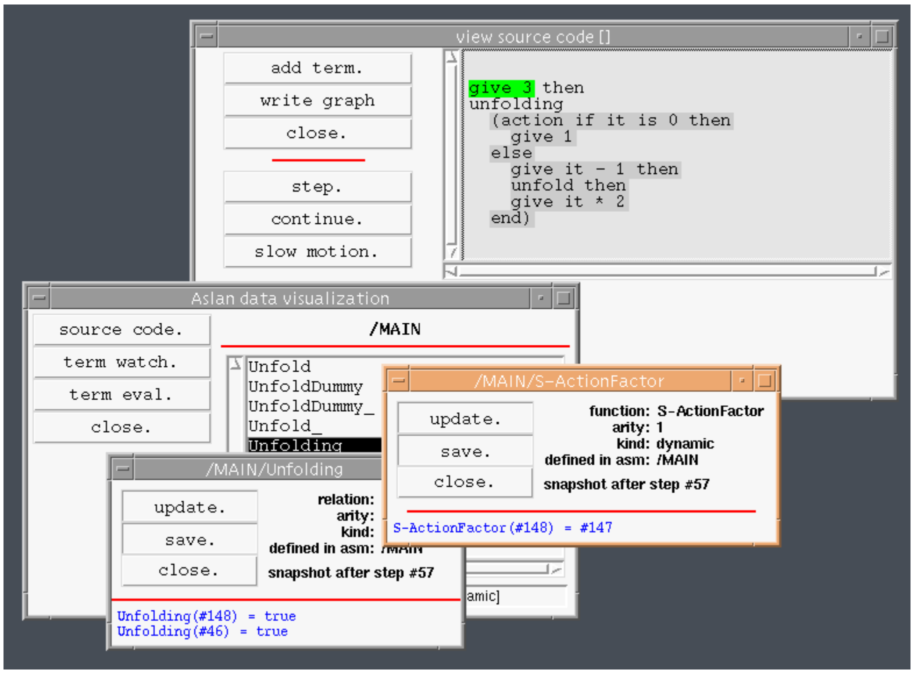


Fig. 20. Textual Visualization of data structures in the Gem-Mex system

an arrow from *unfold* to its "replacement" given by the argument of *unfolding*. Experiences show that especially this kind of animation is useful to explain and document the formal semantics as specified in the Montages.

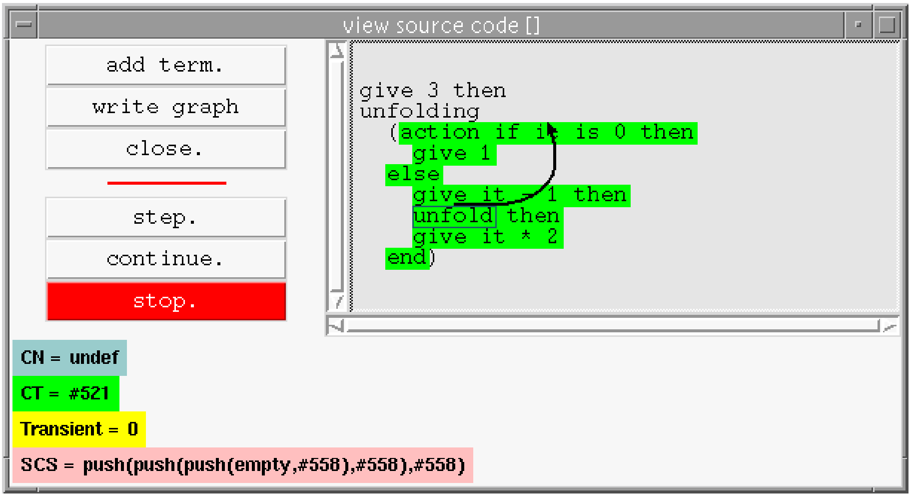


Fig. 21. Graphical animation in the Gem-Mex system

With the "write graph" button in figure 21 one can trigger the production of a graphical representation of the syntax nodes and their interconnections, like data and control flow arrows, selection functions, and initial and terminal arrows. Gem-Mex generates an input file for the "VCG" tool [San95] which can be used to visualize these data structures. As an example, Figure 22 displays a portion of the abstract syntax tree of the examples program displayed in Figure 21.

4.3 Generation of Documentation Frames

As sketched in Figure 19 the Gem-Mex system also generates files that can be used as frames for the documentation of the language specification. Both paper and online presentation of the language specification are automatically generated:

– LaTeX documents illustrate the Montages and the grammar; such documents are easily customizable for the non-specialist user; all Montages in this paper are generated by Gem-Mex;
– HTML versions of the language specification allows to browse the specification and retrieve pieces of specification.

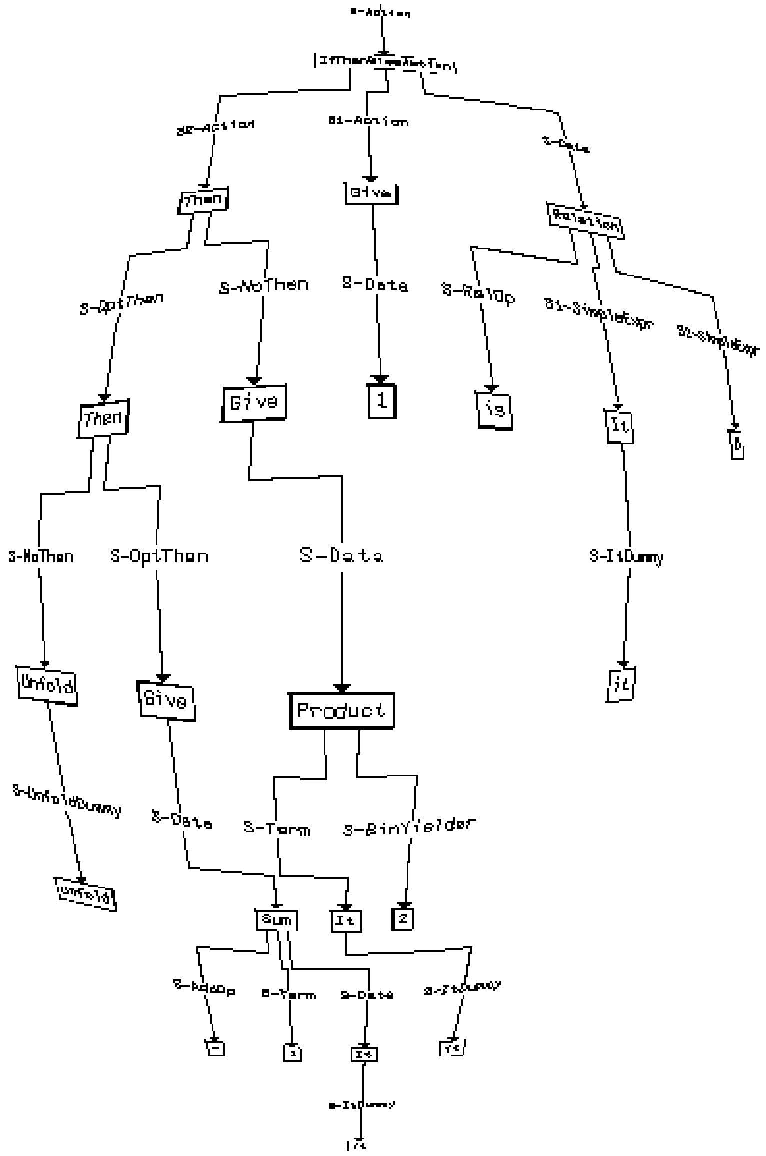


Fig. 22. Visualization of a portion of the abstract syntax tree of the example in Figure 21

4.4 Library of Programming Language Features

A concept for providing libraries of programming language features is currently under development. With this concept is shall be possible to reuse features of programming languages that have already been specified in other Montages. Examples for this kind of features are arithmetic expressions, recursive function call, exception handling, parameter passing techniques, standard control features etc. The designer of a new language can then import such a feature and customize it according to his or her needs. The customization may range from the substitution of keywords up to the selection among a set of variants for a certain feature, like different kinds of inheritance in object-oriented languages, for example. This would allow, for instance, to embed the AN behavior in other languages reusing part of it or extending it. In the Verifix project [HLT98], a number of reusable Montages has been defined with the intention to reuse not only the Montages but as well an associated construction scheme for correct compilers.

Acknowledgments We thank Samarjit Chakraborty, Christoph Denzler, Yuri Gurevich, Wuwei Shen, and Chuck Wallace for their collaboration in the Montages project.

References

[AKP97a] M Anlauff, P. W. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.

[AKP97b] M. Anlauff, P. W. Kutter, and A. Pierantonio. The Gem-Mex Tool Homepage. `http://www.first.gmd.de/~ma/gem/`, 1997.

[AKP97c] M. Anlauff, P. W. Kutter, and A. Pierantonio. The Montages Project Web Page. http://www.tik.ee.ethz.ch/~montages, 1997.

[AKP98] M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced Control and Data Flow Graphs in Montages. 1998. submitted for publication.

[Anl] M. Anlauff. The Aslan Language Manual. Part of the Aslan distribution.

[BH98] E. Börger and J. Huggins. Abstract state machines 1988 – 1998: Commented ASM bibliography. In H. Ehrig, editor, *EATCS Bulletin, Formal Specification Column*, number 64, pages 105 – 127. EATCS, February 1998.

[BW99] D. Brown and D. A. Watt. JAS: a Java Action Semantics. In *Proceedings of AS'99 (to appear)*, BRICS notes series, 1999.

[EGRW98] H. Ehrig, M. Große-Rhode, and U. Wolter. Applications of category theory to the area of algebraic specification in computer science. *APCS (Applied Categorical Structures)*, (6):1–35, 1998.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[GH93] Y. Gurevich and J. K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Theory and Practice of Software Engineering*, pages 1–57. CS Press, 1988.

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

[Gur97] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department Technical Report, 1997.

[HLT98] A Heberle, W. Löwe, and M. Trapp. Safe reuse of source to intermediate language compilations. Fast Abstract, 9th International Symposium on Software Reliability Engineering, September 1998. http://chillarege.com/issre/fastabstracts/98417.html.

[Hug] J. Huggins. Abstract State Machines Web Page . http://www.eecs.umich.edu/gasm.

[KH95] P. W. Kutter and F. Haussmann. Dynamic Semantics of the Programming Language Oberon. Term work, ETH Zürich, July 1995. A revised version appeared as technical report of Institut TIK, ETH, number 27, 1997.

[KP97a] P. W. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *JUCS, Springer*, 3(5):416–442, 1997.

[KP97b] P. W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *JUCS, Springer*, 3(5):443–503, 1997.

[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*, LNCS. Springer, October 1998.

[MM93] M. A. Musicante and P. D. Mosses. Communicative action notation with shared storage. Tech. Mono. PB-452, Dept. of CS, Univ. of Aarhus, 1993.

[Mos92] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in theoretical Computer Science. Cambridge University Press, 1992.

[Mos98a] P. D. Mosses. Modularity in natural semantics (extended abstract). Available at http://www.brics.dk/~pdm, 1998.

[Mos98b] P. D. Mosses. Modularity in structural operational semantics (extended abstract). Available at http://www.brics.dk/~pdm, 1998.

[Ode89] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon*. PhD thesis, ETH Zürich, 1989.

[Ode98] M. Odersky. Programming with variable functions. In *International Conference on Functional Programming*, Baltimore, 1998. ACM.

[Ørb94] P. Ørbæk. OASIS: An optimizing action–based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edingurgh*, volume 786 of *LNCS*, pages 1–15. Springer Verlag, 1994.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.

[San95] G. Sanders. Graph layout through the vcg tool. In I. G. Tollis R. Tamassia, editor, *Graph Drawing, DIMACS International Workshop GD'94, Proceedings*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205. Springer Verlag, 1995.

[vDM96] A. van Deursen and P. D. Mosses. ASD: The action semantic description tools. In Springer, editor, *AMAST'96 Proc., 5th Intl. Conf. on Algebraic Methodology and Software Technology*, number 1101 in LNCS, pages 579 – 582, Munich, 1996.

[Wal97] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, University of Michigan EECS Department Technical Report, 1997.

[Wan97] K. Wansbrough. A modular monadic action semantics. Master's thesis, Dept. of CS, Univ. of Auckland, February 1997.

[Wat91] D. A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, 1991.

[Wat99] D. A. Watt. The Static and Dynamic Semantics of SML. In *Proceedings of the AS'99 (to appear)*, BRICS notes series, 1999.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, 1992.