Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Generating an Action Notation Environment from Montages Descriptions

Matthias Anlauff[1], Samarjit Chakraborty[2], Philipp W. Kutter[2], Alfonso Pierantonio[3], Lothar Thiele[2]

[1] GMD FIRST, Rudower Chaussee 5, D-12489 Berlin, Germany; e-mail: ma@first.gmd.de
[2] Institut TIK, Eidgenössische Technische Hochschule Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland; e-mail: {samarjit, kutter, thiele}@tik.ee.ethz.ch
[3] Dipartimento di Mathematica Pura ed Applicata, Universitá di L'Aquila, I-67100 L'Aquila, Italy; e-mail: alfonso@univaq.it

The date of receipt and acceptance will be inserted by the editor

Abstract. In this paper we present an implementation of action notation based on a formal and modular specification of its semantics. This specification allows an automatic generation of an environment consisting of an interpreter and a debugger which allows the visualization of an action notation program execution and relates it to the given formal specification.

The semantic description presented here is based on Montages, which is a semi-visual formalism for the specification of the syntax and semantics of programming languages. The tool support for Montages, called Gem-Mex is used to execute the action notation specification and thereby generate an environment for executing action notation programs. Moreover, the specification maintains and refines the structuring of actions in terms of *facets* and thus ensures that the modularity present in action notation is retained in its semantic description.

1 Introduction

Action semantics [30,34,52] is a specification formalism for the semantics of programming languages. It was proposed by Peter D. Mosses with the aim of improving the modularity of denotational semantics and making it more pragmatic so that semantic descriptions scale up smoothly to realistic languages and become more widely usable, for example by programmers. Semantic functions in this case, as in denotational semantics, map the abstract syntax of language constructs to semantic entities. However, the difference in this case being that the semantic entities are *actions* rather than the usual higher-order functions expressed using λ-notation. Actions are esentially dynamic computational entities, and the *performance* of an action is supposed to directly represent its information processing behavior. The notation used for specifying actions is called action notation (AN). In this paper we describe a modular operational semantics of AN using Montages [4,29], a formalism which was proposed to give modular and executable specifications of programming languages. A result of this is that the tool support existing for Montages, called Gem-Mex [2, 3], can now directly be used to execute action denotations of programs of a language specified using action semantics and thereby generate a tool environment for action semantics.

Action semantics being a relatively new formalism, currently still in the process of development, there is yet no single established and mature tool support available for it. The Action Semantic Description (ASD) tools [47–49] can be used for denoting programs in a language specified using action semantics, with their action denotations. However, such actions can not be directly executed. The ASD tools implemented using the ASF+SDF system [25], are mainly concerned with the static aspects of action semantic specifications. They provide facilities for checking and maintaining descriptions of languages and also deal with parsing, syntax-directed and textual editing, and type checking of programs in the specified language. Our work presented in this paper can be viewed as an effort to supplement these existing facilities by providing tools concerned with the dynamic semantics of AN, which can be used for generating an environment for debugging and executing action denotations of programs and visualizing such executions.

There exists a number of compiler generators based on action semantics, which generate compilers for languages taking their action semantic descriptions as input. An example of this is the OASIS compiler generator [40] by Ørbæk. It takes action semantic descriptions of languages written in Scheme and produces a compiler in Perl. The compiler takes a textual representation of an abstract syntax tree of a program in the source language and produces an executable code for the SPARC

processor. Another example in the same direction is the Actress system [9] which when supplied with an action semantic description of a language generates a compiler which compiles programs in this language to C code corresponding to their action denotations. A module of the Actress system can also be used to parse a textual action to the corresponding action notation abstract syntax tree and then interpret this tree to finally generate the action's outcome. Other examples of similar compiler generators are the Abaco system [10], extensions of Actress by Doh and Schmidt [12] to support techniques for typing and binding-time analysis, and the Cantor system [41,42] which was used to generate a compiler of a subset of Ada from its action semantic description. The main emphasis in the last case was on the generation of provably correct compilers.

While such compiler generators based on semantic descriptions are useful once the complete semantic specification of a language already exists, writing such specifications is often difficult and error prone. The task is even more complicated if the language being specified is itself in the process of being designed. To remedy this situation, meta-environments supporting a formalism are used while writing the semantic definitions of languages. The aim here is to provide facilities which significantly ease the process of writing and maintaining a specification, experimenting with it, and then finally generating a prototypical environment for programming in the specified language. The ASD tools exactly provide such an environment for action semantics. They are used to perform various checks to ensure the correctness of specifications, for example verifying that the terms used in the equations are written only using symbols that were introduced. Finally, the specification can be used to automatically generate tools like syntax-directed editors and parsers for programs in the specified language. It also translates such programs into their action denotations. The Montages specification of the dynamic semantics of AN can be used to interpret such action denotations generated by the ASD tools, and graphically visualize the executions. Using the tool environment associated with Montages it is possible to establish a direct relationship between the specification and the execution of actions and thereby helps in revealing possible errors in the specification of (the dynamic semantic aspects of) a language, as well as in the programs written in it.

The formulation of AN by Mosses is based on classifying actions into a number of *facets* depending on the kind of information processed by an action. Each facet of actions focusses on at most one kind of information at a time. For example, the *basic* facet refers to only control flow and contains actions that perform independent of any information, while the *functional* facet refers to data passed between successive actions. This automatically leads to a degree of modularity in action semantic descriptions, because actions involving a certain kind of behaviour need not be concerned with actions in other facets. The Montages description of AN follows this partition of actions into facets and therefore preserves the underlying modularity. Since the tool support is based on this modular description, it allows the possibility to develop, test and validate AN specifications in small pieces. Once the correct behaviours of the modules are assessed in isolation, one can then investigate the interaction between them when they are composed together. Such a development process makes the resulting implementation transparent and readily accessible to the user and can easily accomodate possible future extensions in AN.

It should be noted that since it was considered to be difficult to provide a satisfactory denotational semantics for the full AN, its formal semantics was originally defined by Mosses in the Appendix C of [30] using a style of Structural Operational Semantics (SOS) [43]. A drawback of this was that the inherently poor modularity of SOS extended over to this specification making it to a large extent inaccessible, in sharp contrast to the definition that we present here. Very recently (after the first appearance of a preliminary version of this paper as [6]), Mosses has rewritten this original monolithic definition using a new technique called Modular SOS [36], also defined by him. This new semantic definition of AN [37] is more clear, modular, and would facilitate easy extensions to support new semantic structures. Other attempts to provide modular semantics to AN include a modular monadic action semantics by Wansbrough and Hamer [51].

Montages is a semi-visual formalism for the specification of syntax and semantics of programming languages. Its goals are very similar to those of action semantics in the sense that it also aims at being a pragmatic framework which would scale up to the requirements of full-fledged realistic languages and be useful to programmers for language engineering, without sacrificing a formal basis. It relies on context-free grammars (EBNF) for the specification of syntax, finite state machines for graphically specifying control flow, and Abstract State Machines [7,19,23]. for specifying dynamic semantics. The formalism is supported by a tool-suite called Gem-Mex which helps in writing and maintaining a specification and then executing it to generate a prototypical programming environment for the specified language. The generated environment consists of a parser, an interpreter and a debugger accompanied with animation facilities.

To summarize the main contributions of this paper, we present a formal semantics of action notation using Montages following the same modular formulation by Mosses which classifies actions into different facets depending on the kind of information processed by them. The specification can be directly executed and such executions can be visualized. Combined with the ASD tools, this can form a complete environment for language specification using action semantics. While the ASD tools deal with the static aspects of a specification, this work

addresses its dynamic behaviour. The next section briefly describes action notation, followed by a description of the aspects of the Montages formalism necessary for understanding the paper in Section 3. In Section 4 we describe the semantics of action notation using Montages. The complete description is available[1] and can be executed using the Gem-Mex tool. Section 5 gives an impression of the generated action notation environment and shows how it serves as a platform for conducting an empirical validation of the dynamic semantic aspects of a language specification.

2 Action Notation

For the sake of completeness, in this section we briefly summarize the main concepts behind AN. Further details can be found in [30] or in the tutorial [35] by Mosses.

An action-semantic description of a language is composed of three parts: the abstract syntax of the language, the semantic entities, and the semantic functions which map (abstract syntax of) programs to actions. The specification of the semantic entities consist of a specialization of the general AN to the specific needs of the language being defined. All the three components of the specification are based on the framework of unified algebras [31–33] which is similar to many-sorted algebraic specifications. For example, in the definition of abstract syntax trees in this framework, if the start symbol of the grammar of the specified language is called *PROGRAM*, then there would be a sort `PROGRAM` containing all possible abstract syntax trees representing programs in this language. There would also be many subsorts of this sort, corresponding to the different grammar productions. A non-terminal on the left hand side of a grammar production is interpreted as a sort construct and the alternatives on the right-hand side as sort terms, combined with sort union. This is in contrast to the usual interpretation of grammars as signatures in algebraic specifications.

However, most of unified algebra specifications can be read as ordinary many-sorted algebraic specifications. As an example, in the specification of AN, the *action* abstract data type is defined by introducing a sort `ACTION` and defining certain equalities over terms over this sort. These terms, or actions, are the semantic entities denoting the meaning of programs, and AN is the notation for specifying these actions. Apart from the sort `ACTION`, the algebraic specification of AN involves a signature introducing a number of function symbols which take as arguments terms of `ACTION` and return a value of `ACTION`. Constants of the sort `ACTION`, for example *complete*, denote *primitive actions* (which is explained below) and the function symbols denote action combinators used to combine different primitive actions. As an example, the `_ and then _` combinator for sequential left-to-right execution of two actions can be given as follows.

[1] `http://www.tik.ee.ethz.ch/~montages/an/`

```
_ and then _ :: ACTION, ACTION -> ACTION
```

In addition to AN, there also exists a *data notation* which is used to define a collection of data types like integers, strings, etc. The data notation is also defined in the same style as the AN using algebraic specifications. The specification of a language generally contains definitions of various new data types which involves extending the general data notation with further sorts of data, and a specialization of the standard sorts with sort equations.

The third and the last part of a language description are the semantic functions which are ordinary equationally-specified operations mapping (abstract syntax of) programs in the specified language to their action denotations.

Here we would like to point out that our model of specifying the semantics of AN, as will be described in the next two sections, is based on the Abstract State Machine (ASM) formalism. This is a state based formalism for operationally specifying the semantics of arbitrary algorithms where states are defined by means of algebras similar to ones used in algebraic specifications.

For writing action-semantic descriptions of languages Mosses has developed a *meta notation* (MN) based on unified algebras. This notation is slightly different from a conventional algebraic specification language and offers some special features like modularization constructs and special facilities to define abstract syntax. The MN is described in the Appendix F of [30], and the full MN specification of AN is given in the Appendix B of the same book. The ASD tools for action semantics provide a library of MN modules which include the full AN and data notation modules. While writing the action-semantic description of a language using these tools, before using the AN one has to specify the information processed by actions because, as already mentioned, this would usually vary according to the language being specified. In summary, the MN is used to specify syntax, AN, data notation, data types, and mappings from syntax to AN. For the purpose of this paper we do not need to know any details about this notation.

So far we have described the denotations of programs, i.e. the semantic entities of action semantics, to be actions. There are actually three kinds of entities: actions, *data*, and *yielders*. While the main kind are actions, data and yielders are subsidiary. It was already mentioned that the performance of an action, represents the dynamic behaviour of the program to which this action corresponds, and the information processed by actions consist of data items. The data referred to by the actions is accessed by means of yielders. These are essentially entities which when evaluated during action performance yield data. As an example, a yielder might evaluate to the datum currently stored in some particular cell. Based on the kind of information processed, actions are grouped into a number of *facets*. The actions of a particular facet are concerned with the processing of at most one kind of data, as described below.

- The *basic* facet refers to only control flow and the actions perform independently of any information.
- The *functional* facet deals with the processing of transient information i.e. data which is passed between successive actions.
- The *declarative* facet is concerned with the processing of scoped information i.e. bindings of tokens to data, corresponding to symbol tables.
- The *imperative* facet is for the processing of stable information i.e. the storage of data in cells. Actions in this case *reserve* and *unreserve* storage cells, and *change* the data stored in the cells.
- The *communicative* facet is used to specify message passing, i.e. communication of data between distributed actions. Actions in this case *send* and *receive* messages.

These different facets of actions perform independently of each other. The performance of an action can have four possible outcomes. It either *completes*, which corresponds to normal termination; or *escapes*, corresponding to exceptional termination; or *fails*, corresponding to abandoning an alternative; or *diverges*.

As previously mentioned in this section, AN consists of *primitive actions* and action *combinators*. The primitive actions are essentially single-faceted, affecting only one kind of information. The combinators combine actions from different facets, resulting in multi-faceted actions. As an example, the `_ and then _` combinator mentioned above, when used as `A1 and then A2` combines the actions `A1` and `A2` resulting in the performance of `A2` only when `A1` completes. Using the MN, terms standing for actions, combinators, as well as data and yielders are written as ordinary English words which result in action-semantic descriptions to be very readable. So an action might read like `check it and then escape`. Terms denoting data and yielders are nouns like `the items of the given list`. Finally, as a simple example consider the following equation. It gives an impression of the notation and shows how complex actions can be built from primitive actions, combinators, and yielders.

```
execute[[ "if" E "then" S1 "else" S2 ]] =
  evaluate E then
  (  (check it and then execute S1) or
     (check not it and then execute S2) )
```

Here `execute` is a semantic function giving the semantics of the abstract syntax enclosed within the square brackets. The term on the right-hand side of the equality is the action corresponding to the abstract syntax. The performance of this action begins by calling `evaluate E`. The functional action combination `A1 then A2` results in the transient information generated by `A1` on its completion being given to `A2`. The control flow follows a left-to-right sequencing from `A1` to `A2`. Therefore, in this example, the result of evaluating the expression `E` is passed on to the next action by `then`.

The action `check` in this case when applied to the yielder `it` succeeds if the value yielded is true and fails otherwise. The action combinator `_ or _` represents an implementation-dependent choice between alternative actions. If the alternative currently being performed fails, then it is abandoned and the other alternative is performed if possible. Therefore in this example if the yielder `it` evaluates to true then the action `execute S1` is performed, otherwise `execute s2` is performed.

3 Montages

In this section the Montages formalism is introduced, on which the results of the paper are based. Like action semantics, Montages is also a formalism for the specification of programming languages. Here we shall describe only those features which are relevant to the description of action notation. A more detailed description of the formalism is available in [4,29].

The primary aim of Montages is to allow a formal documentation of the decisions taken during the design process of realistic programming languages. The GemMex tool [2,3] supporting the formalism on one hand to allows the designer to 'debug' a semantic description by empirically testing whether the intended decisions have been properly formalized, and on the other hand it allows automatic generation of a correct (prototype) implementation of a programming environment for the specified language. Such an environment includes interpretation, visualization, and debugging facilities.

Our work on Montages was originally motivated by the formal specification of the C language [20][2], which showed how the state-based Abstract State Machine formalism (ASMs) [16,17,23] is well-suited for the formal description of the dynamic behavior of full-fledged programming languages. In essence, ASMs constitute a formalism in which a state is updated in discrete time steps. However, unlike many other state-based systems, the state is given by an algebra i.e. 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 [20] for describing the dynamic semantics of C was based on the assumption of the existance of a control and data flow graph. This was a major limitation of the model since the control and data flow graph is a crucial part of any language specification. Montages addresses this problem by extending the approach of [20] and introducing a mapping which describes how to obtain the control and data flow graph

[2] Historically the C case-study was preceded and paralleled by work on Pascal [16], Modula2, Prolog, and Occam. See [7] for a commented bibliography on ASM case studies.

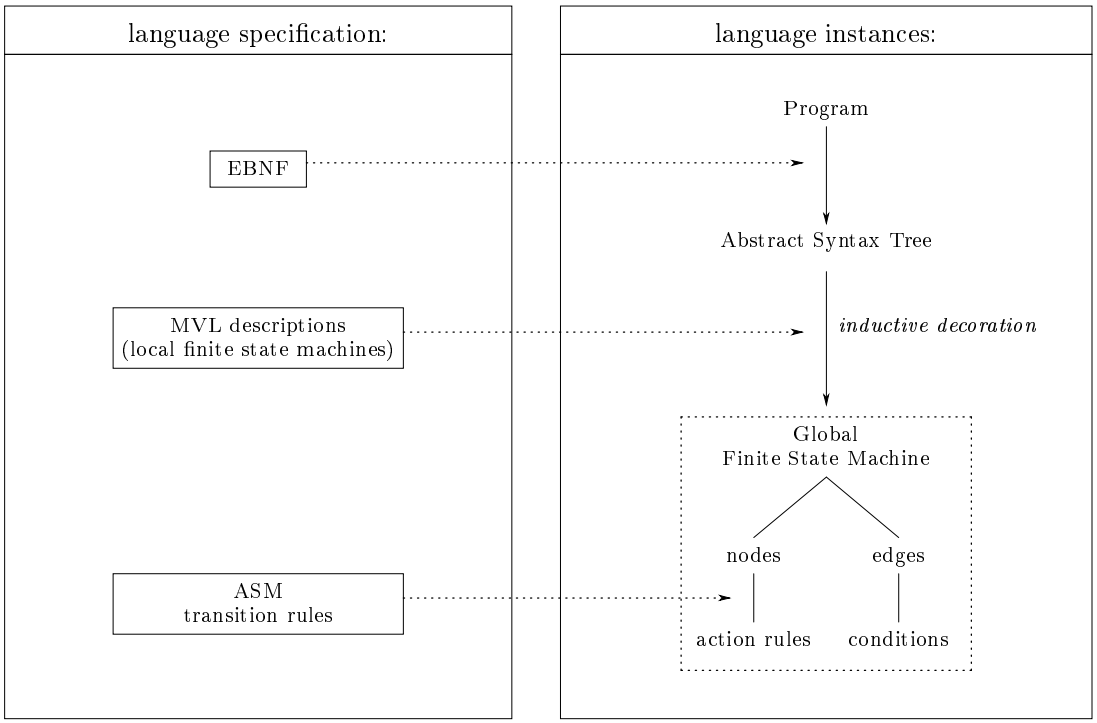

Fig. 1. Relationship between language specification and instances.

from an abstract syntax tree of a program in the specified language.

The original formulation of Montages was strongly influenced by a case study where the Oberon language [26,27] was specified. Other more recently done case studies include a specification of Java [50], the use of Montages as a front-end for correct compiler construction in the Verifix project [15,22], applications of Montages to component composition [5] and its use in the design and prototyping of a domain-specific language [28]. These have led to several improvements in the formalism which have been reported in [4]. For example, the control flow is now formulated as state-transitions in a hierarchical finite state machine. These changes enhanced the pragmatic qualities of the formalism like extensibility and readability and shortens the learning curve of an user considerably. Complete documentation related to the formalism, details of the case studies, and information about the Gem-Mex tool can be obtained from the Montages homepage[3].

As depicted in Figure 1, a language specification using Montages can be considered to consist of three parts.

1. The EBNF production rules are used for the context-free syntax of the specified language L, and allow the generation of 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 ASTs. Section 3.1 describes the details of this mapping. In Figure 1 the dotted arrow from the EBNF rules indicate 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 (FSMs). A MVL description associated with an EBNF rule defines a *local* FSM, along with the information required for plugging this FSM into a *global* FSM corresponding to a program via an inductive decoration of the abstract syntax tree. Towards this end, each node of the tree is decorated with a copy of the finite state machine fragment given by its Montage. References to descendent nodes in the AST defines an inductive construction of the global FSM. In Section 3.2 we define how this construction works.
3. Finally, any node in the FSM may be associated with an ASM *rule*. This rule is fired when the node becomes the current state of the FSM. As shown in Figure 1, the specification of these rules is the third part of a Montages specification. The underlying abstract state machine formalism is described in Section 3.3.

A complete language specification is structured into specification modules, called Montages. Each Montage corresponding to a language construct describes the semantics of that construct, and can be considered in some sense a "BNF-extension-to-semantics". As described above, it specifies the context-free grammar rule (by means of EBNF) of the construct, the (local) finite state machine

[3] `http://www.tik.ee.ethz.ch/~montages/`

(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 to exactly one Montage.

As an example, the Montage for a nonterminal with name `Sum` is shown in Figure 2. The topmost part of this Montage is the production rule defining the context-free syntax. The second part defines the static aspects of this construct by means of an MVL description. Finally, the Montage contains an action rule which is executed when the control reaches the sum node.

The description of a construct usually also contains a fourth section which is devoted to the specification of the static semantics of the construct. As we shall not use this feature in this paper, we refrain from describing it here. Future work will use this feature to formalize the static-semantics of AN, following the work in [40] and [49].

3.1 From Syntax to AST

In this section, the first step in Figure 1 is described. As a result of this step we get the abstract syntax tree of a program in the specified language. This also forms the basis for composing the Montages corresponding to the different constructs of the language based on the structure of the AST of the program.

EBNF rules The syntax of the specified language is given by the collection of all the EBNF rules defined in the different Montages. 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 has 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*. It is then guaranteed 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*[4] whose instances are associated with the corresponding nodes in the abstract syntax tree. Symbols on the right-hand side of a characteristic EBNF rule are called the (direct) *components* of the Montage, and symbols which are reachable via these components are called the *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 rule given above, the B, C, and D components of an A instance can be retrieved by the selectors S-B, S-C, and S-D respectively. In Figure 3 a possible representation of the A-Montage as a class and an abstract syntax tree (AST) with two instances of A and their components are depicted.

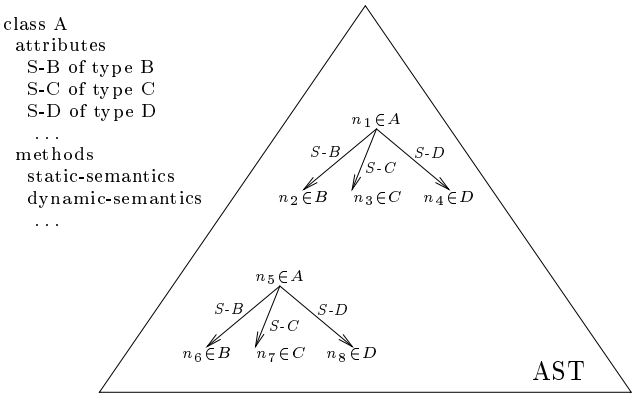

Fig. 3. Montage class A, instances in the AST, selectors S-B, S-C, S-D

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 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 an automatic generation of AST from the concrete syntax given by EBNF; see also the work in [38].

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. 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 Figure 4. The use of such a path in a Montage definition imposes a number of constraints on the other EBNF rules of the

[4] In this context we consider a 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.

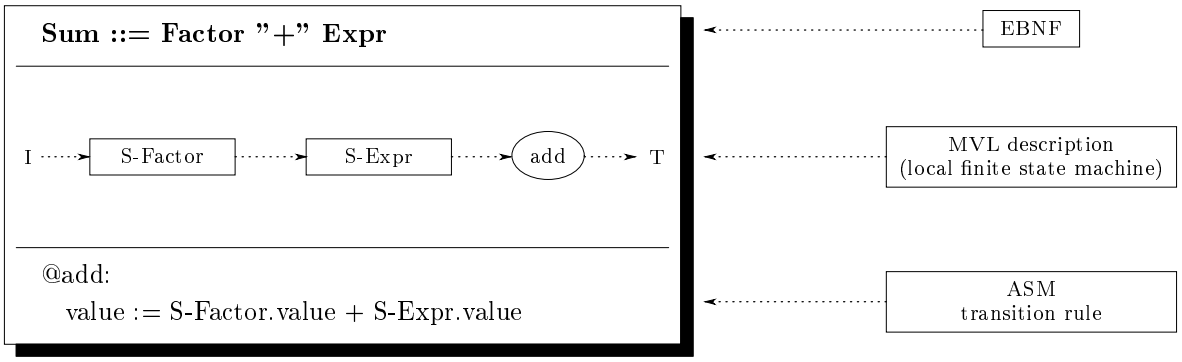

Fig. 2. Montage components.

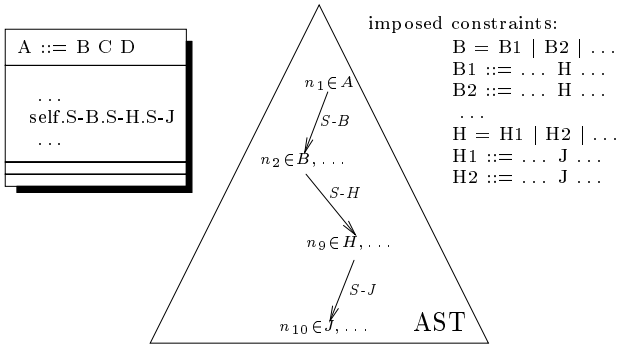

Fig. 4. Montage A using path self.S-B.S-H.S-J, situation in AST, and constraints on EBNF rules of B, H.

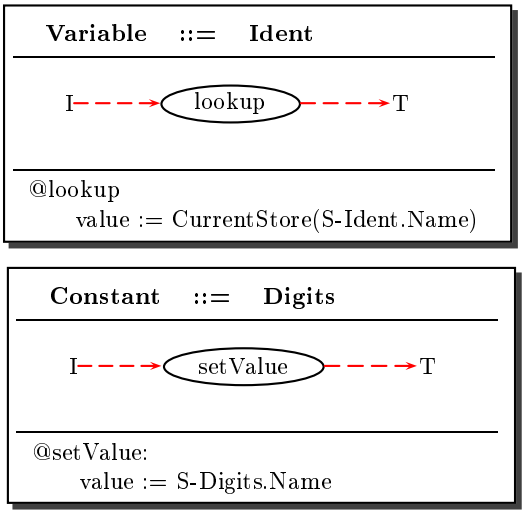

Fig. 5. The Montages for the language $\mathcal{S}$.

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.

Example 1. As a running example we give a small language $\mathcal{S}$. The expressions in this language 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 Figure 2 and Figure 5 we obtain the structure represented in Figure 6.

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.

3.2 From AST to Control Flow Graphs

According to Figure 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.

As we have seen in Figure 2 and Figure 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 with the node of

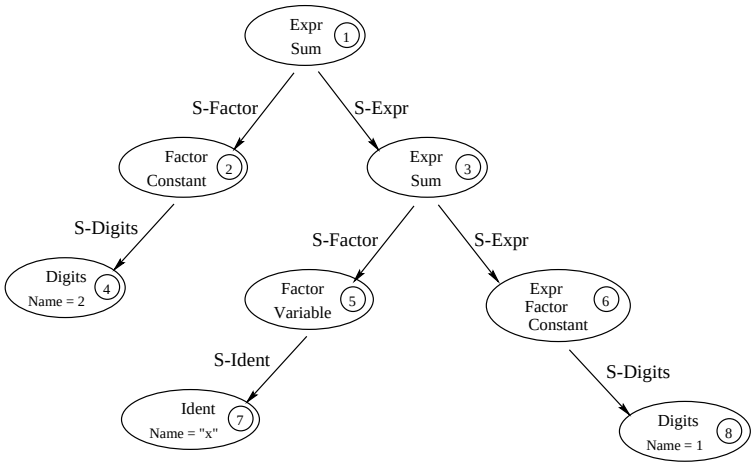

Fig. 6. The abstract syntax tree and composition of Montages for `2 + x + 1`

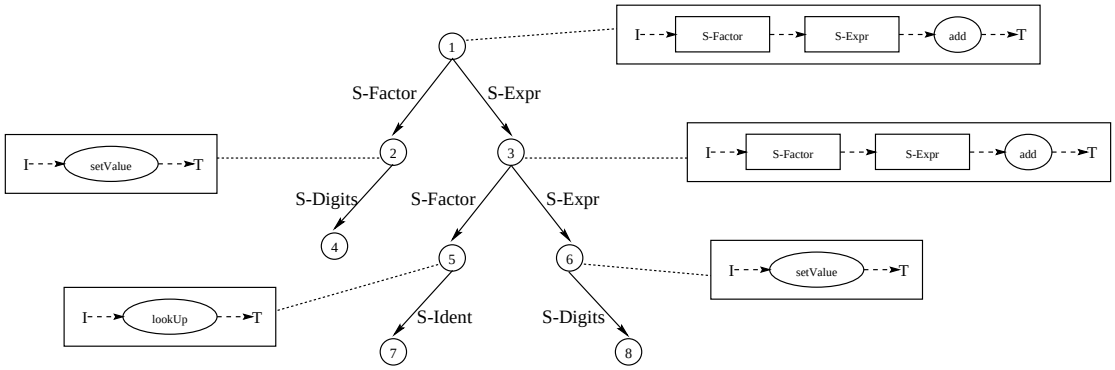

Fig. 7. The finite state machines belonging to the nodes.

the AST and (b) the information required for embedding of this local state machine into a global state machine. Using our running example, Figure 7 just represents the MVL sections of the Montages as they are associated with the corresponding nodes of the abstract syntax tree. The hierarchical state transition graph resulting from the inductive decoration is shown in Figure 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 with the AST nodes corresponding to the Montages. The oval nodes are labeled with an attribute. It 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 3.1. 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,

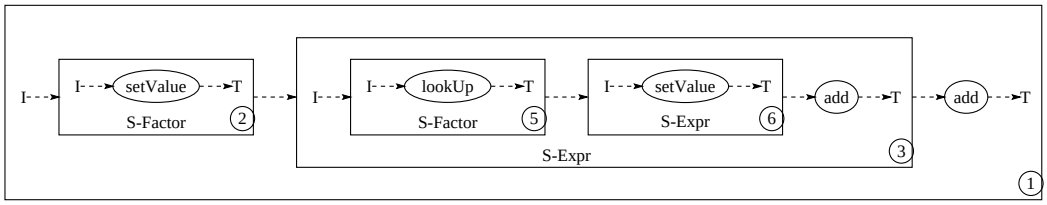

Fig. 8. The constructed hierarchical finite state machine.

in this part of a Montage, one may specify further action rules to be performed during the static analysis phase, for example building up data structures necessary for the static and dynamic semantics.

Now it only remains to be shown how the hierarchical finite state machine (for example as that of Figure 8) is built and how its dynamic semantics 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. Recall 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 Figure 8 is obtained.

Dynamic Semantics After the static analysis phase, action rules defining the dynamic semantics of the language are then executed. The steps involved in this process are as follows.

- States of the finite state machines are visited sequentially.
- The action rule associated with a visited state is executed. The specification of these actions is based on the ASM formalism and is specified in Section 3.3.
- The control is passed with 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 with the source node.
  If there is more than one possible next state, the system behaves like a nondeterministic FSM. In this paper we do not make use of this feature.
- 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 and respectively the last state that is visited. Following the above

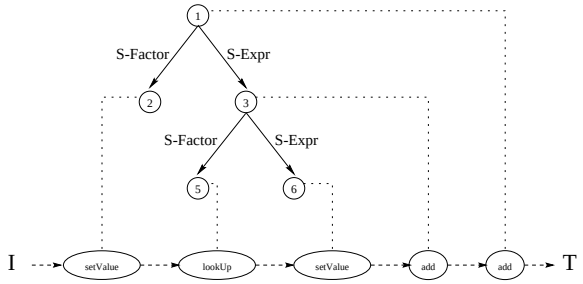

Fig. 9. The flat finite state machine and its relation to the AST.

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 Figure 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 shown in Figures 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 attribute *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.

It may be noted here that all these concepts informally described above have been formalized in terms of ASMs. 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 4 via the function *JumpTo*. The underlying ASM formalism is described in the next section.

3.3 Dynamic Semantics by means of ASM rules

Basic ASM formalism The fundamental concept behind ASMs is that a state is represented by an algebra i.e. a collection of functions and universes, and state transitions occur by updating functions and extending universes with new elements. Functions here are defined over a set $\mathfrak{U}$ which in ASM parlance is called the *superuniverse*. This set always contains the distinct elements *true*, *false*, and *undef*. Apart from these $\mathfrak{U}$ can contain numbers, strings, and possibly anything – depending on what is being modeled. For example in our context, i.e. describing the semantics of programming languages, we will assume the existance of an abstract syntax tree and all the nodes of this tree will be elements of $\mathfrak{U}$.

Formally, the *state* λ of an ASM is a mapping from a signature Σ (which is a collection of function symbols) to actual functions. We use f_λ to denote the function which corresponds to the symbol f in the state λ. Attributes of the nodes of an AST are modeled by unary functions and the value of an attribute a of a node n can be retrieved by the function application $a(n)$. For convenience we also allow the dot notation which is used in attribute grammars and object-oriented programming.

In our framework Σ contains a unary function for each attribute of a Montage. Examples of this are the selector attributes, and specific to our running example, the attribute *value* shown in Figure 2. Subsets of the superuniverse $\mathfrak{U}$, called *universes*, are modeled by unary functions from $\mathfrak{U}$ to *true*, *false*. Such a function returns *true* for all elements belonging to the universe, and *false* otherwise. A function f from a universe U to a universe V is a unary operation on the superuniverse such that for all $a \in U$, $f(a) \in V$ and $f(a) = undef$ otherwise. The universe *Boolean* consists of *true* and *false*, and in our case there also exists a universe *Node* whose elements are the nodes of the abstract syntax tree. Additionally, the abstract syntax definition introduces a number of node-types, following which the nodes of the same type can be grouped together in a universe. Universes like *Number*, *String*, etc. can be used to group together the respective elements.

A basic ASM *transition rule* is of the form

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

where $f(t_1, \ldots, t_n)$ and t_0 are closed terms (i.e. terms containing no free variables) in the signature Σ. The semantics of such a rule is this: evaluate all the terms in the given state, and update the function corresponding to f at the value of the tuple resulting out of evaluating $(t_1, \ldots, t_n)$ to the value obtained by evaluating t_0. Rules are composed in a parallel fashion, so the corresponding updates are all executed at once. Apart from the basic transition rule shown above, there also exist *conditional* rules where the firing depends on the evaluated boolean condition-term, *do-for-all* rules which allow the firing of the same rule for all the elements of a universe, and lastly *extend* rules which are used for introducing new elements into a universe. Transition rules are recursively built up from these rules.

Of course not all functions can be updated. The basic arithmetic operations (like add, which takes two operands) are typically not redefinable. Other examples of static functions are the attributes defined by static semantic equations, which therefore can not change throughout the entire program execution. An example of a dynamic function would be an attribute *value* of the `Sum` nodes in Figure 6, to hold the current value of an expression. The update of this attribute will be given by the rule

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

which abbreviates

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

The meaning of this rule is that the attribute *value* of *self* is updated to the sum of the values of the left and the right arguments 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 and their construction is given by ASm rules.

The formal semantics of a rule R in a state λ is given by its denotation $Upd(R, \lambda)$, which is a set of updates. The resulting state-transition changes the functions corresponding to the symbols in Σ in a point-wise manner, using *updates*. An update is a triple

$$(f, (e_1, \ldots, e_n), e_0)$$

where f is a n-ary function symbol in Σ and $e_0, \ldots, e_n$ are elements of $\mathfrak{U}$. Intuitively, firing this update in a state

λ changes the function associated with the symbol f in λ at the point $(e_1, \ldots, e_n)$ to the value e_0, leaving the rest of the function (i.e. its values at all other points) unchanged.

Firing the updates in $Upd(R, \lambda)$ in the state λ results in its successor state λ'. For any function symbol f from Σ, the relation between f_λ and $f_{\lambda'}$ is given by

$$f_{\lambda'}(e_1, \ldots, e_n) = \begin{cases} e_0 & if \quad (f, (e_1, \ldots, e_n), e_0) \in Upd(R, \lambda) \\ f'_\lambda(e_1, \ldots, e_n) & otherwise \end{cases}$$

The different forms of rules are given below. We use $eval_\lambda$ to denote the usual term evaluation in the state λ.

Basic Update
if $R = f(t_1, \ldots, t_n) := t_0$
where $t_0, \ldots, t_n$ are terms over Σ, then
$Upd(R, \lambda) = (f, (eval_\lambda(t_1), \ldots, eval_\lambda(t_n)), eval_\lambda(t_0))$

Parallel Composition
if $R = R_1 \ldots R_m$
then $Upd(R, \lambda) = \bigcup_{i \in \{1, \ldots, m\}} Upd(R_i, \lambda)$

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

Do-for-all
if $R =$ **do forall** x **in** U
R'
enddo
then $Upd(R, \lambda) = \bigcup_{e \in U} Upd(R', \lambda_{x \mapsto e})$
where $\lambda_{x \mapsto e}$ is the same as the state λ with the addition that the nullary function symbol x is interpreted as the element e.

Extend
if $R =$ **extend** U **with** x
R'
endextend

then $Upd(R, \lambda) = Upd(R', \lambda_{x \mapsto e})$,
where e does not belong to the domain or the co-domain of any of the functions corresponding to the symbols in Σ.

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

Classes and Methods In Section 3.1 we introduced the concept of classes in Montages, 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 4 this technique will be used to present several abstract data types which encapsulate basic concepts of Action Notation. We have found that this enables the semantics and the tool for Montages to readily extendible to new areas.

In addition, classes allow multiple inheritance and recursive, dynamically bound methods. The sub-typing of classes and synonym classes mentioned in Section 3 is an application of inheritance. The method calls have a value parameter semantics, and are used in several places in Section 4.

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 one-to-one and total.

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

(i) If for all $t_1, \ldots, t_n$ such that $eval_\lambda(t_i) \neq undef$ with $1 \leq i \leq n$, then
$$f(t_1, \ldots, t_n) \neq undef$$

(ii) If for each m-arity $g \in \Sigma_c$ and $t_1, \ldots, t_n$ such that $eval_\lambda(t_i) \neq undef$, then
$$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$$
iff
$$(f = g) \text{ and } (n = m) \text{ and } (eval_\lambda(t_i) = eval_\lambda(s_i))$$
for all $1 \leq i \leq n$

where $eval_\lambda(t)$ stands for the evaluation of the term t in state A of the ASM. Informally speaking it means that each constructive function is total with respect to $\mathfrak{U}$ and 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 $eval_\lambda(t)$.

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*, $eval_\lambda(empty) \in Stack$ and $eval_\lambda(push(t_1, t_2)) \in Stack$, 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 $t_1 =\sim t_2$. 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

$$x.push(y).pop = x$$
$$x.push(y).top = y$$
$$empty.top = undef$$
$$empty.pop = empty$$

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 4 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 abstract data type *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 [39]. In Section 4 many examples for this technique are shown, including a refinement from a functional specification of an ADT Map into a more imperative specification. In [39] it is shown how such refinements can be proved to be correct.

The tool support of Montages is based on the ASM compiler XASM [1]. XASM is a conservative and faithful implementation of the ASM formalism as defined by Gurevich in [17] and [18]. Furthermore XASM allows extensions to basic ASMs like classes and constructors, whose semantics is formalized using Montages. The use of XASM for a pure ASM application is presented in [46].

4 Action Notation Specification

As already mentioned in the previous sections, the Montages specification of AN consists of a collection of Montages and ADTs as those described in the last section. These are grouped into a number of modules and the architecture of these modules follows and refines the usual partition of AN into facets. Section 4.1 gives an overview of this architecture. In Section 4.2 it is shown how the data notation is specified by combining techniques described in Sections 2 and 3, followed by a description of how the different facets are specified in the remaining sections.

4.1 Architecture

Our formal description of AN consists of a collection of interconnected specification modules. Each module consists of a number of Montages and ADT descriptions and define either a facet or a part of it. A module might also import Montages and ADTs described in other modules. Additionally, each module also contains Montages and ADTs which allow the module to be run and tested in isolation. When several modules are composed, these *test* Montages and ADTs do not however carry over and compose, but have to be rewritten for the combined larger module.

The specification architecture is illustrated in Figure 10, where the solid arrows denote an import relation between modules and the dotted arrows indicate composition (which is described below).

The module *DataNotation* described in Section 4.2 is used in all the other modules. The module *Stack* consists of the ADT Stack along with some other functions which are described in Section 4.3. *Stack* is used by the four modules *Or*, *Unfold*, *Escape*, and *And* which partition the actions of the basic facet. Each of these modules works in isolation and does not refer to parts of other modules. In Section 4.3 the *Or* and the *Unfold* modules are introduced in detail.

The module *Map* mainly consists of the definition of an ADT Map. This ADT is used both in the imperative facet and the declarative facet. In Section 4.4 *Map* is informally introduced and is used to explain the imperative facet. Later, in Section 4.5, a simple and refined specification of *Map* is presented.

The specification of the declarative facet uses a combination of techniques used for specifying the basic and the imperative facets described in Sections 4.3 and 4.4 respectively. Since this specification does not pose any problems or give rise to any new questions, we do not describe it further in this text. It has however been specified and entered in the Gem-Mex tool. The functional facet is briefly described at the end of Section 4.3. The communicative facet, which is concerned with the communication of data between distributed actions has not yet been implemented since the present version of the Gem-Mex tool does not support concurrency. There are several possibilities of expressing concurrency in the Montages and work in this direction is currently in progress. In future we plan to incorporate the communicative facet into the existing specification.

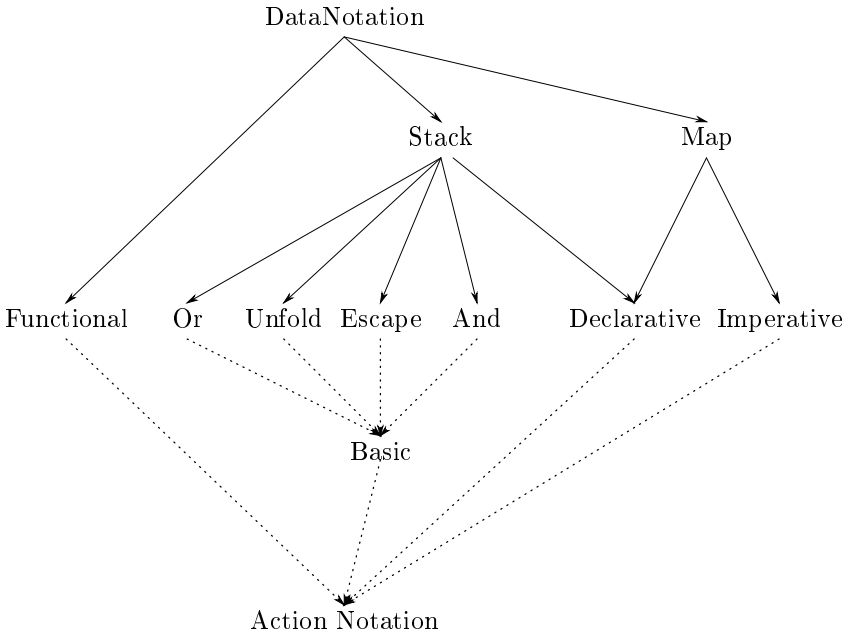

Fig. 10. Specification architecture

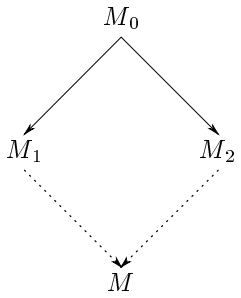

Fig. 11. A simple module decomposition.

The described modules can be arbitrarily combined, leading to the specification of *sublanguages* of AN. The composition of modules along the dotted arrows can always be reduced to expressions of the following form, as depicted in Figure 11.

$$M = M_1 \oplus_{M_0} M_2 \tag{5}$$

This means that the module M is obtained from the union of the modules M_1 and M_2, both of which use the module M_0. We call this the *composition* of M_1 and M_2 via M_0.[5] For instance, the specification of *Basic* is obtained as

$$\mathit{Basic} = \mathit{Or} \oplus_{\mathit{Stack}} \mathit{Unfold} \oplus_{\mathit{Stack}} \mathit{Escape} \oplus_{\mathit{Stack}} \mathit{And} \tag{6}$$

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

[5] In frameworks which have been categorically characterized, e.g. many-sorted algebraic specification, this operator usually corresponds to the *push-out* construction [13,14].

The complete specification of AN is obtained by combining all the modules, as shown in Figure 10. The sublanguages obtained from combining two or more modules can be independently tested leading to a modular testing procedure. The module M obtained by composing M_1 and M_2 as shown in Figure 11 allows the reuse of test cases for M_1 and M_2. A successful test of reused test cases serves as a validation of the composition. For example, we have developed test examples for the composition of the *Functional* and the *Or* modules, using which features like the non-deterministic choice and operaters like fail and cut have been tested.

4.2 Data Notation and Yielders

In addition to AN there is a Data Notation (DN), which as mentioned in Section 2 is a collection of abstract data types given in terms of algebraic specifications. The Montages specification of a language models each parse tree as an algebra, as explained in Section 3. In the case of DN we use in parallel the parse tree as well as the data notation model described in Section 2, whose terms we refer to as *constructor terms*.

Formally the set *DN* contains all terms built up with these constructors. The function *Eval*, used to evaluate data notation is given by

$$\mathit{Eval} : \mathit{DN} \to \mathfrak{U}$$

and its implementation is specified by the abstract data types described in Section 2. The concrete syntax rules of DN are included in the EBNF of the Montages specification. The concrete syntax of a DN term t is parsed and transformed to the abstract syntax tree for t. The

root of the tree (and of all subtrees) is in turn associated with the correct DN constructor term by means of the attribute

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

Figure 12 illustrates this concept. 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 concrete syntax of a DN fragment. 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.

Yielders extend the DN with constructs whose *evaluation* depends on the state and the current information. In Section 4.4 we shall describe examples of yielders, the result of whose evaluation would depend on the current storage. The semantics of a yielder is given by defining how it is evaluated by means of *Eval*.

4.3 The Basic Facet

In the basic facet four different forms of control flow are supported. The actions related to each of these forms are grouped in a module. The *Or* module contains actions for modeling nondeterministic choice, the *Unfold* module is concerned with recursive models of iteration, the *And* module with parallel interleaved execution, and finally the *Escape* module deals with basic exception handling. All these forms differ considerably from the control flow induced by the finite state machines in Montages. The control flow primitives in AN originate from a declarative tradition, whereas Action Notation programs are treated as freely generated terms from the AN grammar. In Montages, AN programs are treated as parse trees, where all components have their own object-identity. Declarative techniques therefore do not carry over to our setting, and the declarative intuition behind control flow primitives have to be replaced by a state based imperative intuition.

Such an intuition is given with the FSM support for Montages, thereby defining 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 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 a structural control flow stack (SCS).

$$\textit{SCS} :\rightarrow \textit{Stack}$$

The ADT *Stack* was introduced in Section 3.

[6] An alternative approach would be to use recursive method calls taking the ASTs as argument.

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

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

The function *GetInformation* returns 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 definitions 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 the 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 given by 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.

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

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

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

The Unfold Montage in Figure 13 therefore has two action nodes, one executing the above rule and the other serving as the resume point after a completed Unfold.
In the Unfolding Montage in Figure 14, first the *Action* component is executed. After this execution, if there is a return address *return*(&r) on the SCS, then the control is passed back to &r, otherwise the Unfolding terminates.

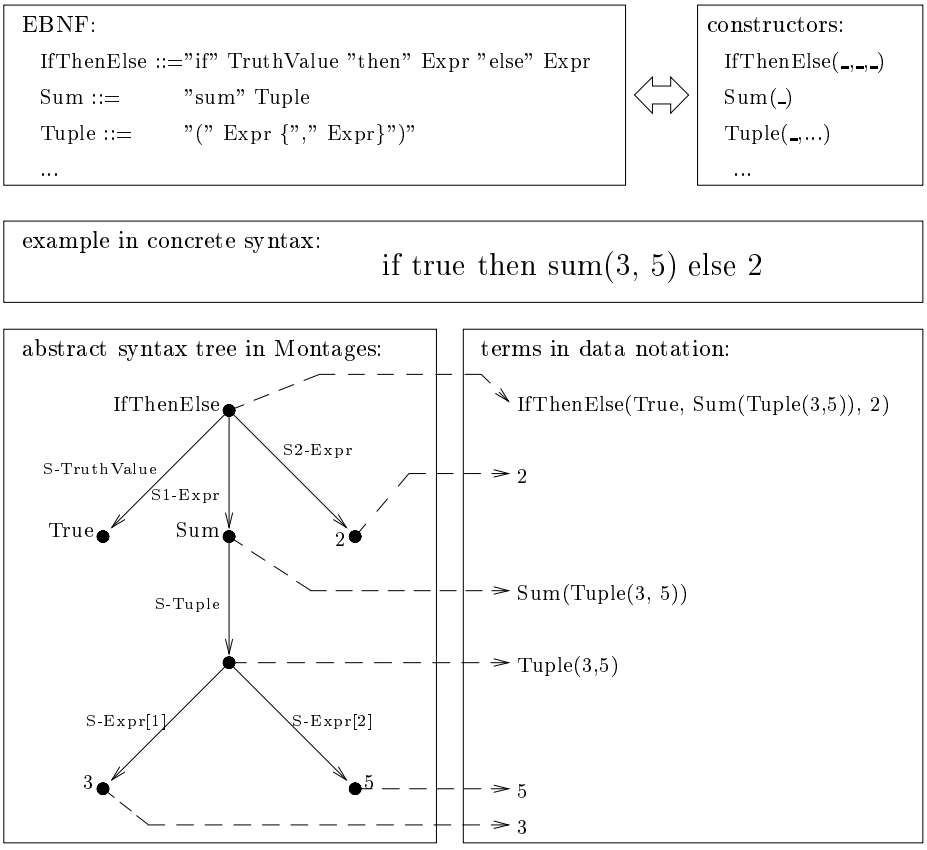

Fig. 12. The mapping from DN trees to DN terms

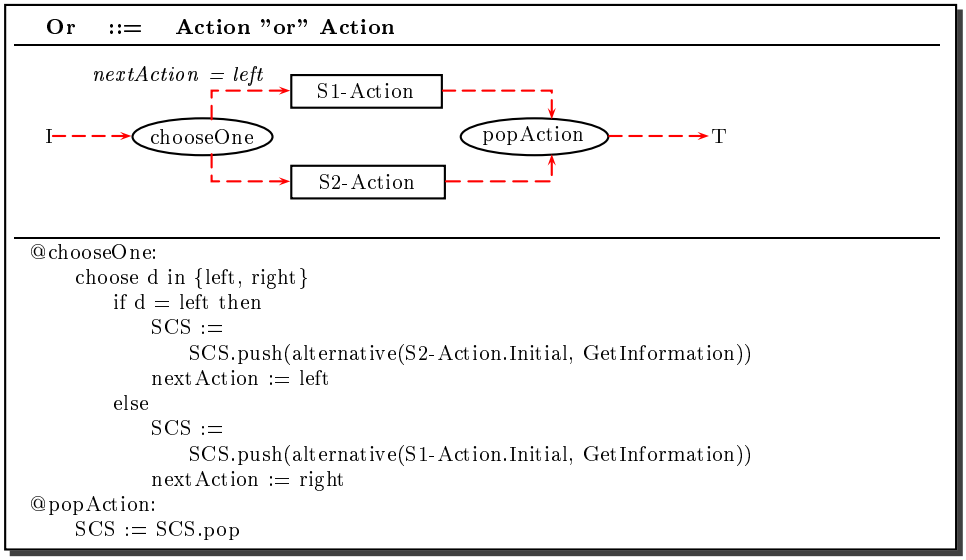

Fig. 15. The Or Montage

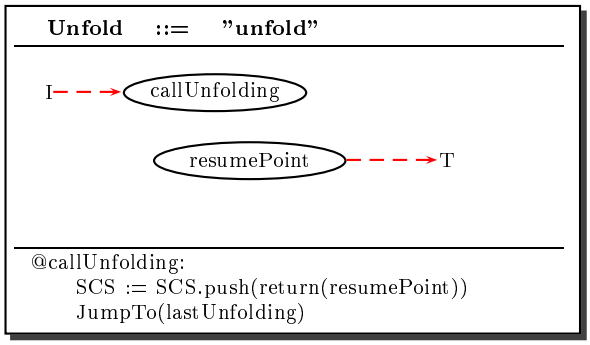

Fig. 13. The Unfold Montage

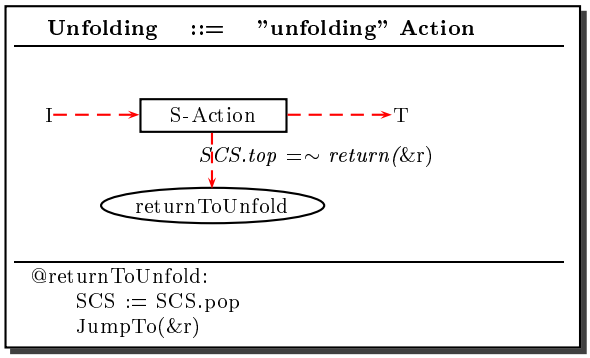

Fig. 14. The Unfolding Montage

Or, Fail, and Commit The action *A1 or A2* represents an implementation-dependent choice between two alternative actions *A1* and *A2*. If the alternative currently being performed fails, then it is abandoned and if possible, the other alternative is performed.

As shown in Figure 15, the first action node in the Or Montage nondeterministically chooses among the constants *left* and *right*. If the constant *left* is chosen the alternative right branch *S2-Action.Initial* is pushed onto the stack together with the current information as

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

where *alternative(_, _)* is a constructor. The control is then passed to the left branch *S1-Action.Initial*. Alternatively, if the chosen constant is *right* then the left branch is pushed onto the stack and the control is passed to the right branch.

If the chosen branch terminates normally the second action node called *popAction* discards the alternative originally pushed onto the stack. Otherwise, the *Fail* action can be used to in the execute the alternative branch.

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

```
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
    stdout := "the program failed.."
```

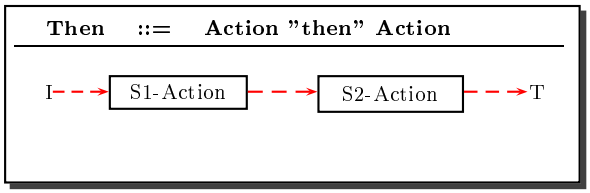

Fig. 16. The montage for the Then constructor

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

The primitive action *Commit* replaces all *alternatives* on the stack with *failed* so that no backtracking is possible anymore. The combination of *Or*, *Fail*, and *Commit* is used for giving semantics of logic programming languages like Prolog.

The functional facet is used to pass transient information between actions. In our imperative setting, we model transient information using a global variable called *Transient*. The abstract function *GetInformation* is therefore defined as *Transient* and *SetInformation(v)* as *Transient := v*. *CombineInformation(_,_)* returns the tuple formed from its two arguements. As an example we next describe the action combinator *then*.

Then action combinator The definition of the *Then* action combinator is given by the Montage shown in Figure 16. The control flow graph of this Montage graphically defines the left action (accessible with the selector *S1-Action*) to be the *initial* and the right action (accessible with *S2-Action*) to be the *terminal*, and therefore defines a left-to-right sequencing of the control flow, with transient information being passed from the left to the right action. Since this action combinator does not alter any information, no action nodes are required for this Montage.

Example 2. The following example calculates 2 to the power of 3. The action in the unfolding is given by two mutually exclusive alternatives and behave like an imperative *if-then-else* clause.

```
give 3 then
unfolding
( (check it is 0 then give 1)
or
```

```
  (check it is greater than 0 then
    give it - 1 then
    unfold then
    give it * 2
  )
)
```

The AST of this example is shown in Figure 17. In Figure 18 the nodes of this tree are decorated with the FSMs, and in Figure 19 the result of constructing the hierarchical FSM is depicted. Lastly, in Figure 20 the flattened version of the hierarchical FSM and the correspondance between the action nodes and AST nodes is shown.

4.4 The Imperative Facet

The imperative facet is the part of AN which can be specified most naturally with Montages, because of the state-based nature of the framework. The simplest model of imperative bahavior is a unary dynamic function *CurrentStore* as used in the Montage shown in the Figure 5. A basic update such as *CurrentStore(x) := y* is used to update the store at the position x to y. For specifying the imperative facet of AN we however need a more elaborate scheme than such basic updates.

In this subsection we give the specification of the imperative facet using an abstract data type *Map*. In contrast to the Montage in Figure 5, the *CurrentStore* in this case is a 0-ary function, ranging over the instances of *Map* as

$$CurrentStore :\rightarrow Map$$

The different operations involved are explained informally where needed, and in Section 4.5 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 component *Yielder*. If the result is of the specified sort then 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 into DN, a constructor *theStoredAt(_, _)* is introduced, and the field *data* is defined as explained in Section 4.2.

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

Here y ranges over the instances of *YieldTheStoredAt*.

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

$$\begin{aligned}&\text{Eval(theStoredAt}(s,\ y)) =\\ &\quad \text{let } r = \text{CurrentStore.lookUp}(y.\text{Eval}) \text{ in}\\ &\quad\quad \text{if } r \text{ of type Eval}(s) \text{ then } r \text{ else } nothing\end{aligned}$$

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 yielders such as

$$cell_0, cell_1, \ldots$$

which evaluate to instances of the 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*, and the definition of *data* is given by

$$y.data = currentStorage$$

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 Montage *StoreIn* 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.
We have therefore defined the following operations associated with the abstract data type *Map*.

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

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

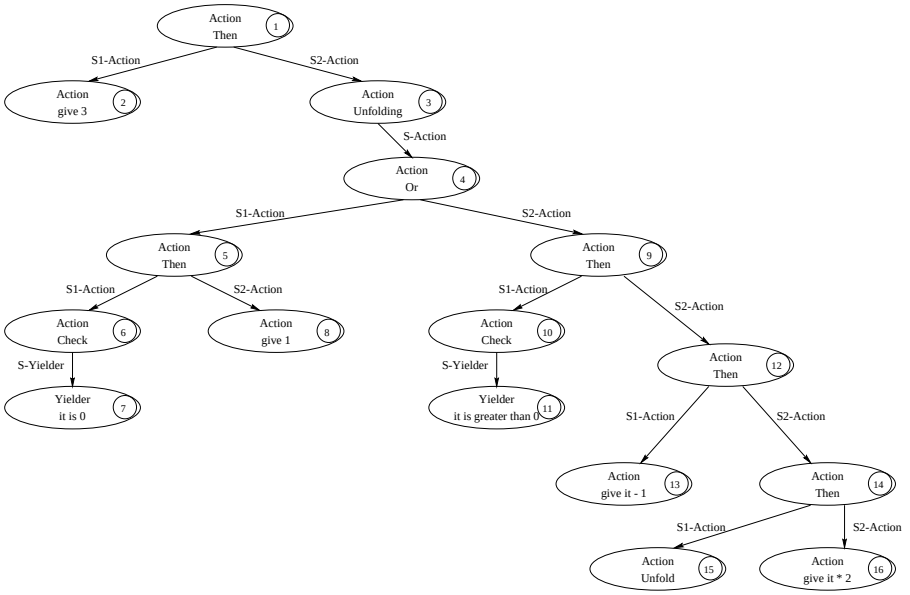

Fig. 17. The abstract syntax tree for Example 2

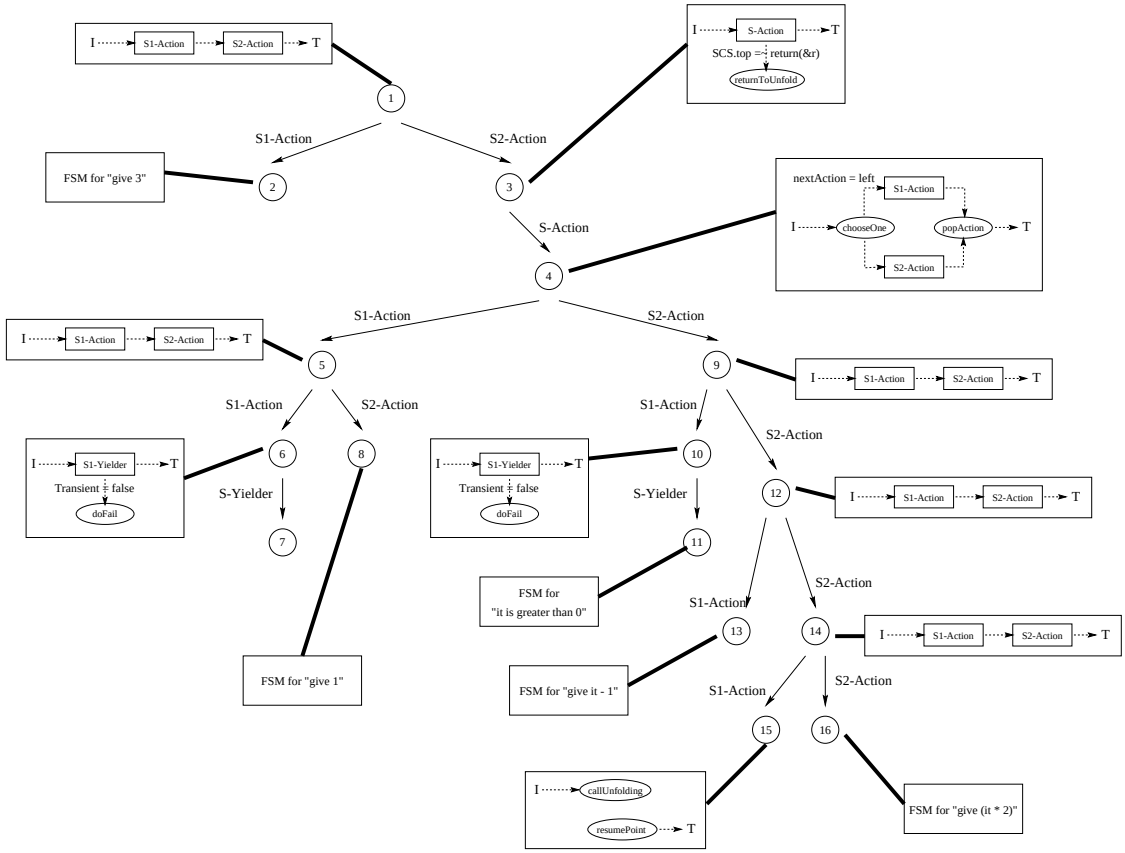

Fig. 18. The finite state machines belonging to the nodes of the AST

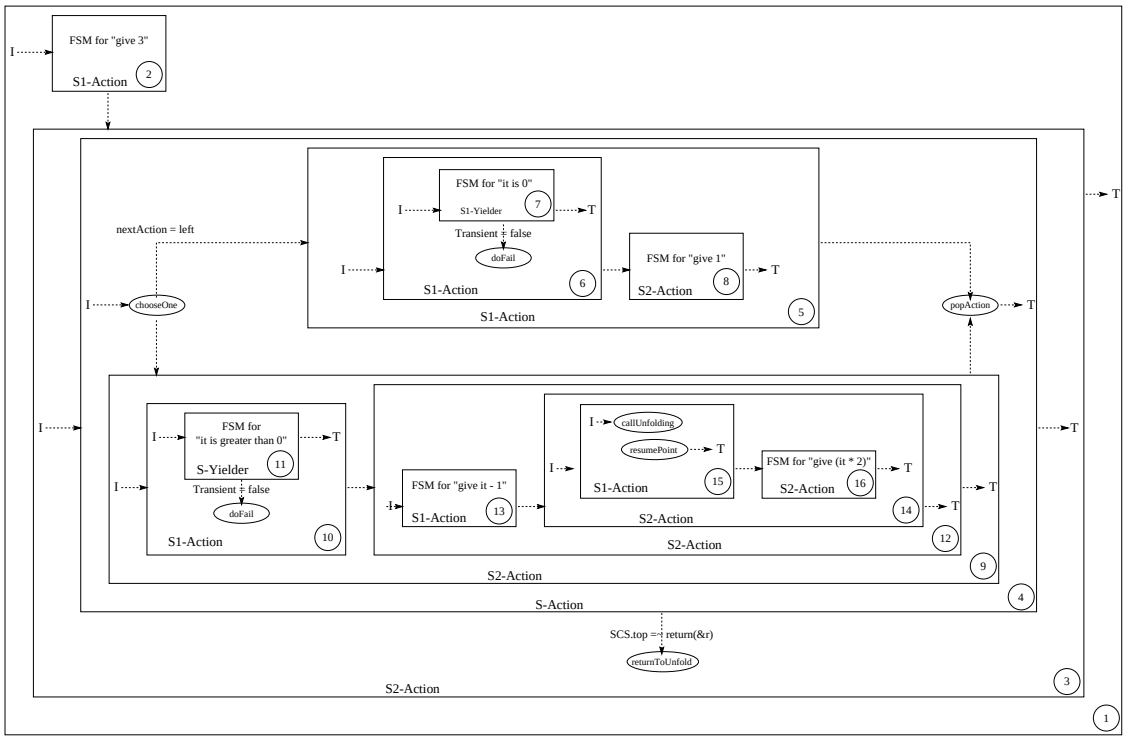

Fig. 19. The constructed hierarchical finite state machine

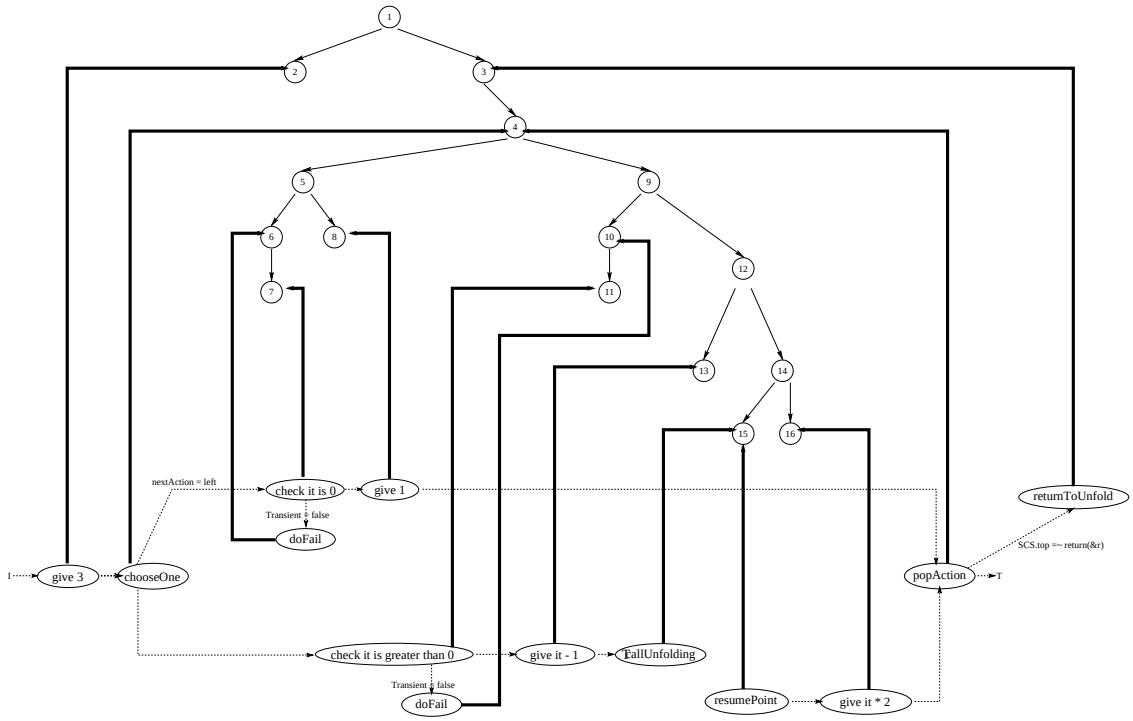

Fig. 20. The flat finite state machine and its relation to the AST

version where not only the first but also the third operation is implememnted using imperative techniques. Since both versions are given in the ASM framework, the correctness of the refinements can be proven, as shown for related examples in [39].

Example 3. 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 4. 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.

4.5 The ADT Map

The missing part for our model of the imperative facet is the definition of *Map*. We start with a simple definition that illustrates the use of ASMs and later present a more refined version. This later version illustrates how our approach helps in solving efficiency bottlenecks on the specification level which might result from a purely functional style where the complete state of the system has to be carried from one calculation step to the next.

Simple Definition of Map Both the simple and the refined definition of the ADT *Map* has an attribute *map*. In the imperative facet this attribute is used to map cells to values[7]. In the object oriented style of our ASM specification the signature of *Map* is given as

```
class Map  is
  attr map(_)
  method lookUp(_)
  method upDate(_,_)
  method getCopy
```

Initially the attribute *map* is undefined everywhere. Only the arity of the attributes are needed and no typing information is required. The *lookUp* and *upDate* operations are defined as follows.

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

```
method lookUp(x) is
  return map(x)

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

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

Consider the execution of Example 3 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 now be added to the simple definition as follows. A new *Map* is allocated and then the complete definition of *map* is copied:

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

This simple and abstract solution is usful while presenting a specification. However, because of its nature it makes the tool unusable for large AN descriptions using the imperative facet. Therefore in the next paragraph we propose an alternative refined definition of *Map*. A special property of this refinement is that we mix functional and imperative styles of specification.

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

$$lastMap : Map \rightarrow Map$$

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 the 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, thereby 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 given by

```
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
    return map(x)
  elseif lastMap != undef then
    return lastMap.lookUp(x)
  else
    return 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
  return 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 a copy has not 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
   return self
 else
   extend Map with m
     m.lastMap := self
     m.map(x) := y
     return m
```

Assume that we execute Example 4 using the refined definition. At the end of the execution three instances of *Map* exist, reflecting the state of the store after steps 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 Figure 21 the control flow of *StoreIn* therefore consists 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.

5 The Generated Environment

The development environment to support the Montages formalism is provided by the Gem-Mex tool suite [2,3].

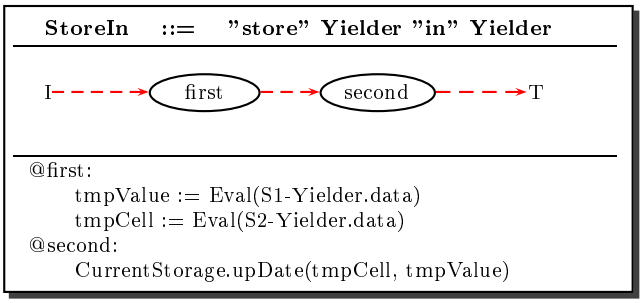

Fig. 21. The refined Montage for the StoreIn action

It assists the designer in a number of activities related to the language design process, and consists of the following interconnected components.

- The Graphical Editor for Montages (Gem) is a sophisticated graphical editor using which a specification can be entered into the tool and documentation be generated automatically from such specifications. Figure 22 shows the editor opened for the *Or* Montage.
- The Montages executable generator (Mex) automatically generates a correct and efficient implementations of the specified 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 this case the generated environment for AN has a different purpose compared that generated for any programming language. In particular, during the process of developing and extending the AN specification, it served as a testbed for empirically testing and validating the intended semantics. Additionally, it can be used to visualize and debug actions corresponding to programs in any language whose semantics have been specified using action semantics.

5.1 Generating 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 from the user. Gem-Mex system is *XASM*, which stands for e*X*tensible *ASM* language and provides a fully-fledged implementation of the ASM formalism along with extensions like classes and constructors. XASM can also be used as a stand-alone, 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 trans-

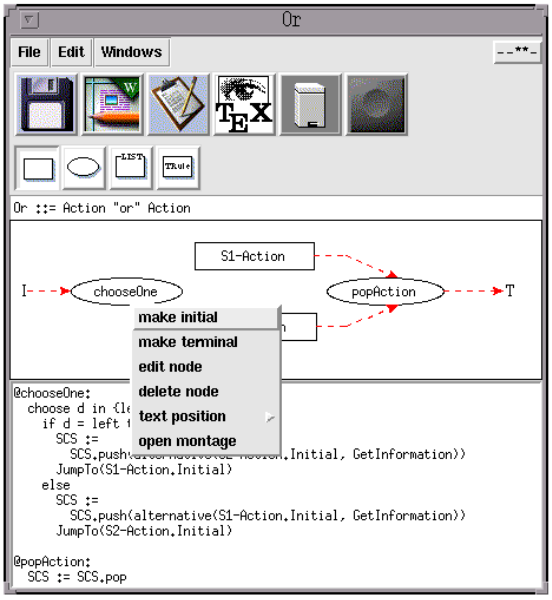

Fig. 22. The graphical editor of the Gem-Mex tool

lated to an ASM specification ("`montages2asm`" in Figure 23).

2. The resulting ASM description is processed by the XASM compiler, generating an executable which serves as an interpreter implementing the formal semantics description of the specified language.

The use of XASM as the core of the Gem-Mex system provides the user the possibility to exploit the full power of the ASM framework and enrich the graphical ASM macros provided by Montages with additional ASM code.

5.2 Generating Visual Programming Environments

Besides purely language interpreters, the Gem-Mex system is able to generate visual programming environments for the specified languages[8]. This is done by providing a generic debugging 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 necessary information required to visualize program executions. In particular, the visual environment can be used to debug a specification, animate the execution of programs, and generate documents representing snapshots of data structures built during the process of an execution. The debugging features include stepwise execution, textual representation of ASM data structures, definition of break points, interactive term evaluation, and replay of executions. Figure 24 shows an example, where

[8] This feature is available for kinds of ASM descriptions specified using XASM, and not only those generated from a Montages specification.

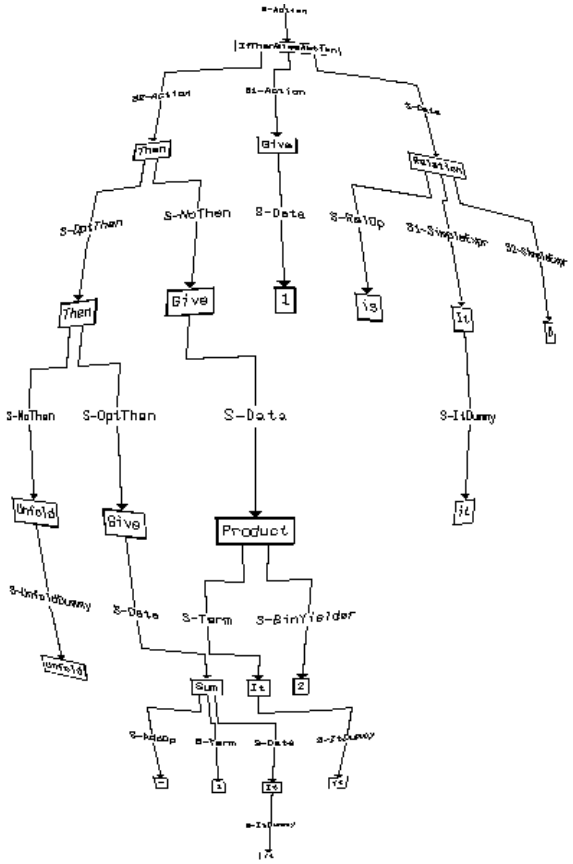

Fig. 26. Visualization of a portion of the abstract syntax tree of the example in Figure 25

the AN description of Example 2 is shown in the topmost window (entitled *view source code*). For this particular description, the universe *Unfolding* contains the node denoted by #225 (shown in the window entitled *Unfolding*, in which the node #29 corresponds to the XASM class definition of *Unfolding*). The selector function *S-ActionFactor* links the occurrence of the unfolding node with its action argument (node #224), which is highlighted in the text (in the window entitled *S-ActionFactor*).

Figure 25 shows an example of the graphical animation facility offered by the Gem-Mex system. On the right-hand-side of the window the AN program from Example 2 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 specification. In Figure 25, a change in the value of the "current-task" function CT is animated by drawing an arrow from *unfold* to its "replacement" given by the argument of *unfolding*. Our experiences have shown that this kind of animation often helps in debugging and understanding specifications.

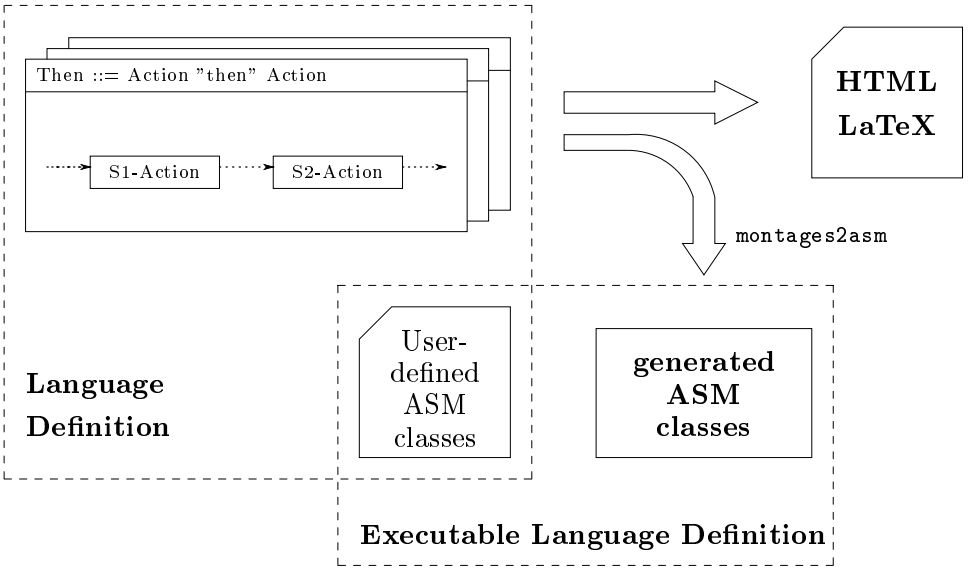

Fig. 23. The architecture of the Gem-Mex system

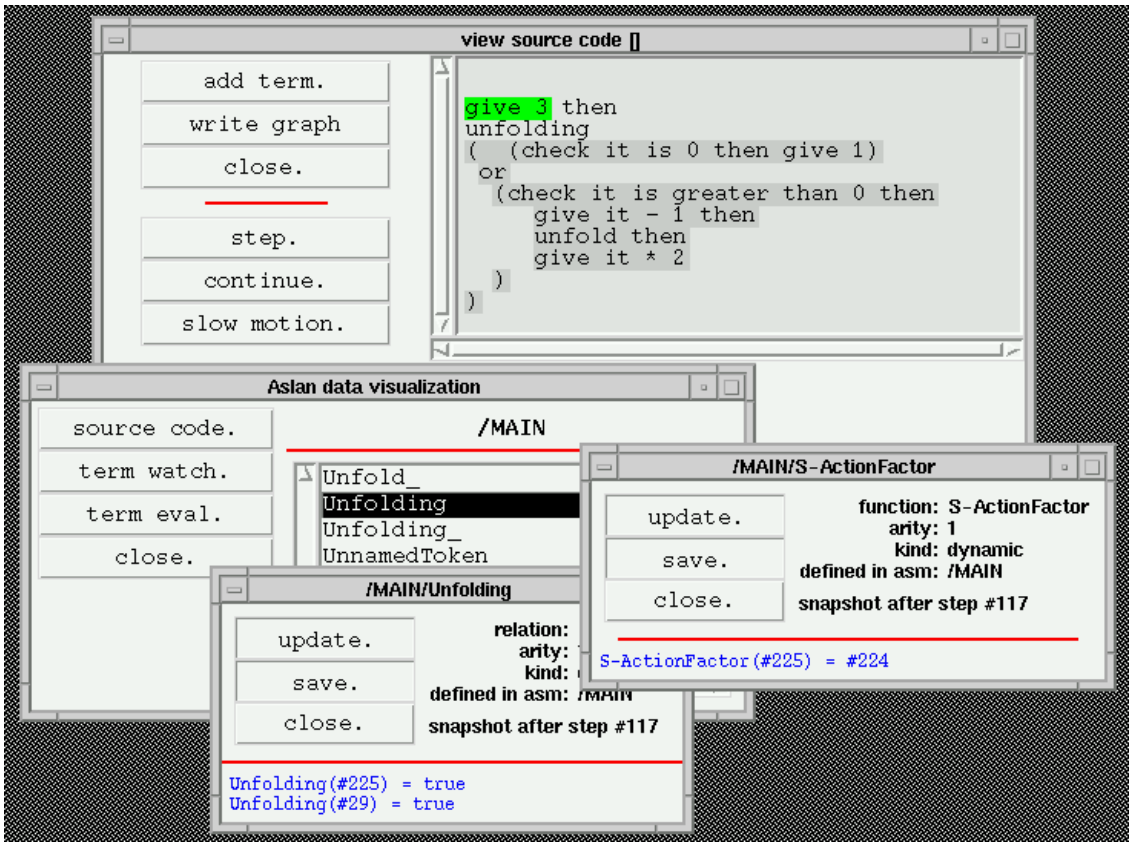

Fig. 24. Textual Visualization of data structures in the Gem-Mex system

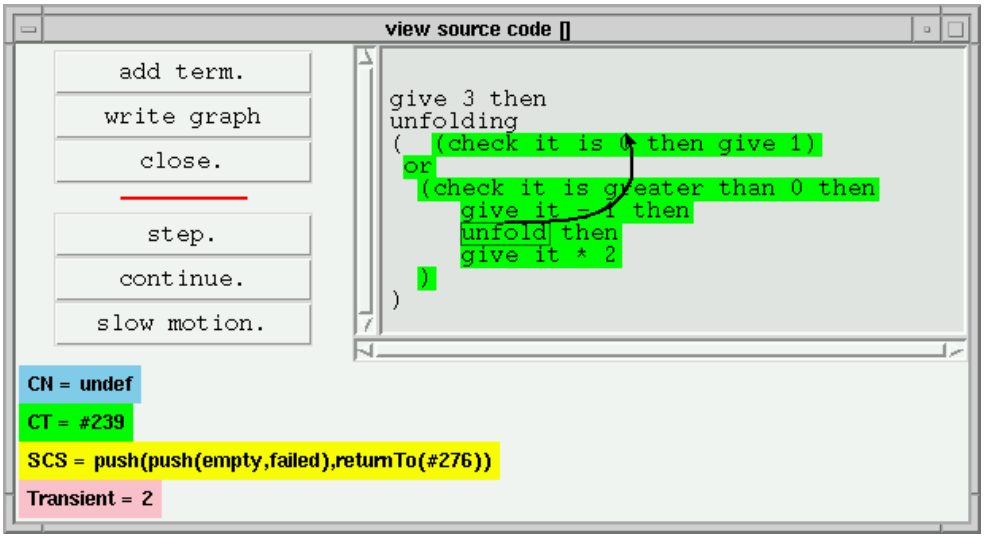

Fig. 25. Graphical animation in the Gem-Mex system

With the "write graph" button in Figure 25 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 [44] which can be used to visualize these data structures. As an example, Figure 26 displays a portion of the abstract syntax tree of the program displayed in Figure 25.

5.3 Generating Documentation Frames

As sketched in Figure 23 the Gem-Mex system also generates files that can be used as frames for documenting language specifications. Both paper, as well as online presentations of language specifications are automatically generated:

- LaTeX documents illustrate the Montages and the grammar of a specified language and such documents are easily customizable for non-specialists. All Montages in this paper were generated using Gem-Mex.
- An HTML version of a language specification generated using Gem-Mex allows one to conviniently browse through it.

5.4 Library of Programming Language Features

A concept of providing libraries of programming language features is currently under development. With this concept it will be possible to reuse programming language constructs that have already been specified using Montages. Examples of this kind of constructs may be arithmetic expressions, recursive function calls, various exception handling features, parameter passing techniques, standard control constructs, etc. The designer of a new language can then import such existing constructs and customize them according to the needs of this language. Such customization might range from a simple substitution of keywords, to a selection among a set of variants for a particular feature (like different kinds of inheritance mechanisms for an object-oriented language). This would allow, for instance, to embed the AN behavior in other languages, reuse parts of it, or extend it. In the Verifix project [22] a number of reusable Montages have been defined with the intention of reusing not only the Montages, but also an associated construction scheme for constructing correct compilers.

6 Concluding Remarks

In this paper we have presented a Montages semantics of action notation. This semantic description is executable and generates an action notation environment in which action denotations of programs can be executed and debugged. The suitability of using Montages for this hinges on the fact that an action is essentially a dynamic computational entity representing program behaviour, and therefore its semantics can be easily captured by an operational semantics formalism like Montages. Moreover, the modularity in Montages allows the semantic description to follow the classification of actions into facets, and therefore retains the modularity inherent in action notation. This is unlike the original SOS description of action notation by Mosses.

After the first submission of this paper, there has been some work on creating an executable environment for action semantics. This is based on the Maude Action Tool [8] which maps the recently defined Modular SOS of AN into rewriting logic. This allows the execution of programs based on their action semantics, and the mapping to rewriting logic opens up possibilities of reasoning formally about the properties of AN definitions. Here we

point out that there has been some recent work on formally reasoning about ASM specifications using model checking [45,53]. Since the underlying formalism in Montages is ASMs, the semantics of action notation in terms of Montages might enable the extension of these formal reasoning tools developed for ASMs to reason about action notation.

Full Montages descriptions of languages like C and Java (see [24] and [11] for reports) have been successfully entered in the Gem-Mex tool and executed, confirming the scalability of the tool. Furthermore, as already mentioned previously, Montages is also used in the Verifix project [15,22] as the front-end for generating verifiable compilers for realistic languages. Therefore we do not foresee any difficulties in using the generated action notation environment, as currently implemented, to execute action denotations of full-fledged programs in any realistic programming language.

Lastly, we point out that although the semantic description of action notation as described in this paper is intended to be equivalent to the original semantics given by Mosses in terms of SOS, this equivalence has not been formally proved. It would be interesting to see if any such equivalence proof can be formulated, especially in relation to the new modular SOS semantics of action notation.

References

1. M. Anlauff. XASM - An Extensible, Component-Based Abstract State Machines Language. In Gurevich et al. [21], pages 69–90.
2. M. Anlauff, P.W. Kutter, and A. Pierantonio. The Gem-Mex tool homepage. URL: `http://www.gem-mex.com`.
3. M. Anlauff, P.W. Kutter, and A. Pierantonio. Formal aspects of and development environments for Montages. In M. Sellink, editor, *Proc. 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer Verlag.
4. M. Anlauff, P.W. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjorner, M. Broy, and A.V. Zamulin, editors, *Perspective of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, pages 40 – 53. Springer Verlag, 1999.
5. M. Anlauff, P.W. Kutter, A. Pierantonio, and Asuman Sünbül. Using domain-specific languages for the realization of component composition. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *Lecture Notes in Computer Science*, pages 112 – 126, 2000.
6. M. Anlauff, P.W. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation environment from montages descriptions. In P.D. Mosses and D.A. Watt, editors, *2nd International Workshop on Action Semantics*, number NS-99-3 in Notes Series. BRICS, Dept. of Computer Science, 1999.
7. 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.
8. C. Braga, E.H. Haeusler, J. Meseguer, and P. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logics. In *Proc. 8th Intl. Conference on Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, 2000.
9. D. Brown, H. Moura, and D.A. Watt. Actress: an action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Proc. 4th International Conference on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 1992.
10. L.C. de Sousa Menezes and H.P. de Moura. The Abaco system: An algebraic based action compiler. In A.M. Haeberer, editor, *Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 527–529, Amazonia,Brazil, 1999. Springer Verlag.
11. Ch. Denzler and P.W. Kutter. An executable, animated, and modular version of the java 1.0 language specification. Technical Report 91, Institute TIK, 2000.
12. K.-G. Doh and D.A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, 19(4):213–233, 1993.
13. 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.
14. 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.
15. G. Goos and W. Zimmermann. ASMs and Verifying Compilers. In Gurevich et al. [21], pages 177–202.
16. 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.
17. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
18. Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department Technical Report, 1997.
19. Y. Gurevich. Sequential ASM Thesis. *Bulletin of European Association for Theoretical Computer Science*, (67):93–124, February 1999. Also Microsoft Research Technical Report No. MSR-TR-99-09.
20. Y. Gurevich and J. K. Huggins. The Semantics of the C Programming Language. In E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M.M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer Verlag, 1993.
21. Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.
22. 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.
23. J. Huggins. The Abstract State Machine Homepage at Michigan, URL: `http://www.eecs.umich.edu/gasm/`.

24. J.K. Huggins and W. Shen. The static and dynamic semantics of c: Preliminary version. Technical Report CPSC-1999-1, Computer Science Program, Kettering University, February 1999.
25. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
26. 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.
27. P. W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
28. P. W. Kutter, D. Schweizer, and L. Thiele. Integrating formal Domain-Specific Language design in the software life cycle. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 196 – 212. Springer Verlag, October 1998.
29. P.W. Kutter and A. Pierantonio. Montages: Specification of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416 – 442, 1997.
30. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in theoretical Computer Science. Cambridge University Press, 1992.
31. P.D. Mosses. Unified algebras and action semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science*, volume 349 of *Lecture Notes in Computer Science*, 1989.
32. P.D. Mosses. Unified algebras and institutions. In *Proc. 4th Annual Symposium on Logic in Computer Science*, pages 304–312. IEEE, 1989.
33. P.D. Mosses. Unified algebras and modules. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 329–343. ACM, 1989.
34. P.D. Mosses. Theory and practice of Action Semantics. In W. Penczek and A. Szalas, editors, *Proc. 21st Intl. Symp. on Mathematical Foundations of Computer Science (MFCS'96)*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61, Cracow, Poland, 1996. Springer Verlag.
35. P.D. Mosses. A tutorial on action semantics. In *Tutorial notes for FME'96: Formal Methods Europe*. Oxford, March 1996. Available from: `ftp.brics.dk/pub/BRICS/Projects/AS/Papers/Mosses96DRAFT/`.
36. P.D. Mosses. Foundations of Modular SOS (extended abstract). In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science (MFCS'99)*, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80, Szlarska-Poreba, Poland, 1999. Springer Verlag. The full version appears as Tech. Report RS-99-54, BRICS, Department of Computer Science, University of Aarhus.
37. P.D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Department of Computer Science, University of Aarhus, 1999.
38. M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon*. PhD thesis, ETH Zürich, 1989.
39. M. Odersky. Programming with variable functions. In *International Conference on Functional Programming*, Baltimore, 1998. ACM.
40. P. Ørbæk. OASIS: An optimizing action–based compiler generator. In P.A. Fritzson, editor, *CC'94, Proc. 5th Intl. Conf. on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15, Edinburgh, 1994. Springer Verlag.
41. J. Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proc. 4th IEEE International Conference on Computer Languages (ICCL'92)*, pages 117–126. IEEE, 1992.
42. J. Palsberg. A provably correct compiler generator. In *Proc. European Symposium on Programming (ESOP'92)*, volume 592 of *Lecture Notes in Computer Science*, pages 418–434. Springer Verlag, 1992.
43. G.D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Department of Computer Science, University of Aarhus, 1981.
44. G. Sanders. Graph layout through the VCG tool. In R. Tamassia and I.G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD'94, Proceedings*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205. Springer Verlag, 1995.
45. M. Spielmann. Model Checking Abstract State Machines and Beyond. In Gurevich et al. [21], pages 323–340.
46. J. Teich, P.W. Kutter, and R. Weper. Description and Simulation of Microprocessor Instruction Sets Using ASMs. In Gurevich et al. [21], pages 266–286.
47. A. van Deursen. *Executable Language Definitions – Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, September 1994.
48. A. van Deursen and P. D. Mosses. Executing Action semantics descriptions using ASF+SDF. In T. Rus and G. Scollo, editors, *Proc. 3rd Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing, pages 415–416. Springer Verlag, 1993.
49. A. van Deursen and P. D. Mosses. ASD: The Action semantic description tools. In M. Wirsing and M. Nivat, editors, *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 579 – 582, Munich, 1996. Springer Verlag.
50. 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.
51. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Proc. USENIX Conference on Domain-Specific Languages*, pages 157–170, Santa Barbara, California, October 1997.
52. D. A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall, 1991.
53. K. Winter. Methodology for Model Checking ASM: Lessons learned from the FLASH Case Study. In Gurevich et al. [21], pages 341–360.