Enhanced Control Flow Graphs in Montages

Matthias Anlauff[1], Philipp W. Kutter[2], and
Alfonso Pierantonio[3]

[1] GMD FIRST, D-12489 Berlin
ma@first.gmd.de
[2] Federal Institute of Technology, CH-8092 Zürich
kutter@tik.ee.ethz.ch
[3] Università di L'Aquila, I-67100 L'Aquila
alfonso@univaq.it

Abstract. A semi-visual framework for the specification of syntax and semantics of imperative programming languages, called Montages, was proposed in an earlier work by the authors. The primary aim of this formalism is to assist in recording the decisions taken by the designer during the language design process. The associated tool Gem-Mex allows the designer to maintain the specification and to inspect the semantics to verify whether the design decisions have been properly formalized.
Experience with full-scale case studies on Oberon, Java, and domain specific languages showed the close relationship to *Finite State Machines* (FSMs). This paper gives a new definition of Montages based on FSMs. It confers to the formalism enhanced pragmatic qualities, such as writability, extensibility, readability, and, in general, ease of maintenance.

1 Introduction

The aim of Montages is to document formally the decisions taken during the design process of realistic programming languages. Syntax, static and dynamic semantics are given in a uniform and coherent way by means of semi-visual descriptions. The static aspects of a language are diagrammatic descriptions of control flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals.

The departure point for our work has been the formal specification of the C language [10][1], which showed how the state-based formalism Abstract State Machines [8, 9, 13] (ASMs), formerly called Evolving Algebras, is well-suited for the formal description of the dynamic behavior of full-blown practical languages. In essence, ASMs constitute a formalism in which a state is updated in discrete time steps. Unlike most state-based systems, the state is given by an algebra that is a collection of functions and universes. The state transitions are given by rules that update functions pointwise and extend universes with new elements. The model presented in [10] describes the dynamic semantics of the C language by

[1] Historically the C case-study was preceded by work on Pascal [8], and other languages, see [5] for a commented bibliography on ASM case studies.

presuming on an explicit representation of *control and data flow as a graph*. This represents a major limitation for such a model, since the control and data flow graph is a crucial part of the specification. Therefore, we developed Montages which extend the approach in [10] by introducing a mapping which describes how to obtain the control and data flow graph starting from the abstract syntax tree.

The formulation of Montages [17] was strongly influenced by some case studies [16, 18] where the object–oriented language Oberon [26] has been specified. Montages have been used also in other case studies, such as the specification of the Java [25] language, the front-end for correct compiler construction [11], and the design and prototyping of a domain-specific languages in an industrial context [19]. The experience showed that the underlying model for the dynamic semantics, namely the specification of a control flow graph including conditional control flow and data flow arrows and its close relationship to the well known concept of *Finite State Machines*, shortens the learning curve considerably. In this paper a new FSM based definition of Montages is given. Complete references, documentation and tools can be obtained via [4].

2 Montages

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

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

The complete language specification is structured in specification modules, called Montages. Each Montage is a “BNF-extension-to-semantics” in the sense

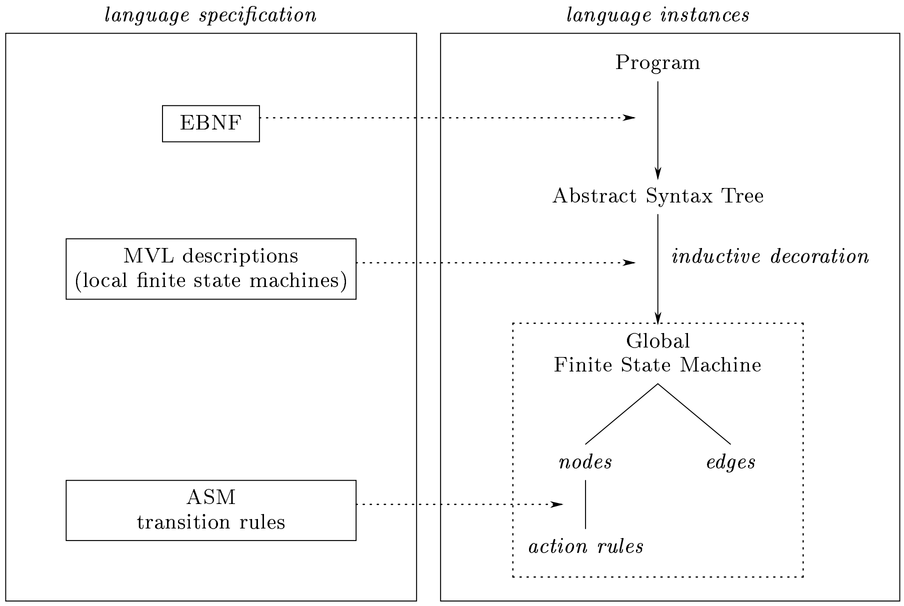

Fig. 1. Relationship between language specification and instances.

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

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

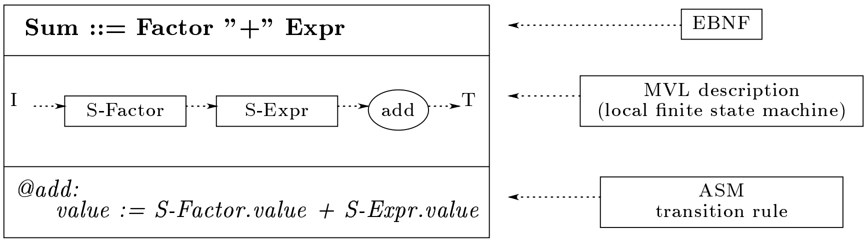

Fig. 2. Montage components.

The definition of Montages usually contains a fourth section which is devoted to the specification of static analysis and semantics. After working with fixed traversal orders and non-local attributions, we found that Reference Attribute Grammars [12] are most suited for our purpose, since they do not restrict the use of non-local references. The result of the attribution can be used to define firing conditions and actions in the FSM.

The use of attribute grammars for static analysis and semantics is a standard technique. In [12] it is shown how reference attribute grammars define static properties of an object oriented languages in a simple and concise way. Further [22] uses a corresponding functional system in combination with ASMs and shows how to describe static and dynamic aspects of full-blown languages. In contrast to these works, Montages has an elaborated visual formalism for the specification of sequential control flow by means of FSMs. These aspects are going to be presented in the next sections.

2.1 From Syntax to AST

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

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

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

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

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

Composition of Montages Each characteristic symbol and certain terminal symbols define a *Montage*. A Montage is considered to be a *class*[2] whose instances are associated to the corresponding nodes in the abstract syntax tree.

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

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

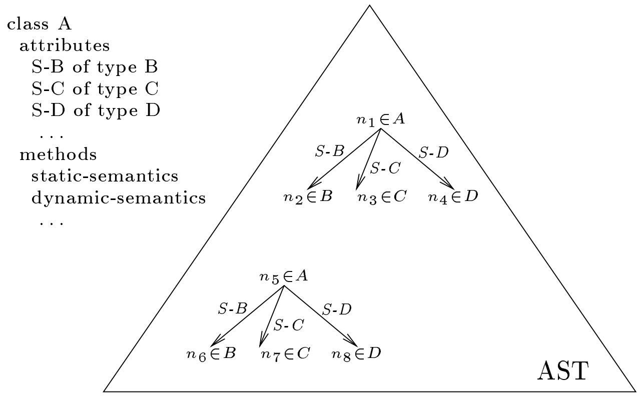

Fig. 3. Montage class A, instances in the AST, selectors S-B, S-C, S-D

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

Terminals, e.g. identifiers or numbers, do not correspond to Montages. The micro-syntax can be accessed using an attribute *Name* from the corresponding leaf node. The described treatment of characteristic and synonym productions allows for an automatic generation of AST from the concrete syntax given by EBNF, see also the work in [21].

Induced structures Inside a Montage class, the term *self* denotes the current instance of the class. Using the selectors, and knowledge about the AST, we can build paths w.r.t. to self. For instance, the path *self.S-B.S-H.S-J* denotes a node of class J, which can be reached by following the selectors S-B, S-H, and then

S-J, see Fig. 4. The use of such a path in a Montage definition imposes a number of constraints on the other EBNF rules of the language. The example *self.S-B.S-H.S-J* requires that there is a B component in the Montage containing the path. Further, every subtype of B must have an H component, and every subtype of H must have an J component. In other words, the path *self.S-B.S-H.S-J* must exist in all possible ASTs.

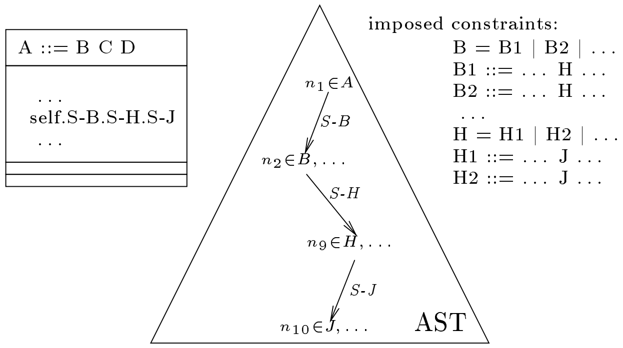

Fig. 4. Montage A using path self.S-B.S-H.S-J, situation in AST, and constraints on EBNF rules of B, H.

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

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

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

```
2 + x + 1
```

As a result of the generation of the AST we obtain the structure represented in Fig. 5. 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.

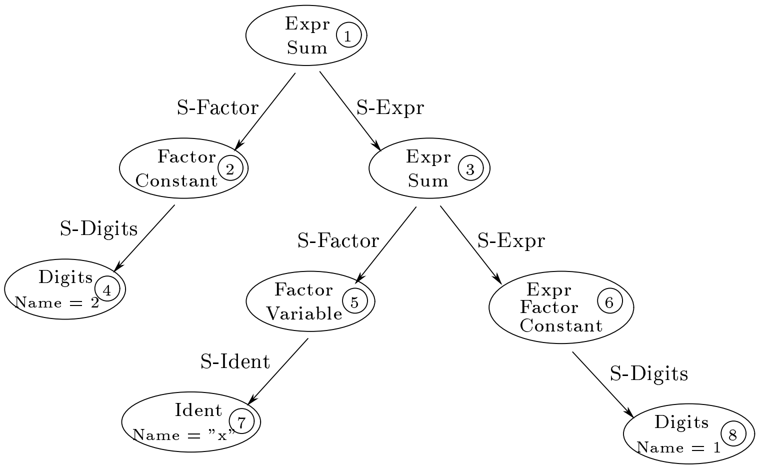

Fig. 5. The abstract syntax tree and composition of Montages for `2 + x + 1`

2.2 From AST to Control Flow Graphs

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

As we have seen in Fig. 2, the second part of a Montage contains the necessary specifications given in form of the *Montage Visual Language* (MVL). The Montages for the productions Variable and Constant are given in Fig. 6. Two kinds of information are represented in the second part of a Montage: (a) the local state machine to be associated to the node of the AST and (b) information on the embedding of this local state machine. Using our running example, Fig. 7 just represents the MVL sections of the Montages as they are associated to the corresponding nodes of the abstract syntax tree. The hierarchical state transition graph resulting from the inductive decoration is shown in Fig. 8 for the running example.

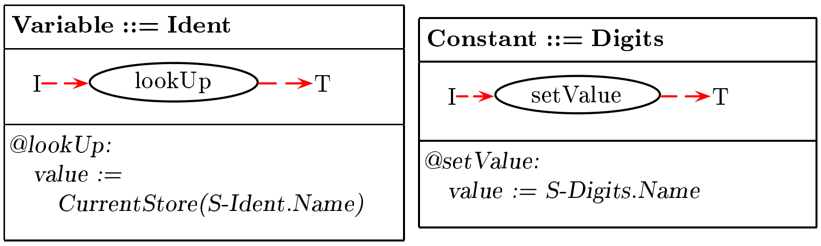

Fig. 6. The Montages for the language $\mathcal{S}$.

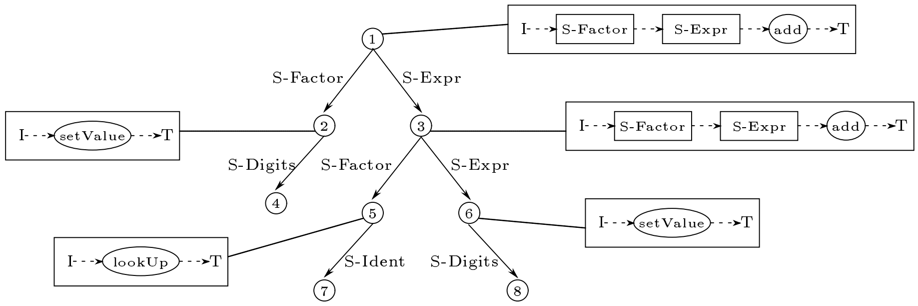

Fig. 7. The finite state machines belonging to the nodes.

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

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

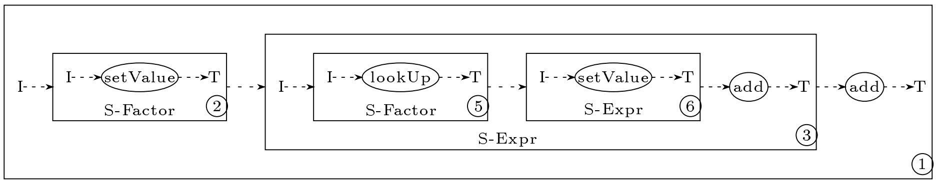

Fig. 8. The constructed hierarchical finite state machine.

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

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

The obtained hierarchical FSM gives the dynamic semantics of the parsed program. Direct execution of the hierarchical FSM is possible. Like in State-Charts a hierarchical state is entered at its initial state. This state is marked with an I-arrow. If a final state (marked with a T-arrow) is reached, the hierarchy is followed upwards.

Flat FSM Alternatively, the FSM can be flattened. For this purpose the arrows from I and to the T symbols define two unary functions, *Initial* and *Terminal* denoting for each node in the AST the first, respectively last state that is visited. According to the semantics of hierarchical FSMs, the inductive definition of these functions is given over the FSM states and the boxes representing instances of Montages.

For each state s in the finite state machines,

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

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

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

$$n.\mathit{Initial} = n.\mathit{tgt}.\mathit{Initial}$$

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

$$m.\mathit{Terminal} = m.\mathit{src}.\mathit{Terminal}$$

Using these definitions, the structured FSM is flattened by replacing each edge e from s to t with an edge from $s.Terminal$ to $t.Initial$. After this replacement the boxes are not related to the FSM anymore. Nevertheless, the information of the AST nodes related to source, target, and definition of an arrow are needed for the evaluation of advanced features usable in the the firing conditions.

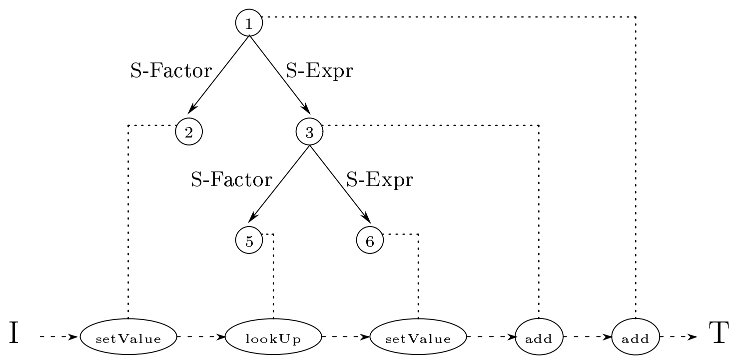

Fig. 9. The flat finite state machine and its relation to the AST.

Applying the replacement to the running example results in the flat state machine of Fig. 9. In the same figure the dotted lines denote the relation of a state to its corresponding Montage instance, which is accessible as *self*. The relation of edges to the instances corresponding to source, target and definition is not given, since our example has no interesting conditions.

Given the flat FSM as in Fig. 9, we can track the behavior given by the actions in Figs. 2 and 6. The initial state is the leftmost *setValue* state. Its rule updates the additional attribute *value* with the constant stored in the field *Name* of the digits-token. After the action rule is executed, the firing conditions of outgoing arrows are evaluated. In our example there is only one arrow with the default condition *true* exists. Visiting the states sequentially the action rules are executed one after another. The most interesting is the *add* action. It accesses the values of its arguments using the selectors *S-Factor*, *S-Expr* 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.

3 Gem-Mex: The Development Environment for Montages

The development environment for Montages is given by the Gem-Mex tool [2, 3]. The intended use of the tool Gem-Mex is, on one hand to allow the designer to 'debug' her/his semantics descriptions by empirical testing of whether the intended decisions have been properly formalized; on the other hand, to automatically generate a correct (prototype) implementation of programming languages from the description, including visualization and debugging facilities.

Gem-Mex is a system which assists the designer in a number of activities related with the language life cycle, from early design to routine programmer usage. It consists of a number of interconnected components

- a specialized graphical editor allows to enter and manipulate Montages in a convenient way;
- frames for the documentation of the specified languages are generated automatically;
- the Montages executable generator (Mex) generates a correct and efficient interpreter of the language;
- the generic animation and debugger tool visualizes the static and dynamic behavior of the specified language at a symbolic level; source programs written in the specified language and user-defined data structures can be animated and inspected in a visual environment.

3.1 Generation of Language Interpreters

Using the formal semantics description given by the set of Montages and a number of ADTs, the Gem-Mex system generates an interpreter for the specified language. The core of the Gem-Mex system is *Aslan* [1], which stands for *A*bstract *S*tate Machine *Lan*guage and provides a fully-fledged implementation of the ASM approach. Aslan can also be used as a stand-alone, general purpose ASM implementation. The process of generating an executable interpreter consists of two phases:

- The Montages containing the language definition are transformed to an intermediate format and then translated to an ASM formalization according to the rules presented in the previous Sections.
- The resulting ASM formalization is processed by the Aslan compiler generating an executable version of the formalization, which represents an interpreter implementing the formal semantics description of the specified language.

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

3.2 Generation of Visual Programming Environments

Besides pure language interpreters, the Gem-Mex system is able to generate visual programming environments for the generated ASM formalization of the programming language semantics[3]. 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 information being necessary to visualize the execution of the formalization. In particular, the visual environment can be used to debug the specification, animate the execution of it, and generate documents representing snapshots of the visualization of data structures during the execution. The debugging features include stepwise execution, textual representation of ASM data structures, definition of break points, interactive term evaluation, and re-play of executions.

3.3 Library of Programming Language Features

A concept for providing libraries of programming language features is currently under development. With this concept is shall be possible, to reuse features of programming languages that have already been specified in other Montages. Examples for this kind of features are arithmetic expressions, recursive function call, exception handling, parameter passing techniques, standard control features etc. The designer of a new language can then import such a feature and customize it according to his or her needs. The customization may range from the substitution of keywords up to the selection among a set of variants for a certain feature, like different kinds of inheritance in object-oriented languages, for example. In the Verifix project [11], a number of reusable Montages has been defined with the intention to reuse not only the Montages but as well an associated construction scheme for correct compilers.

4 Related Work

Denotational semantics has been regarded as the most promising approach for the semantic description of programming languages. But its problems with the pragmatics have been discovered already in case studies of the scale of Pascal and C [23]. Moreover domain definitions often need to be changed when extending the language with unforeseen constructs, for instance a change from the direct style to the continuation style when adding *gotos* [20].

Other well known meta–languages for specifying languages are Natural Semantics [14], ASF+SDF [24], and Action Semantics [20]. For somebody knowing mathematical logic, Natural Semantics are pretty intuitive and we used it for the dynamic semantics of Oberon [15]. Although we succeeded due to the excellent tool support by Centaur [7], the result was much longer and more complex

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

then the Montages counterpart given in [18], since one has to carry around all the state information in the case of Natural Semantics. Similar problems exist if ASF+SDF is applied to imperative languages. Action Semantics solves these problems by providing standard solutions to the main concepts used in programming languages. Unfortunately the set of standard solutions is not easily extendible.

Using ASMs for dynamic semantics, the work in [22] defines a framework comparable to ours. For the static part, it proposes *occurrence algebras* which integrate term algebras and context free grammars by providing terms for all nodes of all possible derivation trees. This allows such an approach to define all static aspects of the language in a functional algebraic system. Since reference attribute grammars [12] correspond to occurrence algebras the static aspects of our formalisms are almost identical to those in [22].

None of the discussed approaches uses visual descriptions of control flow and none of them supports structuring of all specification aspects in a vertical way, e.g. in self–contained modules for each language construct. This way of structuring is novel with respect to existing frameworks, as far as we know. In combination with refinements of involved semantic functions, and renaming of the vocabulary, it allows to reuse large parts of language specifications directly in other specifications. Programming language specifications can be presented as a series of sub-languages, each reusing its predecessor and extending it with new features. This specification structure has been used in ASM case studies [6, 10] and was adapted to the Montages case study of Oberon [18]. Our experience with Montages shows, that such sub-languages are useful, working languages, that can be executed, tested, and explained to the user in order to facilitate understanding of the whole language. The design and prototyping of a language is much more productive if such a stepwise development and testing is possible.

Acknowledgments We would like to thank S. Chakraborty, C. Denzler, B. Di Franco, W. Shen, L. Thiele, and C. Wallace for collaboration in the Montages project. Furthermore we thank G. Goos, A. Heberle, W. Löwe, and W. Zimmermann for the helpful discussions on the topic.

References

1. M. Anlauff. The Aslan Language Manual. Part of the Aslan distribution.
2. M Anlauff, P. W. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd Int. Workshop on the Theory and Practice of Alg. Spec.*, Workshops in Computing. Springer, 1997.
3. M. Anlauff, P. W. Kutter, and A. Pierantonio. The Gem-Mex Tool Homepage. `http://www.first.gmd.de/~ma/gem/`, 1997.
4. M. Anlauff, P. W. Kutter, and A. Pierantonio. The Montages Project Web Page. http://www.tik.ee.ethz.ch/~montages, 1997.
5. 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.

6. E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.
7. P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. Technical Report 777, INRIA, Sophia Antipolis, 1987.
8. 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.
9. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
10. Y. Gurevich and J. K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.
11. A Heberle, W. Löwe, and M. Trapp. Safe reuse of source to intermediate language compilations. 9th Int. Symposium on Software Reliability Engineering, 1998. http://chillarege.com/issre/fastabstracts/98417.html.
12. G. Hedin. Reference attribute grammars. In *Second Workshop on Attribute Grammars and Their Applications*, pages 153 – 172, 1999.
13. J. Huggins. Abstract State Machines Web Page . http://www.eecs.umich.edu/gasm.
14. G. Kahn. Natural Semantics. In *Proceedings of the Symp. on Theoretical Aspects of Computer Science, Passau, Germany*, 1987.
15. P. W. Kutter. Executable Specification of Oberon Using Natural Semantics. Term Work, ETH Zürich, implementation on the Centaur System [7], 1996.
16. 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.
17. P. W. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *JUCS, Springer*, 3(5):416–442, 1997.
18. P. W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *JUCS, Springer*, 3(5):443–503, 1997.
19. P. W. Kutter, D. Schweizer, and L. Thiele. Integrating Formal Domain-Specific Language Design in the Software Life Cycle. In *Current Trends in Applied Formal Methods*, LNCS. Springer, October 1998.
20. P. D. Mosses. Theory and Practice of Action Semantics. In *MFCS'96, 21st International Symposium*, volume 1113 of *LNCS*, pages 37–61. Springer Verlag, 1996.
21. M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon*. PhD thesis, ETH Zürich, 1989.
22. A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997. 1997.
23. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.
24. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping – An Algebraic Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
25. 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.
26. N. Wirth. The Programming Language Oberon. *Software - Practice and Experience*, 18(7):671 – 690, 1988.