Working Paper

Dynamic semantics for language-based programming environments revisited

Author(s):
Anlauff, Matthias; Chakraborty, Samarjit; Kutter, Philipp; Pierantonio, Alfonso; Thiele, Lothar

Publication Date:
1999

Permanent Link:
https://doi.org/10.3929/ethz-a-004287154 →

Dynamic Semantics for Language-Based Programming Environments Revisited

Matthias Anlauff*
GMD FIRST, Berlin

Samarjit Chakraborty†
ETH Zürich

Philipp W. Kutter‡
ETH Zürich

Alfonso Pierantonio§
Universitá di L'Aquila

Lothar Thiele¶
ETH Zürich

Abstract

Montages are a semi-visual language specification formalism recently proposed by some of the authors. This framework has been successfully used to give complete executable specifications of languages like C and Java. This paper compares Montages with action equations, both being paradigms for incorporating a means for specifying dynamic semantics in an environment generator using an attribute grammar framework. The major differences between the two lie in the way they specify the semantic processing at the nodes of the abstract syntax tree and how they specify control flow. Action equations use equations, similar in spirit to the semantic equations in attribute grammars, while Montages make use of Abstract State Machine rules to specify the semantic processing. Control flow in Montages is represented as state-transitions in a finite state machine which is specified using a graphical notation, while action equations specify control flow by propagating events from one node of the syntax tree to the other. Finally, the specifications of common control structures found in any imperative or object-oriented language are compared using the two formalisms and their relative merits are discussed.

1 Introduction

The mid and the late-eighties saw a proliferation of different programming environment generators, some of the best known among them being the Synthesizer Generator [37], Centaur [9], Pan [7], Mentor [12], PSG [6], IPSEN [14], Pecan [36], Mjolner [32], Yggdrasil [10], GIPE [21] and ASDL [27]. Very recently there has been some renewed interest in programming environment generators in the context of implementing Domain Specific Languages (DSLs). For example, the ASF+SDF Meta-Environment [26] has been used to successfully implement several DSLs being used in the industry [38, 39, 40]. [31] and [35] also report work in this direction. The flexibility associated with generating a language implementation from its specification result in significantly improving the ease in maintenance, which is important in the DSL context. In contrast to previous work on programming environment generators where the main focus was on the generation of a language-based editing system, current interests, however, are more related to issues like generating efficient compilers, interpreters, debuggers, and above all, ease in specification. Some of these tools can be generated only if the runtime behavior of a program is contained in the language description. As a result of this the specification of dynamic semantics has gained more importance than in the past.

Among the several mechanisms proposed for specifying programming environments, attribute grammar systems have been one of the most successful ones. The main reason for this lies in the fact that they can be written in a declarative style and are highly modular. However, by itself they are unsuitable for the specification of dynamic semantics. The work of Gail Kaiser on *action equations* (AE) [23, 24] addresses this problem by augmenting attribute grammars with mechanisms taken from *action routines* proposed by Medina-Mora in [33] for use in language based environments. Action routines are based on semantic routines used in compiler generation systems such as Yacc, in which the semantics processing is written as a set of routines in either a conventional programming language or a special language devised for this purpose [1]. Each node in the abstract syntax tree (AST) is associated with such actions and the execution of a construct is triggered by calling the corresponding action routine. In contrast to this, actions in AE are given by a set of rules similar in form to semantic equations of attribute grammars. Such equations are embedded into an event-driven architecture. Events occurring at any node of the AST activate the attached equations in the same sense in which in the action routines paradigm commands trigger the associated action routines. Equations which are not attached to any events correspond exactly to the semantic equations of attribute grammars. Equations in this framework can be of five types: assignments, constraints, conditionals, delays and propagates. Assignments and constraints are exactly similar in form, with the difference being that constraints are not attached to events and hence are active at all times. The propagate equations propagate an event from one node of the AST to another after evaluating the equations in that node. Thus the control flow is modeled by propagation of events from one node to the other.

This paper reevaluates the problem of specifying dynamic semantics in an attribute grammar framework for language defini-

*Mailing address: German National Research Center for Computer Science, GMD FIRST, Rudower Chaussee 5, D-12489 Berlin, Germany. email: ma@first.gmd.de

†Computer Engineering and Networks Laboratory, ETH Zürich. email: samarjit@tik.ee.ethz.ch Mailing address: Institut TIK, ETH Zentrum, Gloriastrasse 35, CH-8092, Zürich, Switzerland.

‡Computer Engineering and Networks Laboratory, ETH Zürich. email: kutter@tik.ee.ethz.ch

§Mailing address: L'ACS - Dip. Matematica Pura ed Applicata, Universit di L'Aquila, I-67100 L'Aquila, Italy. email: alfonso@univaq.it)

¶Computer Engineering and Networks Laboratory, ETH Zürich. email: thiele@tik.ee.ethz.ch

tions in an environment generator, by comparing AEs with Montages [28, 4], a language specification environment proposed recently by some of the authors.

Montages can as well be seen as a combination of Attribute Grammars and Action Routines. For giving the *actions*, Montages use Abstract State Machine (ASM) rules. ASMs where introduced by Gurevich in [16, 18] to formally specify algorithms on arbitrary abstraction levels. There exist a number of case studies applying ASMs to the specification of programming languages. In the case of imperative and object oriented languages, these applications work in the same way as Action Routine specifications, but they have a formal semantics. Besides being formal, those case studies are characterized by being compact, simple, and therefore easy to understand. They have been used to specify the dynamic semantics of a number of programming languages like Modula-2, Occam, C, C++, Oberon, and more recently Java (see [8] for a commented bibliography on ASM case-studies). However, all the ASM case-studies abstract away certain details of the language and also assume an explicit representation of the control- and data-flow graph. Montages adapt and integrate the ASM framework for specifying dynamic semantics with attribute grammars, and a visual notation for specifying control-flow as state transitions in a hierarchical finite state machine (FSM).

In short the differences between AE and Montages can be summarized as follows. In AE, the semantic processing at each node of the abstract syntax tree (AST) is given by sets of *equations* which are attached to particular events. The triggering of an event at a node leads to a reevaluation of these equations. Montages on the other hand uses ASM rules to specify such semantic processing, which is strictly different from the concept of using equations. Section 2 briefly introduces the basic concepts behind ASMs and examines how its rules differ from the equations used to describe the semantics processing in the nodes of the syntax tree in the case of AEs.

As a second difference, control flow in AEs is specified by *propagating* an event from a source to a destination node, thereby activating the equations associated with this event in the destination node. In contrast to this, control flow in Montages is specified by state transitions in a finite state machine, which is described using graphical notation. Section 3 describes the dynamic semantics specification in Montages and the visual notations used for control-flow, and explains the FSM interpretation of the visual descriptions. Section 4 contains a description of a number of different control-structures specified using Montages which are found in any imperative or object-oriented language. These are compared to the corresponding specifications written using AE. In one example (Example 6) we show a programming construct whose ASM-action cannot be given as AE equation and in other example (Example 3) we show that our visual notation makes it substantially easier to understand a specification. In the process of describing with Montages the control structures which were defined in [24] using AE, a small flaw was discovered in Example 3, which would have been hard to overlook in a graphical description.

In Section 5 we describe how the Gem-Mex tool [3] supports the Montages method. From a language specification it generates an environment consisting of a parser, a type-checker, an interpreter and a graphical debugging tool. The debugging tool is specialized towards co-animation of both the program under consideration and the Montages of the currently executed construct. Our tool was used to specify a number of languages like C [22], Oberon [29], Java [41] and for the design of a DSL in an industrial context [30]. We refer the interested reader to the Appendix for Montage describtions of the most commonly used programming language constructs supporting the imperative and object-oriented programming paradigm. Such specifications serve as well as language description front-end in Verifix [15, 19], an integrated environment for the generation of provably correct compilers. Section 6 concludes the paper.

2 Abstract State Machines

Over the last decade a number of algebraic formalisms has been proposed to cope with the problem of the specification of systems with dynamic behavior ([11, 5, 17] to mention a few). These works motivated the concept of Dynamic Abstract Data Types [13] where a notion of state is explicitly introduced as a class of algebras and operations which change the state are non-homomorphic algebra transformations. In other words as the state evolves a non-monotonic evolution takes place and certain properties (e.g. the validity of an equation) are not retained in the target state. Abstract State Machines are a Dynamic Abstract Data Type framework, where the fundamental concept is that functions are defined over a set $\mathcal{U}$, and can be changed point-wise. The set $\mathcal{U}$ referred to as the *superuniverse* in ASM parlance, always contains the distinct elements *true*, *false*, and *undef*. Apart from these $\mathcal{U}$ can contain numbers, strings, and possibly anything, depending on what is being modeled. In our context, i.e. describing the dynamic semantics of programming languages, we will always assume the existence of an abstract syntax tree (AST) T_0 and all the nodes of T_0 will be elements of $\mathcal{U}$.

Being slightly more formal, we define the *state* λ of a system as a mapping from a signature Σ (which is a collection of function symbols) to actual functions. We write f_λ for denoting the function which interprets 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. Details regarding the use of unary functions as attributes and the implementation choices available are discussed in [34].

Subsets of $\mathcal{U}$, called universes, are modeled by unary functions from $\mathcal{U}$ 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 might be a universe *Node* whose elements are the nodes of T_0. 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.

Example 1 Consider the following abstract syntax where the non-terminal `Sum` has two components, both of the type `EXPRESSION`, the non-terminal `Variable` has the component `name`, and `Constant` has the component `number`. Instances of an `EXPRESSION` can be a `Sum`, a `Variable`, or a `Constant`.

```
Sum      ::= left_expr:  EXPRESSION
             right_expr: EXPRESSION
Variable ::= name:       IDENT
Constant ::= number:     DIGITS
```

Following this abstract syntax, the term "2 + x + 1" can be mapped to the tree shown in Figure 1.

The superuniverse in this case, among other things, contains the nodes 1 through 5. A universe *Sum* contains the nodes 1 and 3, a universe *Constant* contains nodes 2 and 5, and lastly a universe *Variable* contains only node 4. The component `left_expr` of node 1 maps to node 2, and in node 3 it maps to node 4. Correspondingly, the mapping for `right_expr` is as shown in the figure. □

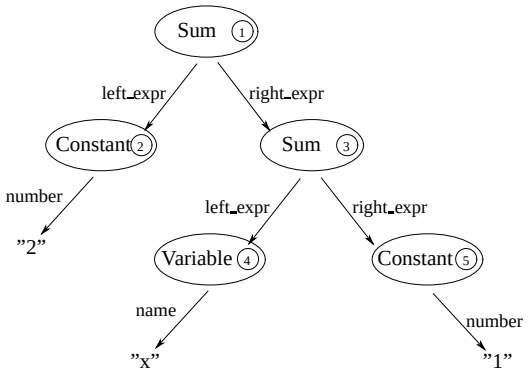

Figure 1: The abstract syntax tree for `2 + x + 1`

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 1, to hold the current value of an expression. The update of this attribute will be given by the rule

value := left_expr.value + right_expr.value

Formally, the 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 $\mathcal{U}$. 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.

Example 2 Let us assume that the *value* attribute of all the nodes of the syntax tree of Example 1 have been calculated, during a traversal of the tree. Now we give an ASM rule which generates an exception-object for each *Sum* node, when the result of the addition is larger than a constant `MAX_INTEGER`. The exceptions are grouped in a universe *Expection*.

```
do forall n in Sum
   if (n.value > MAX_INTEGER) then
      extend Exception with e
        e.kind := "overflow"
        e.reason := n
      endextend
   endif
enddo
```

This example illustrates the use of all the ASM rules described above and also shows how the value of the attribute *reason* can be a node in the AST. □

3 Specification of Control Flow

In this section the control flow specification of the two formalisms are compared to prepare for the examples described in Section 4. We consider only the application of the event-driven mechanism to situations with a single sequential thread of control. For such situations it is possible to represent the control flow as state transitions in a FSM.

3.1 Describing Control Flow in Montages

Traditional ASM specifications of programming languages typically use an abstract program counter that points to nodes of the AST and conditional rules trigger the right action depending on the type of the node. A number of ASM case-studies led to the conclusion that the transition rules of such models can be separated

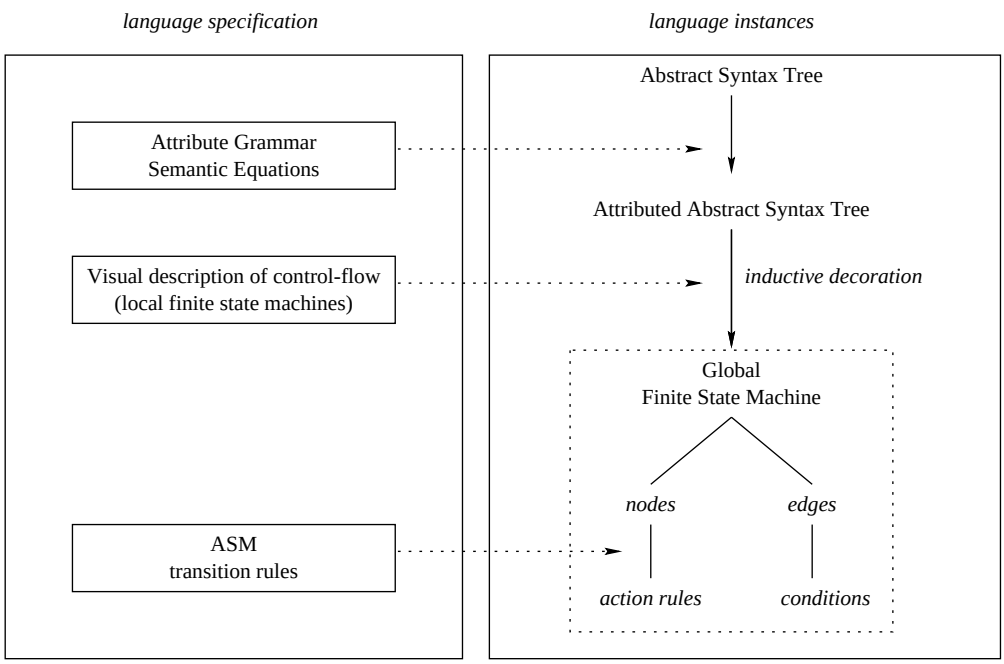

Figure 2: Relationship between language specification and instances.

into two parts. The first updates the abstract program counter and models the control flow, while the second is concerned with the remaining calculations, for example those related to updating variables. The first part can be very naturally as a FSM, whose states are associated with ASM rules that describe the computations corresponding to the second part. At any state of the FSM, the corresponding action rules are fired followed by a state transition to the next state. As in other state-based formalisms (such as Harel's Statecharts), the control arrows can be labeled with boolean predicates which determine the flow of control. Labels may be omitted. A control arrow without a label has a default-behavior, e.g. if no other arrow originating from its source has a label evaluating to true, then the unlabeled arrow gets active. This section shows how the Montages paradigm uses this view and structures a language specification.

A Montages specification consists of four parts: EBNF production rules, specification of static semantics, a visual notation for specifying control-flow, and the specification of dynamic semantics. The EBNF production rules are used to generate a parser, and then map in a canonical way a parsed program into an abstract syntax tree (AST). The details of this are not the concern of the present paper, so throughout this article only the abstract syntax is shown. Specifying static semantics using attribute grammars is a standard technique, and [20] explains how allowing attributes to be references to nodes in the syntax tree simplifies the description of problems with non-local dependencies. One example of this is the complex scoping rules for modular and object-oriented languages. The third part of a specification describes the control-flow in terms of state transitions in a finite state machine (FSM). A visual description is associated with each syntax rule which defines a *local* FSM and also specifies how this FSM can be plugged into a *global* FSM via an inductive decoration of the AST. Towards this end, each node of the AST is decorated with a copy of the FSM fragment given by its Montage. The references to descendents of a node defines an inductive construction of the global FSM. This is described in Section 3.1.1. Finally, the last part consists of ASM rules of the form described in Section 2, specifying the dynamic semantics. Any state of the FSM may be associated with an ASM rule, which is fired when this state is reached. The transitions in the FSM are conditional, the conditions being build up from attributes of the nodes of the AST. Figure 2 depicts a specification and its corresponding instantiation.

A complete language specification is structured into specification modules called Montages. Each Montage corresponding to a language construct describes the semantics of that construct. As an example, Figure 3 shows the Montage for the non-terminal *Sum*, whose abstract syntax was introduced in Example 1. This example is used in the later sections as well, and we refer to the expression language as L and the term "2 + x + 1", whose AST is shown in Figure 1, as P.

3.1.1 From an AST to a Control-Flow Graph

Figure 4 shows the Montages for the productions *Variable* and *Constant* of the language L. The visual notation in each Montage contains the information about the local FSM to be associated with each node of the AST corresponding to the Montage, and the information required to embed this FSM into the global FSM which corresponds to the input program. Figure 5 shows how the graphical fragments from each of the Montages are associated with the nodes of the AST corresponding to the program P, and Figure 6 shows the hierarchical state transition graph resulting from an inductive nesting of these fragments.

Visual Notation for Control-Flow The semantics of the graphical notation used in the Montages can be described as follows:

1. There are two kinds of nodes - ovals and boxes. The ovals represent the ordinary states of the FSM. They are also labeled with an action name, the action (specified using ASM rules) being fired when the state is reached. The boxes correspond to *superstates*, which are FSMs themselves. These are the FSMs corresponding to the components on the right-hand side of the abstract syntax rule. The definition of these FSMs are given in the Montages corresponding to the respective components.

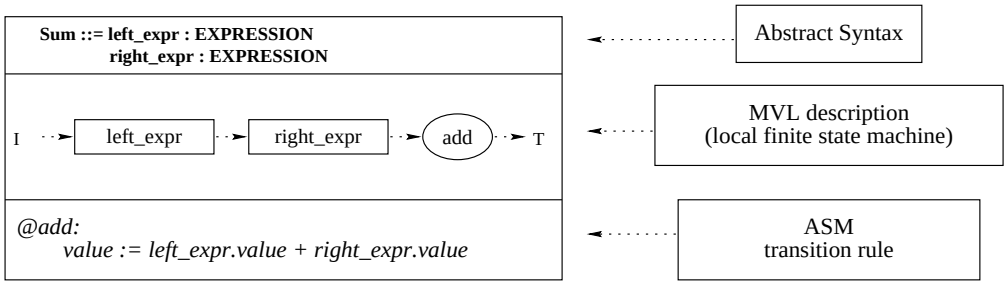

Figure 3: Montage for the *Sum* construct

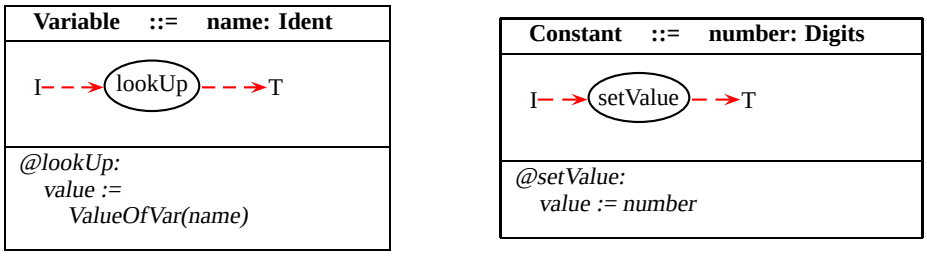

Figure 4: The Montages for the language $\mathcal{S}$.

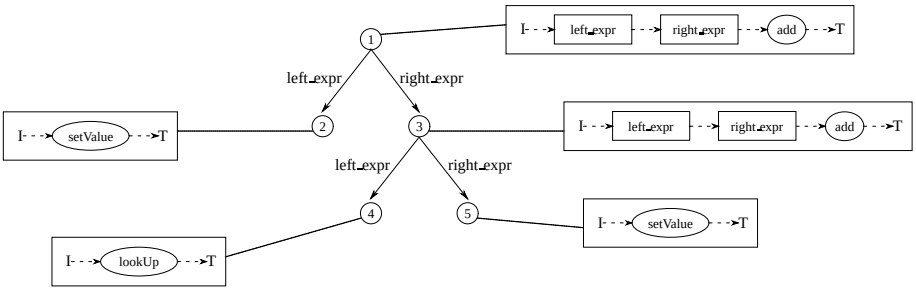

Figure 5: The finite state machines belonging to the nodes.

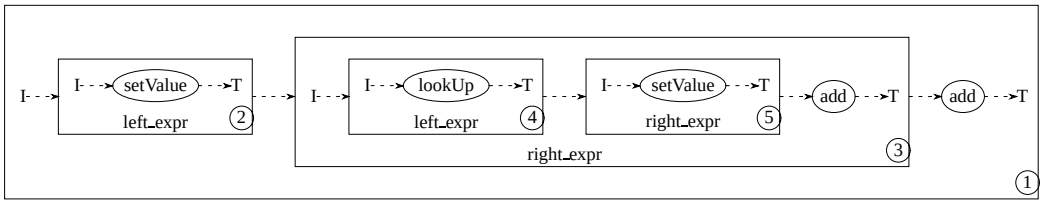

Figure 6: The constructed hierarchical finite state machine.

2. The arrows correspond to edges in the hierarchical state transition graph of the generated FSM. The source and the target of an arrow can be either a box or an oval. In addition to this, there are two arrows, one with its source marked as *I* (for Initial) and target as a box or an oval, and the other with its target marked as *T* (for Terminal) and its source as a box or an oval. The arrow marked with *I* indicates the entry into the local FSM and that marked with *T* indicates the exit from it. A transition to a superstate of the FSM results in an entry to the first state of the FSM (marked by the *I* arrow) which constitutes this superstate.

Hierarchical FSM The boxes in the visual description in each Montage are references to the corresponding state transition graphs. Resolving these references in the case of our example expression language described in Example 1 leads to the FSM shown in Figure 6. This hierarchical FSM gives the dynamic semantics of the program and a direct execution of this is possible. Like in Statecharts, a hierarchical state is entered at the initial state, which is indicated by an *I* arrow. If the final state (marked with an out-going *T* arrow) is reached, there is a transition to a state which is one level above in the hierarchy.

3.1.2 Executing the Dynamic Semantic Rules

Given the FSM shown in Figure 6, it is possible to trace the execution of our example program using the action rules given in the Montages in Figures 3 and 4. The initial state is the superstate $left\_expr$ (node 2), the entry into this state being marked by the *I* arrow in the graphical fragment attached to the root node of the AST shown in Figure 5. Since this is a superstate, it leads to a sequence of transitions through the states of the FSM which constitutes this superstate. In this case the only state inside this superstate is the simple state shown with the oval, which is labeled with the action *setValue*. This action is specified in the *Constant* Montage shown in Figure 4 and results in updating the *value* attribute of node 2 with the constant stored in the *Name* component. After this action rule is executed the conditions on the out-going arrows from this state is evaluated. In this example there is only one out-going arrow with the default condition *true*, and it results in a transition to the superstate marked with $right\_expr$ in the fragment attached to node 1 of the tree in Figure 5. Visiting all the states one by one in this way the action rules *lookUp*, *setValue*, *add*, followed by an *add* once again, are executed. The *add* action accesses the *value* attributes of the children of the *Sum* node and sets the *value* attribute of the node to their sum.

3.2 Control Flow in Action Equations

As described in Section 1, the AE paradigm is based on the concept of attaching a set of equations with non-terminals of the grammar, and thereby with the instances of the non-terminals as the nodes of the AST. The occurrence of an event at a node of the AST leads to an evaluation of the equations attached to that particular event in that node. Events, like attributes in attribute grammars, can be either synthesized or inherited. The events associated with the left-hand non-terminal of a production, as shown below, are synthesized.

production
  event$_1$ $\rightarrow$
    equation$_{1,1}$
    ...
    equation$_{1,m}$
  ...
  event$_p$ $\rightarrow$
    equation$_{p,1}$
    ...
    equation$_{p,q}$

Here $equation_{1,1}$ through $equation_{1,m}$ are attached to $event_1$, and similarly for the other events. Inherited events with their attached equations are associated with the right-hand non-terminals of a production. In [24] the left-hand non-terminal is referred to as the goal-symbol, the non-terminals on the right as the components of the goal symbol, and the context-free grammar notation is the same as that introduced in Example 1. Using this notation the inherited events are given as

goal symbol ::=
  component$_1$**: type**
  ...
  component$_n$**: type**

    event$_a$ **On component**$_1$ $\rightarrow$
      equations
    event$_b$ **On component**$_1$ $\rightarrow$
      equations
    ...

    event$_z$ **On component**$_n$ $\rightarrow$
      equations

The *On* keyword is used to denote that the inherited event is associated with the named component. It was also mentioned in Section 1 that the *propagate* equation is used to propagate an event from a source to a destination node of the AST. This has the effect of activating the equations at the destination node attached to the named event. Formally the equation is stated as

Propagate event To destination

Using these equations at each step of the computation, set of equations is dynamically determined and activated. The reevaluation of these equations results in the redefinition of a number of attributes. This redefinition of attributes is used for side-effects. The next Section shows the AE specifications for common control constructs and compares these with Montages specifications for the same constructs. Throughout the Section, sequential control flow is modeled with two kind of events, *Execute* and *Continue*.

4 Examples of Control Structures

Example 3 As first example how to model dynamic semantics with AE we take the if statement, as it is described in [24]. The *ifStm* has two children, the condition-part being an expression, and the thenpart, being a statement.

```
ifStm ::= condpart: EXPRESSION
          thenpart:  STATEMENT
```

When the *Execute* event occurs at an instance of *ifStm*, the *Execute* is propagated to the *condpart*.

```
Execute ->
  Propagate Execute To condpart
```

After any semantics processing involving the *condpart* are completed (including, for example, the setting of its *value* attribute), then the *condpart* propagates the *Continue* event to itself. A *Continue* on the *condpart* activates the following pair.

```
Continue On condpart ->
 If condpart.value
```

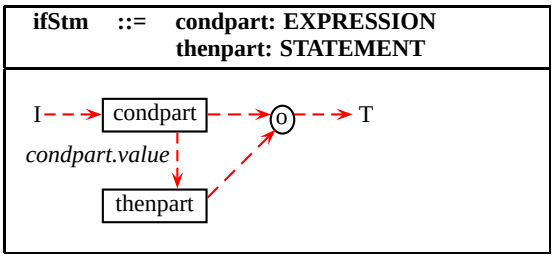

Figure 7: The ifStm Montage

```
Then Propagate Execute To thenpart
Else Propagate Continue To self
```

If the *value*-attribute evaluates to true, *Execute* is propagated to the *thenpart*. If not, the if statement has completed execution, and *Continue* is propagated to itself.

After the *thenpart* terminates, the *Continue* is correspondingly propagated to the ifStm.

```
Continue On thenpart ->
  Propagate Continue To self
```

Figure 7 we see how the same mechanism is given in terms of a FSM. It the *ifStm* is executed, the first visited state is the *condpart*. The semantics processing involving the *condpart* is given by the related FSM, whose actions set for instance its *value* attribute. The *condpart* has then two outgoing control edges along which the processing of the *ifStm* continues. One of the edges is labeled by

$$condpart.value$$

and the other has no label. In such cases, the non-labeled edge is assumed to represent the else-case, e.g. the case when all labels of other edges evaluate to true. Consequently, if the *condpart.value* is true, control continues to the *thenpart*, otherwise control leaves the *ifStm* through the terminal T. When the semantic processing of the *thenpart* terminates, control leaves the *ifStm* along the unique outgoing arrow.

The advantage to have an explicit visual representation of the control flow is that it is much easer to understand and validate the semantics of a construct like the *ifStm*. This is even indicated by the fact that while we entered the above example we found that the "Continue On thenpart" rule is missing in [24]. This rule corresponds to the unique outgoing arrow from the *thenpart*, and it the user would forget this arrow it would be immediately clear that something is missing.

Example 4 The following AE description gives the semantics of a lazy evaluated boolean and as available for instance in Pascal. The second operand must not be evaluated, if the first operand evaluates to false. This is important for the semantics, since expressions may have side effects. After the evaluation of the operands, the value is equal to the value of *operand2*, if the value of *operand1* is true, otherwise it is equal to false.

```
boolAnd ::= operand1: EXPRESSION
            operand2: EXPRESSION

  Execute ->
    Propagate Execute To operand1

  Continue On operand1 ->
    If operand1.value
```

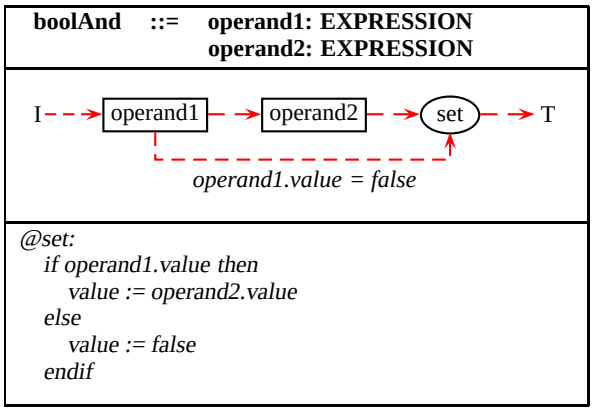

Figure 8: The boolAnd Montage

```
    Then Propagate Execute To operand2
    Else Propagate Continue To self

  Continue On operand2 ->
    Propagate Continue To self

  Continue ->
    If operand1.value
    Then value := operand2.value
    Else value := false
```

In Figure 8 we see the equivalent Montage. While the form of the value calculation remains the same, the visualization of the control flow shortens the length of the textual elements considerably.

Example 5 Another example is the following loop construct. After initialization, the control loops until the condition evaluates to false. In each cycle, the reinitialization is executed. While in Figure 9 the cyclic control structure is explicitly visible, in the following AE description it is encoded using the events.

```
loop ::= initialization:   STATEMENT
         condition:        EXPRESSION
         body:             STATEMENT
         reinitialization: STATEMENT

  Execute ->
    Propagate Execute To initialization

  Continue On initialization, reinitializa-
tion ->
    Propagate Execute To condition

  Continue On condition ->
    If condition.value
    Then Propagate Execute To body
    Else Propagate Continue To self

  Continue On Body ->
    Propagate Execute To reinitialization
```

Example 6 In a last example we consider a simple construct that repeats a statement n-times, where n is a constant, positive integer.

```
constRepeat ::= constant: DIGITS
                body:     STATEMENT
```

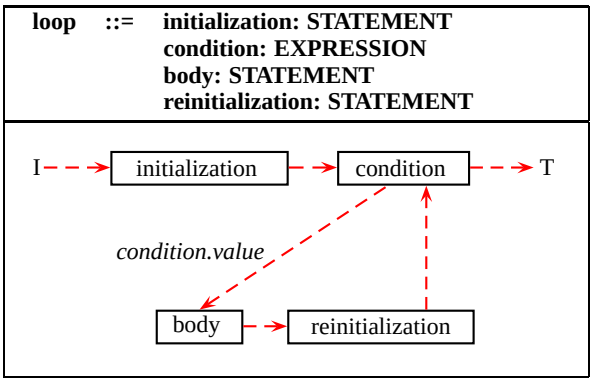

Figure 9: The loop Montage

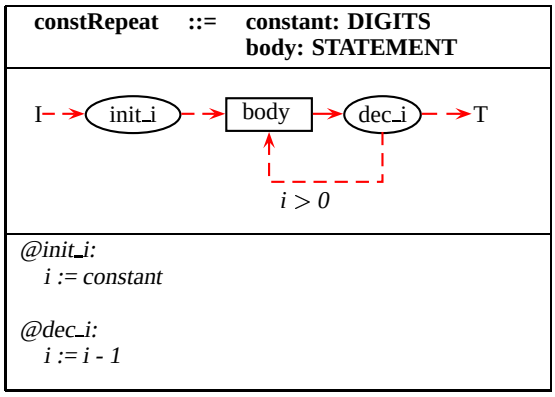

Figure 10: The constRepeat Montage

In a Montages specification we would introduce an attribute i, initialize it with *constant*, and after each time we executed the *body* we decrease the value of i by one. If after this i is still larger than 0, the body is reevaluated, else *constRepeat* terminates. In Figure 10 the complete Montage is given, using the name *init_i* and *dec_i* for the two states doing the initialization and the decreasing.

Naively one would model this in a similar way with AEs:

```
constRepeat ::= constant: DIGITS
                body:     STATEMENT
  Execute ->
    i := constant
    Propagate Execute To body

  Continue On body ->
    if (i - 1) > 0 then
       Propagate Execute To body
       i := i - 1
    else
       Propagate Continue To self
```

But using the AE framework, the formalization of

$$x = x - 1$$

is not possible with one equation. There is an intrinsic circular dependency in such an equation and the try to evaluate it would not lead to a solution.

The only possible solution is to introduce a help-attribute h, and to activate in a first step the equation

$$h = x - 1$$

and then in a next step to activate the equation

$$x = h$$

In order to introduce an intermediate step, one needs to introduce a new event *helpEvent*. Using this the complete AE solution is:

```
constRepeat ::= constant: DIGITS
                body:     STATEMENT
  Execute ->
    i := constant
    Propagate Execute To body

  Continue On body ->
    if (i - 1) > 0 then
       h := i - 1
       Propagate helpEvent To self
    else
       Propagate Continue To self

  helpEvent ->
     i := h
     Propagate Execute To body
```

This solution introduces an additional complexity which makes the designer's task more tedious and specifications more verbose, respectively. In this respect, being Montages based on ASM, which is a Dynamic Abstract Data Type framework, presents the advantage that one can express directly the following update

$$x := x - 1$$

requesting that the original value of the 0-ary function x can be discarded and replaced by a new one without an intermediate step, i.e. by means of a non homomorphic transformation of the algebra modeling the state before the modification.

5 Implementation

All the features of Montages described here are fully implemented in the Gem-Mex tool-suite [2]. This provides the user a graphical environment where he/she can enter a language specification. For the implementation of the tool-suite we use a special technique that guarantees the correctness of the implementation under certain conditions. In fact our approach splits the correctness problem of the implementation into two smaller sub-problems. First we define the meaning of a Montages in terms of an ASMs. To do this, we need to give the meaning of the syntax rules, semantic equations, finite state machines and actions in terms of ASMs, and we need to further define in terms of ASMs how these specification parts interact. In order to be able to do this we developed a dialect of ASMs that supports several additional features, allowing to incorporate the semantics of all components. Especially a component composition mechanism based on dynamic scoping is helpful in that respect. After this, we need to provide an efficient compiler for our ASM dialect. Given that the translation of semi-visual Montages to the textual ASMs is correct, and given that our ASM compiler is correct, we can guarantee the correctness of our implementation. During the development and for maintenance of the tool this shows to be a very useful property. Furthermore, we can further increase our confidence in the correctness, by improving the method of mapping from visual to textual description, and by further developing our ASM compiler technology. This improves the productiveness

with respect to the traditional way of implementing the tool for a new formalism from the scratch. Due to the state based nature of our formalism we do not experience any efficiency problem.

In [24] the same mechanism used for dynamic semantics are used for the definition of input/output, suspension/ continuation, single stepping, and tracing of programs. For that purpose they extend the syntax of the described language and introduce new events that correspond to the listed mechanisms. This approach is interesting, since it can be used not only for giving debugger functionality, but as well for dynamic semantic actions that involve the same kind of mechanisms. For instance the suspension/continuation can be used in simple mechanisms of distributed programming languages.

Concerning input/output, we can provide a seamless integration of ASM-rules with the existing input/output mechanisms of various operating systems. Input/output channels of the surrounding systems can be accessed like normal functions in ASMs, where input channels are read-only functions, and output-channels are write-only.

For providing debugger-functionality we have chosen to generate a specialized graphical environment, providing suspension/ continuation, single stepping, and tracing of programs. (rather than including it in the language spec). All this functionalities are accessible via a graphical user interface, and they can be used for any language specification without changing the specification. The explicit visualization of control flow is mapped in an intelligent way to the current program under examination. If control flows from one state to the next, along a control-edge, the tool animates this flow in two ways. First the program text-fragments corresponding to the source and the target state are colored, and the flow from the source text-fragment to the target text-fragment is indicated by a dynamically drawn arrow in the (original) program text. Second the Montage containing the definition of the corresponding control arrow is shown, and that arrow is highlighted as well. Like this, the user sees in parallel the visualization of control on the program under consideration, and the abstract, visual specification of the same situation in the Montage. The Montages serve thus not only as a means to enter visually the specification, but as well as a means to display visually the specification of the currently executed construct.

Single stepping and execution with variable speed are further variants how the described visual execution can be used. In addition it is possible to go not only forward in the control like this, but backwards as well, and if desired not only the current Montage can be displayed, but as well the previous and the next one.

For tracing the state, the user can open state-visualization windows, whose view on the state is freely customizable. For instance, he may trace the value of an invariant, that he expects to be true. He can as well set a break point, not statically assigned to a program point, but for instance to the event that the invariant does not hold true. Like this the system can be used to validate invariant information.

6 Conclusion

This paper compared two different paradigms which extend the attribute grammar framework in different ways, for the specification of dynamic semantics in a programming environment generator. Most of the previous work on environment generators were more concerned with the generation of a language-based editing system. The design of the AE paradigm followed this line, the main focus being incremental semantic processing during editing. In contrast to this, the Montages framework is concerned with the rapid prototyping of a language and focuses on issues like ease of specification.

It is understandable that the event oriented view is helpful and probably even necessary for the specification of a system which has to do some interactive processing. Apart from the *Execute* and the *Continue* events of AE described in this paper which models the control flow, other events arising from the functionality required in an editor include events like *Create*, *Delete*, *Clip*, etc. Although an editor is currently not generated in the Gem-Mex tool-suite for Montages, we do not foresee any difficulties in doing so.

The event-based framework of AE can result in triggering a set of rules from different nodes of the AST. As a result of this equations in different nodes can be active at the same time. Such a system is highly distributed and well suited for situations other than dynamic semantics of sequential languages. In this paper we consider only the application of the event-mechanism to situations with a single sequential tread of control. For these situations we are able to present the sequential control flow in terms of FSMs. For distributed situations FSMs would have to be replaced with PetriNets or StateCharts.

A concept for providing libraries of programming language features is currently under development. With this concept it shall be possible to reuse constructs/features of programming languages that have already been specified using Montages. Examples of this kind of features are arithmetic expressions, recursive function call, exception handling, parameter passing techniques, standard control features etc. The specifications of different languages can then import such a feature, and it can be customized according to specific needs. The customization may range from the substitution of keywords to the selection of a feature from a set of variants, like different kinds of inheritance mechanisms in object-oriented languages. We hope that the possibility to reuse a specification which captures a particular concept in two different languages can also be used to decide how closely related their semantics are. A further step in that direction will be the development of systems where software written in different languages can be combined and new languages can be designed and integrated in the development process.

Acknowledgments We would like to thank C. Denzler, W. Shen, 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 Attribute Grammars and the Verifix approach, and additional thanks are due to Yuri Gurevich for the numerous mails and discussions related to ASMs.

References

[1] V. Ambriola, G. E. Kaiser, and R. J. Ellison. An action routine model for ALOE. Technical Report CMU-CS-84-156, Department of Computer Science, Carnegie Mellon University, August 1984.

[2] M. Anlauff. The Gem-Mex Tool Homepage URL: `http://www.gem-mex.com`.

[3] M. Anlauff, P. Kutter, and A. Pierantonio. Formal aspects of and development environments for Montages. In *Second International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing. Springer-Verlag, 1997.

[4] M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In *Perspective of System Informatics*, number 1726 in LNCS, 1999.

[5] E. Astesiano and E. Zucca. D–oids: A Model for Dynamic Data Types. *Special Issue of MSCS*, 1994.

[6] R. Bahlke and G. Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467 – 479, October 1992.

[7] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on*

Software Engineering and Methodology, 1(1):95 – 127, January 1992.

[8] 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.

[9] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Proc. SIGSOFT 88: 3rd. Annual Symposium on Software Development Environments*, Boston, November 1988. ACM, New York.

[10] M. Caplinger. *A Single Intermediate Language for Programming Environments*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, 1985. Available as COMP TR85-28.

[11] P. Dauchy and M.C. Gaudel. Algebraic Specification with Implicit States. Technical report, Univ. Paris–Sud, 1994.

[12] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience. In D. R. Barstow, H. E. Shrobe, and E. Sandewell, editors, *Interactive Programming Environments*, chapter 7, pages 128 – 140. McGraw-Hill, New York, 1984.

[13] H. Ehrig and F. Orejas. Dynamic Abstract Data Types: An Informal Proposal. *Bull. EATCS*, (53), June 1994.

[14] G. Engels, C. Lewerentz, M. Nagl, W. Schafer, and A. Schurr. Building integrated software development environments Part I: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135 – 167, April 1992.

[15] W. Goerik, A. Dold, T. Gaul, G. Goos, F. Heberle, F. von Henke, U. Hoffmann, H. Langmaak, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In *International Conference on Compiler Construction*, number 1060 in LNCS, 1996.

[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. 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.

[19] A. Heberle, W. Löwe, and M. Trapp. Safe reuse of source to intermediate language compilations. Proceedings of the 9th International Symposium on Software Reliability Engineering. Fast Abstracts and Industrial Tracks.

[20] G. Hedin. Reference Attribute Grammars. To appear in *Informatica*. Preliminary version appeared in the Second Workshop on Attribute Grammars and their Applications, March 1999.

[21] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT'85: Status Report of Continuing Work, Part I*, pages 467 – 477. North-Holland, 1986.

[22] 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.

[23] G. E. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburg, Pennsylvania, May 1985.

[24] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169 – 193, April 1989.

[25] S. Kamin, editor. *Proc. First ACM SIGPLAN Workshop on Domain Specific Languages*, Paris, January 1997. Published as University of Illinois at Urbana Champaign Computer Science Report URL: `www-sal.cs.uiuc.edu/~kamin/dsl`.

[26] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[27] B. Kramer and H-W. Schmidt. Developing integrated environments with ASDL. *IEEE Software*, pages 98 – 107, January 1989.

[28] P. W. Kutter and A. Pierantonio. Montages: Specification of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416 – 442, 1997.

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

[30] 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*, number 1641 in Lecture Notes in Computer Science. Springer-Verlag, October 1998.

[31] J. Lampe. Depot4 - A generator for dynamically extensible translators. *Software - Concepts & Tools*, 19:97–108, 1998.

[32] B. Magnusson, M. Bengtsson, L-O. Dahlin, G. Fries, A. Gustavsson, G. Hedin, S. Minor, D. Oscarsson, and M. Taube. An overview of the Mjolner/ORM environment: Incremental language and software development. In *Proc. Second International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pages 635 – 646, Paris, June 1990.

[33] R. Medina-Mora. *Syntax-directed Editing: Towards Integrated Programming Environments*. PhD thesis, Carnegie Mellon University, March 1982. Tech. Rep. CMU-CS-82-113.

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

[35] P. Pfahler and U. Kastens. Language design and implementation by selection. In Kamin [25]. Published as University of Illinois at Urbana Champaign Computer Science Report URL: `www-sal.cs.uiuc.edu/~kamin/dsl`.

[36] S. P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276 – 285, March 1985.

[37] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1989.

[38] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In *Proc. AMAST'96, 5th International Conference on Algebraic Methodology and Software Technology*, Munich, Germany, July 1996. Springer-Verlag. Lecture Notes in Computer Science 1101.

[39] A. van Deursen and P. Klint. Little languages: Little maintenance ? In Kamin [25]. Published as University of Illinois at Urbana Champaign Computer Science Report URL: `www-sal.cs.uiuc.edu/~kamin/dsl`.

[40] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75 – 92, 1998.

[41] 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.

Appendix

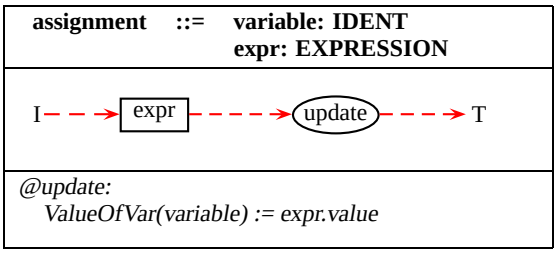

Figure 11: The assignment Montage

We have seen Montages for expressions and several control statements. Here we give additional examples of constructs typical for programming languages in the imperative and object oriented paradigm. In Figure 11 we see an assignment statement that works together with the variable Montage in Figure 4. While the action of the variable Montage reads the dynamic function *ValueOfVar*, the update-action of the assignment updates that function.

For the sake of simplicity we use just identifiers to refer to variables and procedures. In realistic scenarios the usual attribute techniques are used to determine use-definition relations, including the resolution of various forms of name scopes. Reference attribute grammars are especially useful for that task.

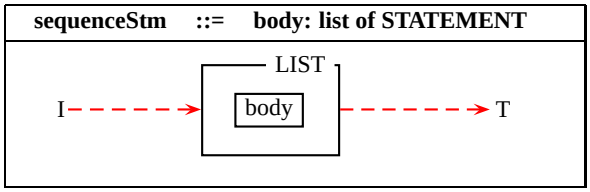

Figure 12: The sequenceStm Montages

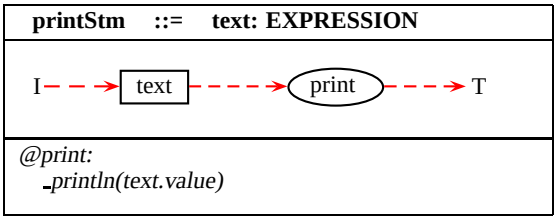

Figure 13: The printStm Montage

In Fig. 12 we show how sequential flow through a list of components is modeled in Montages. The print statement (Fig. 13) fires an action using the Aslan syntax for printing to the standard output. Its use is to test the behavior of the other statements.

Non Structural control flow We have shown how FSMs are constructed along the structure of the AST. For certain programming constructs, it is hard to model the control flow like this. These constructs include procedure calls, goto's, exceptions. In this Section we show how to construct FSMs for these constructs.

More technically, the problem is to define arrows of the FSM relating states that belong not to the same node in the AST. Our solution is to allow a reference from within a Montage to all instances of a certain type, for instance a reference to all instances of *EXPRESSION*. Visually such a reference to the instances of t is given by a box labeled by "ANY t", and we call them reference-boxes. A control arrow to or from a reference-box denotes a family of arrows to or from all corresponding instances. As in the case of boxes labeled with components, incoming arrows are connected with the initial state and outgoing arrows are connected with the terminal state.

As an example consider again the AST from Fig. 1. A square box labeled with "ANY EXPRESSION" would refer to all nodes, a square box labeled with "ANY Sum" would refer to nodes 1 and 3 and so on. In this constellation a control arrow into a "ANY Sum" reference-box would denote 2 control arrows ending in the initial states of the nodes 1 and 3, a control arrow departing from the same reference-box would denote 2 control arrows departing from the terminal states of nodes 1 and 3.

The conditions on a family of arrows typically depend on attributes of source and target nodes of each arrow. [1] The syntax to refer to these nodes is *src and* trg.

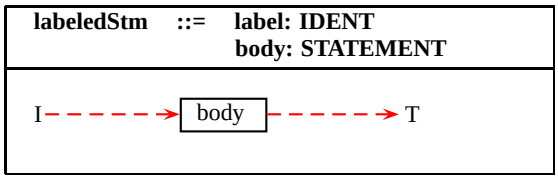

Figure 14: The labeledStm Montage

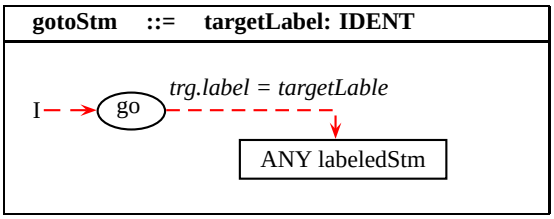

Figure 15: The gotoStm Montage

As a first example for non-structural control flow we give Montages for the labeled statement (Figure 14) and the goto statement (Figure 15). The unique component of a goto statement is its *targetLable*. The semantics of goto is to lead control directly to the instance of labeled-statement which has a *label*-component corresponding to its *targetLable*. Formally, the box labeled with "ANY labeledStm" is a reference to all instances of *labeledStm*. The edge from the go-state to the ANY-box denotes thus a family of FSM-edges from the "go" state going to the initial state of each labeled Statement, and the condition

$$trg.label = targetLabel$$

[1] Since attributes are associated with AST, we will provide means to reference the source and target AST-node, rather than the source and target state in the FSMs.

on the edges evaluates only to true if *targetLabel* and the *label* of the *labeledStm* match.

An example program in our language is

```
A: print 1;
   goto  B;
C: goto  A;
B: print 2;
   goto  C;
```

the corresponding FSM and AST is given in Fig. 16. The result of executing the FSM is the sequential printing of 1, 2, 1, 2, 1, 2, ...

Exception Throw and Catch For the specification of a simple exception mechanism, we assume that an attribute

$$leastEnclosingCatchStm$$

points from each node to the least enclosing instance of the *catchStm* production. The definition of such an attribute is standard technique in reference attribute grammar. Since such attributes are very often used, and since their definitions are typically spreading over several Montages, in Gem-Mex there are built-in attributes for referencing least enclosing instances of arbitrary productions.

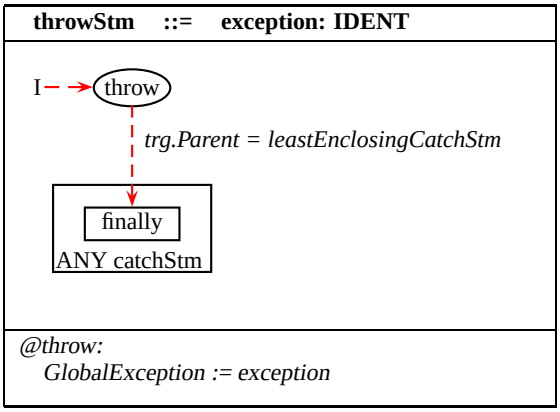

Figure 17: The throwStm Montage

The throw statement (Figure 17) sets the global nullary function *GlobalException* to its *exception*-component. For simplicity exceptions are just given by identifiers. Then control is lead to the least *finally*-component of the least enclosing catch statement. The nested *finally*-component inside the "ANY catchStm" reference-box is illustrating another feature of Montages, namely to refer not only to direct components, but as well to components of components, or as in this case to components of reference-boxes.

The *Parent* attribute is another built-in attribute pointing to the parent of each node. Thus the term *trg.Parent* in the control-edge from the *throw*-state to the *finally*-component of the reference box actually references to the instances of *catchStm*. Control is lead to those instance being equal to the least enclosing instance of *catchStm*.

The catch statement Montage is shown in Figure 18. Control enters first the *body* component, then the *finally* component, after this, the *catch* state is reseting *GlobalException* to *undef* if it matches with the *exception* component of the catch. If after the *catch*-state the *GlobalException* is not *undef*, control is lead to the *finally* component of the least enclosing instance of *catchStm*. Otherwise control leaves the construct normally.

A desired consequence of the given specification is, that the *finally*-part is always executed, even when an exception is raised.

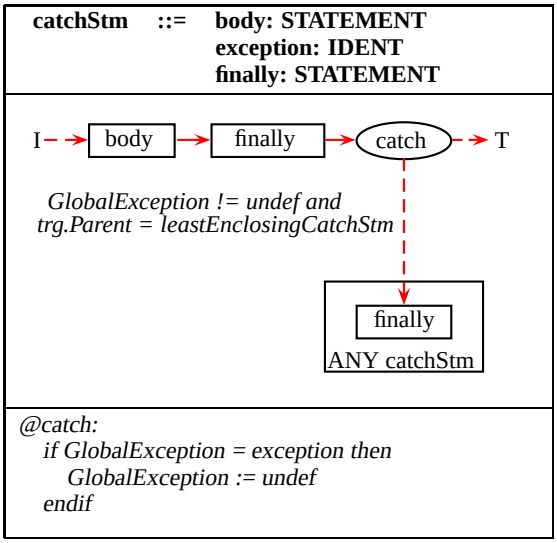

Figure 18: The catchStm Montage

Procedure Declaration and Call For the specification of procedure declaration and call we need a global nullary function

$$Frame$$

which represents the activation frame stack. The functions *push(_)*, *last*, and *top* are defined as usual:

$$\begin{aligned} s.push(n).top &= n \\ s.push(n).last &= s \end{aligned}$$

The procedure call Montage (Figure 19) shows that first the actual parameters are evaluated, then the *call*-action pushes the *procedureCall* instance on the stack and control is lead to the procedure-declaration with matching name.

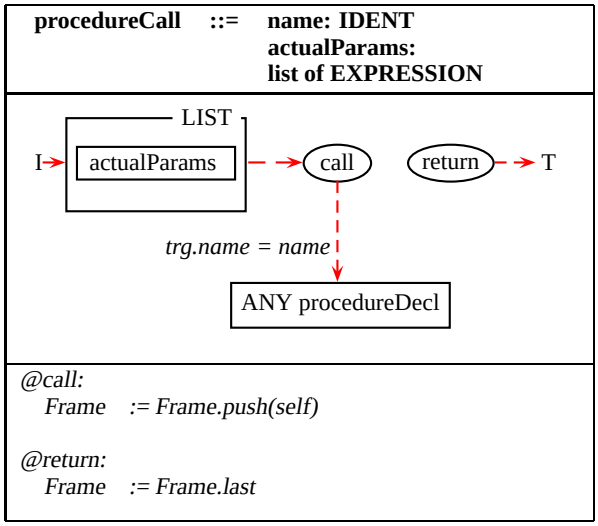

Figure 19: The procedureCall Montage

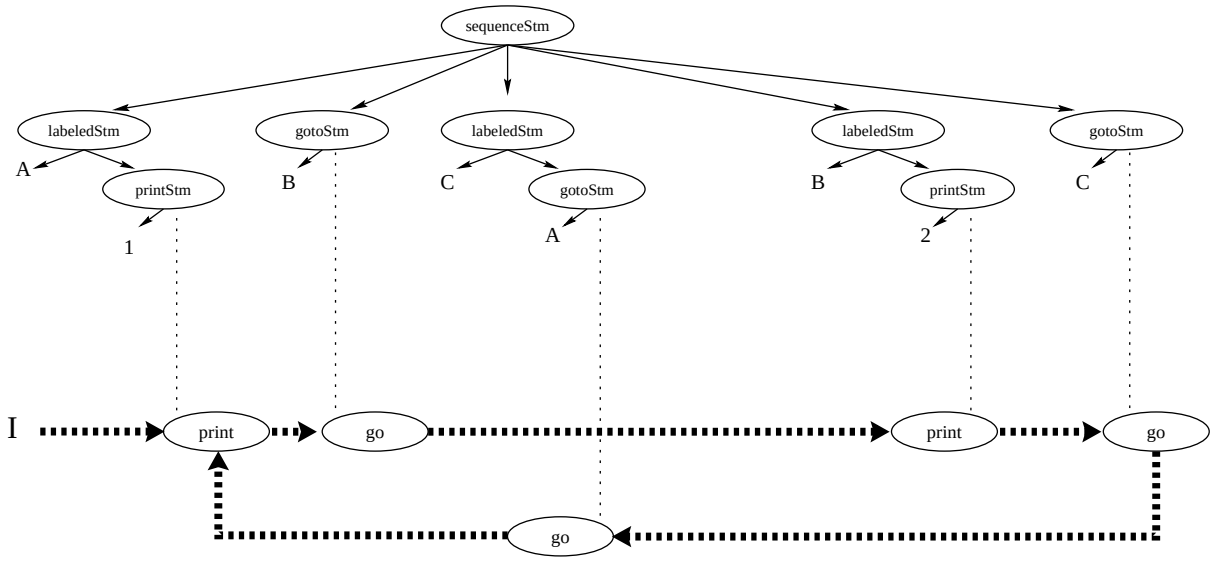

Figure 16: The flat finite state machine and its relation to the AST.

In order to make recursive calls working, both the *value* and the *VarOfVariable* functions must be reallocated at each activation frame. Formally this is done by introducing two new function, called *recValue(_)* and *RecVarOfVariable(_, _)* The functions from the model without recursion are now redefined using the new functions:

$$\begin{aligned} value &= recValue(Frame) \\ VarOfVariable(v) &= RecVarOfVariable(v, Frame) \end{aligned}$$

With this redefinitions the previously introduced Montages for expressions and statements can be reused in a context with recursion without adopting them.

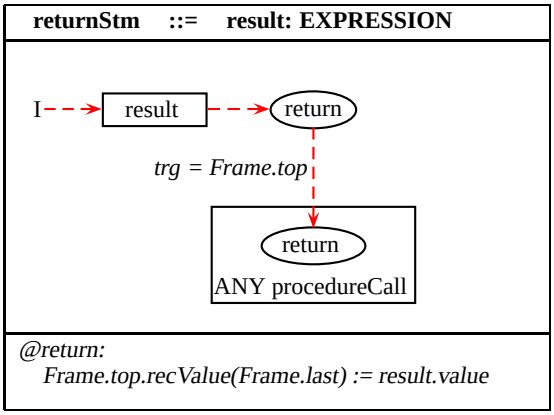

Figure 20: The returnStm Montage

In the return Montage (Figure 20) the result-expression is evaluated and then the *return*-action returns this value. *Frame.top* holds the previously pushed *procedureCall* instance. We call this node the *caller*. The value of the caller has to be set to the value of the *result* component, such that the value is visible after the current frame is removed from the stack. Thus not the *value* is updated but the *recValue(Frame.last)*. Then control is sent to the *return*-state of the caller. The action associated with this state removes the top frame from the stack by setting *Frame* to *Frame.last* as shown in Figure 19.

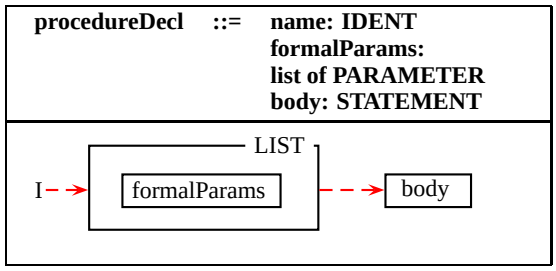

Figure 21: The procedureDecl Montage

In the procedure declaration (Figure 21) it is just shown that first all formal parameters have to be visited, then the body of the procedure call is visited. We assume that each possible thread of control in the procedure body ends with a return statement. For simplicity we do not show how such conditions are formalized with Montages. The definition of the parameters is given in Figure 22. The *actualize*-action sets the value of the formal parameter to the value of the corresponding actual parameter. Correspondence is given by the same position in the list, e.g. by the predefined attribute *listPos*. The value of the actual parameters has to be read on the caller, denoted by *Frame.top*, and one has to read the value in the last activation frame, e.g. by using *recValue(Frame.last)*.

Classes, Objects, Fields, Methods, Inheritance, Overriding, Dynamic Binding In the following we introduce programming constructs known from object oriented languages. The class declaration in Figure 23 has four components. The name is used to refer

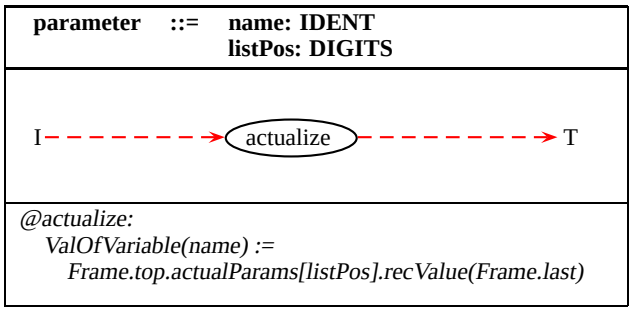

Figure 22: The parameter Montage

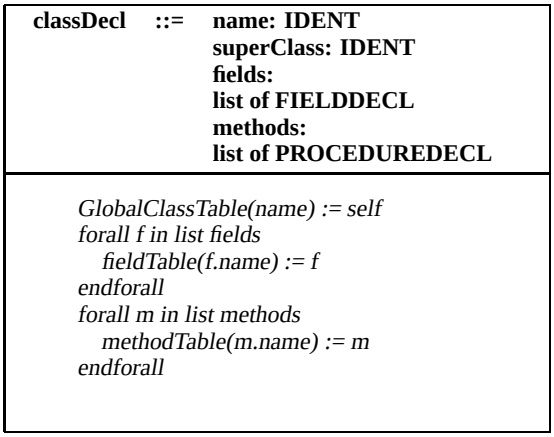

Figure 23: The classDecl Montage

to the class. A global function

$$GlobalClassTable(\_)$$

is used to map class-names to their declarations in the AST. The definition of *GlobalClassTable* is determined by the first semantic equation

$$GlobalClassTable(name) := self$$

The *superClass* component gives the name of the super-class. The

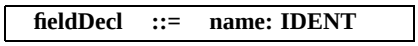

Figure 24: The fieldDecl Montage

super-class itself can be accessed as *GlobalClassTable(superClass)*. The fields component is a list of field-declarations (Figure 24). A functional attribute

$$\_.fieldTable(\_)$$

is used to map the names of the class fields to their declaration in the AST. The all-quantified semantic equation

$$fieldTable(f.name) := f$$

is defining this attribute for all fields f in the list. Similarly the list of methods is given, and a functional attribute *_.methodTable(_)* is used. The all-quantified semantic equations denote a set of semantic equations. Since the set of equations depends only on the AST, this does not destroy the declarative character of attribute grammars.

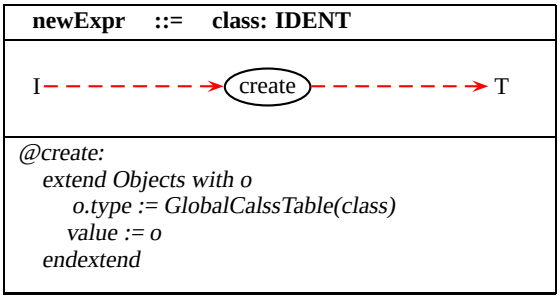

Figure 25: The newExpr Montage

The new expression (Figure 25) is used to allocate a new object of some type, and it shows the use of the ASM extend-rule. The universe *Objects* of objects in the runtime system of our language is extended with a new element o. The *type* attribute of o is set to the corresponding class declaration, and the value of the new-expression is set to o.

The value of the fields of an object are stored in a binary function

$$\_.fieldValue(\_)$$

where the first argument is the object, and the second argument is the AST node corresponding to the declaration of the field. We must use the AST-node, since due to overriding the same object can have several fields with the same name. If a field is referenced or updated, the right field is determined dynamically. This is in

contrast to most object-oriented-languages (i.e. Java) where the fields are determined statically, and only the method binding is determined dynamically). In our example language both the binding of fields and methods is dynamic.

In order to determine for an object of type c in which class its field n is declared we have the definition

```
GetFieldClass(n, c) =
  (if c.fieldTable(n) != undef then
    c.fieldTable(n)
  else
    GetFieldClass(n, GlobalClassTable(c.superClass))
```

which looks up whether the field is declared in the class, and otherwise calls the same function recursively for the super class. The field n for class c can then be computed as

```
GetField(n, c) = GetFieldClass(n, c).fieldTable(n)
```

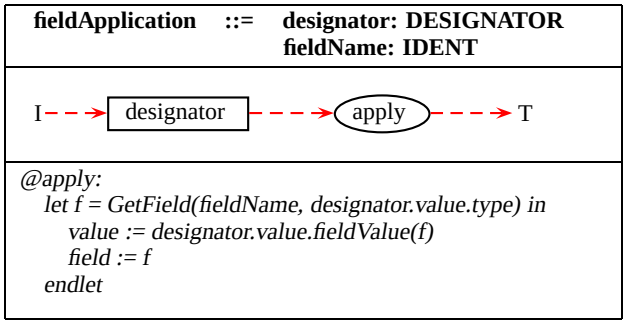

Figure 26: The fieldApplication Montage

For the definition of field applications, we introduce the type DESIGNATOR. Designators are either variable-uses (Figure 4) or field applications. In Figure 26 the definition of the field application is given. First the designator is evaluated, then the field-declaration is calculated, and the value of this field is assigned to the value of the field application. The field-declaration is assigned to *field* in order to reuse this information, if the field-application is used as left-hand-side of a field assignment. The field assignment (Figure 27)

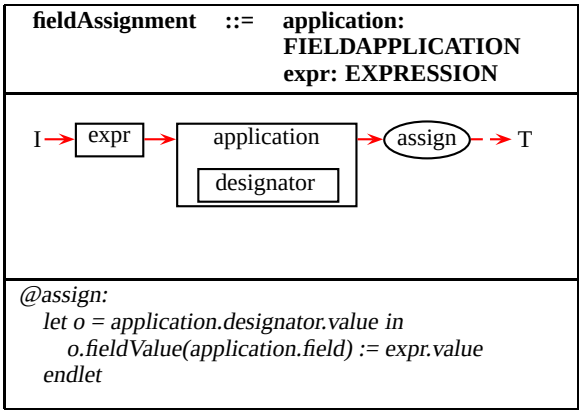

Figure 27: The fieldAssignment Montage

calculates the value of the expr-component and the application-component. The object whose field is changed is the value of the *designator*-component of the *application*-component.

Finally we give the semantics of method application. As usual in object-oriented languages the method to call is determined at run time. The type of the actual object is used to know which method is bound to an identifier. Formally, the method name n and class c are needed to determine the right method as follows:

```
DynamicBinding(n, c) =
  (if c.methodTable(n) != undef then
    c.methodTable(n)
  else
    DynamicBinding(n,GlobalClassTable(c.superClass)))
```

The remaining part of method application (Figure 28) are similar to the procedure call (Figure 19). The access to the current object is given by setting it as the value of the variable "self" in the frame to be called.

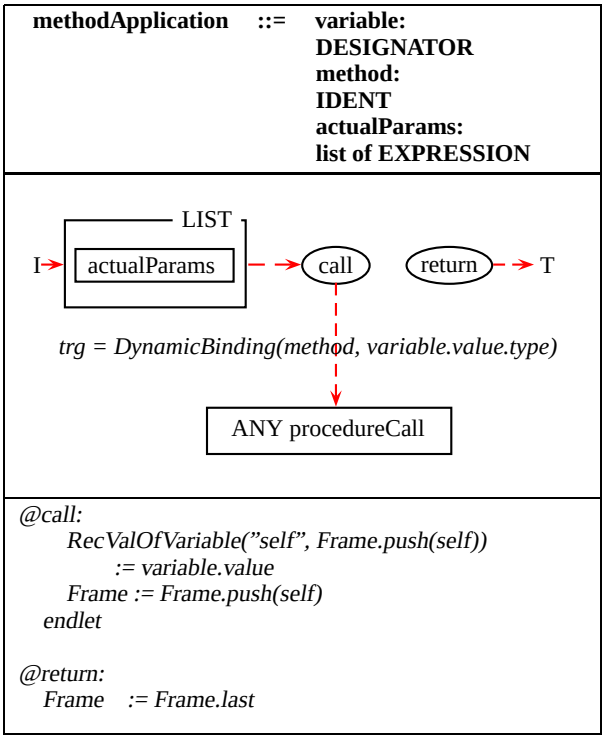

Figure 28: The methodApplication Montage