

Domain Specific Languages in Software Architecture

Matthias Anlauff
GMD FIRST
Software Engineering Group
Rudower Chaussee 5
D-12489 Berlin, Germany
+49-30-6392-1827
ma@first.gmd.de

Asuman Sünbül
Technical University of Berlin
Computer-Aided Information Systems
Einsteinufer 17
D-10587 Berlin, Germany
+49-30-314-79463
asu@cs.tu-berlin.de

ABSTRACT

Building a software product requires knowledge in both software engineering and the domain the software is written for. However, common software engineering methods in fact exclude domain experts from the architectural design of a software system, because the generality of the notations defined by these methods require expert knowledge in the field of software engineering and especially software architecture. In order to avoid this effect, domain-specific languages (DSL) have been proposed as an alternative to the general purpose languages being normally used in software architecture. One of the advantages of DSLs is that they are less cryptic and easier to learn for domain experts. Of course the design of such a language requires deep understanding of the problem domain, on the other hand the existing tools for language design are only mastered by a few experts. In order to make DSL design applicable by a broader range of people - especially domain experts - we use Montages, a semi-visual meta language based on the Abstract State Machines approach. A DSL can be specified in Montages by means of visual descriptions of the language constructs. These descriptions can be fed to a rapid prototyping tool, called GemMex, which automatically generates a visual programming environment for the specified DSL.

KEYWORDS

Domain-Specific languages, software architecture, visual specification, generation of visual programming environments.

1. INTRODUCTION

Building a piece of software settled in a certain problem domain is usually done by software engineers. Although it would often make sense to integrate people with domain-specific knowledge into the software engineering process as a whole, these people normally just contribute at the very beginning and at the very end to that process: in the requirements specification and the software testing and evaluation phase.

Standard software engineering methods like UML [UML98] and Fusion[Fusion94] define general-purpose languages for the description of components occurring in a software development. Because of their generality, the use of these languages requires technical knowledge of software engineering and are not appropriate for being used by other people. This fact prevents experts of the domain the software is written for to actively contribute to the architecture of the software, because the threshold to get acquainted with the general-purpose design languages is too high. Domain-specific languages (DSL), if designed properly for the problem domain of the software, have the advantage, that this threshold can be substantially lowered for domain experts [Bentley86, DeuKli98, CM98].

The approach presented in this paper addresses not only the use of DSL in software architecture but also the *design* of DSL. The advantages gained from the use of domain-specific languages in software architecture stand and fall with the appropriateness and acceptance of these languages. The design of a good domain-specific language requires even more knowledge than the standard software development using general purpose languages, because a DSL must form an abstraction over an entire domain while in the latter case a single concrete problem of that domain must be solved. Thus, domain experts should also participate in the design of a DSL and therefore it is necessary to open the field of DSL design to people not having experience in programming language design and implementation.

In order to make domain-specific language design applicable especially for problem domain experts, we use *Montages* [KP97a,AKP97], a semi-visual meta language for the design of programming languages. Montages are based on the *Abstract State Machines* (ASM) [Gurevich94b] approach which has proven to be appropriate to formally specify the semantics of realistic programming languages. A DSL can be specified in Montages mainly by means of visual descriptions of the language constructs. These descriptions can be fed to a rapid prototyping tool, called Gem-Mex, which automatically generates a visual programming environment for the specified DSL.

In the sequel, we further explain the use of DSLs in software architecture and described the formalism used -- Montages -- and its tool environment Gem-Mex. Section 4 shows how a DSL can be designed for a specific software architectural problem domain using Montages/Gem-Mex.

2. DOMAIN-SPECIFIC LANGUAGES IN SOFTWARE ARCHITECTURE

A DSL can be regarded as a programming or specification language dedicated to a particular domain or problem. The advantage of a domain-specific language in contrast to a general purpose language is that the DSL provides appropriate built-in abstractions and notations. In a broad range of application domains, DSLs are used, although often not recognised as such. Popular examples are the UNIX-shell startup files (e.g. *.bashrc*, *.cshrc*) or the customisation files for X-window manager applications (e.g. *.fwmrc*, *.twmrc*). However, although in each of the above cases the requirements of being a “domain-specific language” are fulfilled, this term is rarely used. More complex examples are languages providing interfaces to program libraries. A DSL makes it much easier to use the functionality of the library, because unnecessary technical details can be incorporated in the DSL. For example, Tcl/Tk provides a Tcl interface on top of the Tk graphic toolkit. More generally speaking, each DSL is specialised for a set of problems that share enough characteristics that it is worthwhile to study them as a whole. Examples for application domains where DSL have been used are graphics [Ell97,KH97], telephone switching systems [GJKW97], financial software [ADR95], robot languages [Bja96], and data-warehouse application [AKP98d].

Why using DSLs in Software Architecture?

In software architecture, several aspects make the use of DSLs interesting:

- *Involving domain experts in the software development process.* Due to the fact that software engineering methods contain general purpose languages for describing the development of a software product, people other than software engineers usually are excluded from the actual production process. On the other hand, experts of the problem domain are often needed not only during the requirements specification but also in the architectural phase. For example, if a developer is uncertain about the desired functionality of a feature that he or she has to specify, this feature must be traced back to the requirement specification, in order to make a communication with the domain expert about this problem possible. The general purpose languages are inappropriate for this purpose, because they are software architecture languages and not domain languages. If properly designed, DSLs provide a chance to also involve these domain experts in the architecture of the software product. We don't claim, that software architects are no longer needed, if DSLs are used, but we claim, that DSLs extend the range of people being able to contribute to the architecture of the software product.
- *Software Architecture using DSLs.* DSLs provide enough abstraction that they can serve as architectural specification language during the design phase of a software development. Nevertheless, because many technical details are already built in the semantics of the DSL, the specification written in a DSL can often be used to automatically generate code that forms the implementation. Thus, DSLs often bridge the gap that exists between the phases of the software engineering process, especially between the design and the implementation phase.
- *Domain-specific software architecture.* The idea of having domain-specific support for a class of similar problems can also be mapped to software architecture. Domain-specific software architecture approaches have been defined as specialisation of general purpose software architecture description languages in order to provide optimised support for a specific domain [MG92, HRPL+95, Tra94, Ves94]. The definition of a domain specific software architecture methods reflects a commonly occurring situation in practise: “off-the-shelf” software engineering and especially software architecture methods are modified so that they meet the requirements of the domain they are appointed to. In this respect, DSLs play a central role while defining a specialised software architecture approach, because they can be used at all places, where abstractions from the general purpose approach should be constituted.

3. DESIGN OF DOMAIN-SPECIFIC LANGUAGES

In general, using DSLs means that a huge amount of domain knowledge is moved from the actual program to the compiler and runtime system of the implementation language. Thus, the design of a DSL requires even more understanding of the problem domain than writing a single program for that domain in a general purpose language. DSLs only make sense, if they contain useful abstractions for the whole class of problems occurring in the corresponding domain. Thus, domain experts must also be involved in the *design* of an DSL, so that the resulting language is a useful vehicle for realistically specifying problem solutions in the corresponding domain.

The way how the semantics of a DSL is given determines for which purposes it can be used during software development. If the semantics is given informally, the DSL is good for documentation purposes in order to ease the

communication between developers and customers, for example. In this case, the DSL is not appropriate for being used as specification language, because the lack of an unambiguous semantics disqualifies the DSL for that purpose. In this case using a formally defined general purpose specification language would be the better choice. If the semantics of the DSL is given formally using some mathematical notation, then the DSL can be used as specification language, because an unambiguous description of the semantics exists. In addition -- if tool support is provided -- the DSL program can be used as part of the implementation of the resulting product, at least in a prototypical version.

In our approach, we try to meet these two seemingly contradicting requirements concerning the design of DSLs: We describe how the Montages method allows to specify the semantics of a DSL formally in an intelligible way using the mathematical notation of Abstract State Machines. Furthermore, DSL specifications are processed by a support tool (Gem-Mex) which automatically generates executable code purely based on the formal semantics descriptions.

4. VISUAL FORMAL SEMANTICS DESCRIPTIONS: MONTAGES

Montages[KP97a,AKP97] constitute a specification formalism for describing all aspects of programming languages. Syntax, static analysis and semantics, and dynamic semantics are given in an unambiguous and coherent way by means of semi-visual descriptions. The static aspects of Montages resemble control and data flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. Montages are designed to provide a mathematical basis for a number of activities from initial language design to prototyping.

The mathematical semantics of Montages is given with Abstract State Machines (formally called Evolving Algebras[Gurevich94b,ASMHomepage]). In short, ASMs are a state-based formalism in which a state is updated in discrete time steps. Unlike most state based systems, the state is given by an algebra, that is, a collection of functions and universes. The state transitions are given by rules that update functions point-wise and extend universes with new elements.

ASMs have already been used to model the dynamic semantics of a number of programming languages, such as C [GurHug93], Occam [BoDuRo94], C++ [Wallace94], Oberon [Kutter97], and Java [BoeSch98a] to mention a few. At the risk of oversimplifying somewhat, one defines the *initial state* of the functions and specifies how they evolve by means of *transition rules*. The *initial state* is assumed to include the results of a static analysis. After this analysis the program's control and data flow is represented in the form of functions between parts of the program text. As usual the control flow functions specify the order in which statements are executed, and the data flow functions specify how values flow via variables through operations. The corresponding *transition rules* update the system state and let the control evolve through the control flow.

The existing case studies showed that it is possible to model with ASMs the dynamic semantics of realistic programming languages, but they have the disadvantage that they do not formalize the static aspects. Montages engineered the ASM's approach to programming language semantics showing how to model consistently not only the dynamic semantics, but the static analysis and semantics as well. In particular, Montages describe how to define intensionally the abstract syntax¹, i.e. the control and data flow, starting from the concrete syntax. This mapping is provided by means of graphs which confer to the specification a great intelligibility.

A language specification is given by a collection of Montages, which is hierarchically structured according to the corresponding context-free grammar. Each Montage is a "BNF-extension-to-semantics", that is a self contained capsule in which all the properties of a given construct are formally defined.

In Fig. 1 the Montage specification of a "While" construct is presented. The topmost part in the working area is the production rule defining the context-free syntax. Below is the local flow, i.e. the graphical definition of the mapping between the concrete and the abstract syntax of the while statement. It describes the portion of the parse tree associated with the while statement and enriches it with some additional information, that is the control and data flow edges. In Montages, the inner nodes of the parse tree are represented with boxes and leaves with ovals, whereas the node corresponding to the while statement is denoted by the *self* oval node. Moreover, nested boxes can be used to represent nodes on lower levels of the parse trees. The solid and dotted arrows denote the data and control flow, respectively. Control flow arrows may be labeled by means of firing conditions, i.e. predicates which determine through which edges the control must flow. For instance, the predicate *guard.value* indicates that the control is passed from the while node (denoted by *self*) to the sequence of statements *Statement.Sequence* whenever the value of the expression *Expression*, which is retrieved by means of the data flow arrow *guard*, is evaluated to *true*. The control flow arrows I (initial) and T (terminal) are special arrows which serve to plug together the local flow-information to the global one. In other words, the abstract syntax (graph) or global flow of a program is obtained from its parse tree by

¹ Since the Montages framework is state-based and not functional, by *abstract syntax* it denotes an abstract syntax graph rather than a tree.

inductively decorating it with the information defined in the local flows of the statements occurring in the program itself.

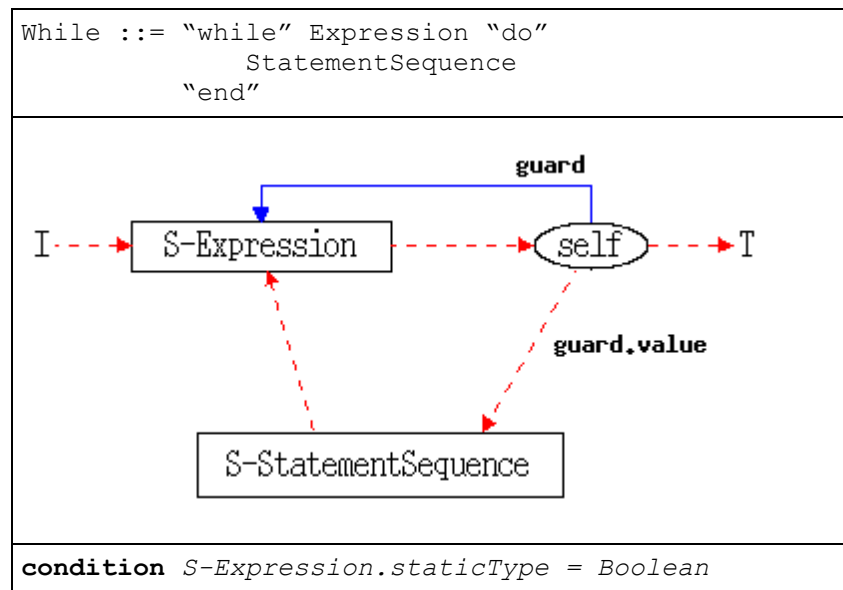


Figure 1: The While Montage

The third part of the while Montage contains the static semantics, i.e. the type-checking condition which requires that the expressions in the while statement must be booleans. In general, the designer may make use of full first-order logic to express context sensitive constraints. In this example, the part containing explicit dynamic semantics rules is missing. This is usual for most of the control statements, but there are also cases in which an additional transition rule is needed to define the complete dynamic semantics as illustrate, for instance, in Fig 7 and Fig 8.

The semantics of a Montages specification of a programming language is therefore the following, given a program

- the context-free grammar obtained by collecting all the EBNF-rules in each Montage defines the concrete syntax;
- the local flow, i.e. the visual part of each Montage, defines how to provide the parse tree with all the information which are needed to define in turn the dynamic semantics, in particular the control and data flow, respectively, of that given construct;
- the condition part is a first-order logic predicate which is evaluated while traversing the parse tree, i.e. for each internal node the corresponding predicate is checked. In general, different traversal strategies can be specified, predicates can be checked before or after the analysis of subtrees, or maybe the designer may prefer to define several passes, e.g. one additional pass to check declarations before the other fragments. A more leisureed and detailed discussion can be found in [KP97a].
- the dynamic semantics part is a transition rules which is fired whenever the control reach that given construct, i.e. the *self* node. For instance, in Fig. 8 whenever the control reaches the node denoted by "self" (which denotes to the node of the construct *EventConj*) the the following rule is enabled

$$fire_condition := check.Sensor(sensor.Name,value.Name)$$

which updates locally the value of the function *fire_condition*.

The logical and formal aspects of Montages are not supposed to be illustrated deeply here, a more detailed discussion can be found in [KP97a]. Nevertheless, it should be stressed how Montages represents a rigorous and formal instrument providing a mathematical framework which can be used by the designer to record unambiguously decisions about a particular language and obtaining new insight into the nature of the language developing description of it. However, our interests focussed more on the pragmatic qualities of the formalism and its effectiveness, rather than its mathematical properties.

5. RUNNING EXAMPLE: A DSL FOR THE SOFTWARE ARCHITECTURE OF VENDING MACHINES

In this section, we will illustrate the design of a DSL using Montages and Gem-Mex. As an example, we want to design a language for specifying the software architecture of simple, state-based vending machines, we will call it „VML“ for „vending machine language“ in the sequel. With this specific DSL one should be able to describe the constituents and the behavior of the control software of these machines. Examples are beverage and food dispensers, automated teller machines etc. . In order to describe these kind of machines, VML must contain constructs to describe

- input and output devices, like sensors or displays together with their possible values;
- actions that can be triggered by the control software;
- states together with permissible events for each state;
- state transitions.

In order to get an idea of the example domain-specific language VML, Figure 2 contains an example description of a vending machine dispensing beverages.

```
BEGIN SimpleBeverageDispenser
DISPLAY BevLed: {on,off};
SENSOR MoneySlot:{active},
      BevButton:{pressed},
      CancelButton:{pressed},
      BevEmptySensor:{empty};
ACTION DispenseBeverage();
ACTION ReturnMoney(),GetMoney();
FORWARD WaitForMoney; FORWARD Initial;
STATE Reset IS
    BevEmptySensor SIGNALS empty ->
        BevLed <- off;
        GOTOSTATE Initial;
    BevEmptySensor SIGNALS nothing ->
        BevLed <- on;
        GOTOSTATE Initial;
ENDSTATE Reset;
STATE Initial IS
    BevButton SIGNALS pressed AND
    BevEmptySensor SIGNALS nothing ->
        GOTOSTATE WaitForMoney;
ENDSTATE Initial;
STATE WaitForMoney IS
    MoneySlot SIGNALS active ->
        GetMoney();
        DispenseBeverage();
        GOTOSTATE Reset;
    CancelButton SIGNALS pressed ->
        GOTOSTATE Initial;
ENDSTATE WaitForMoney;
END
```

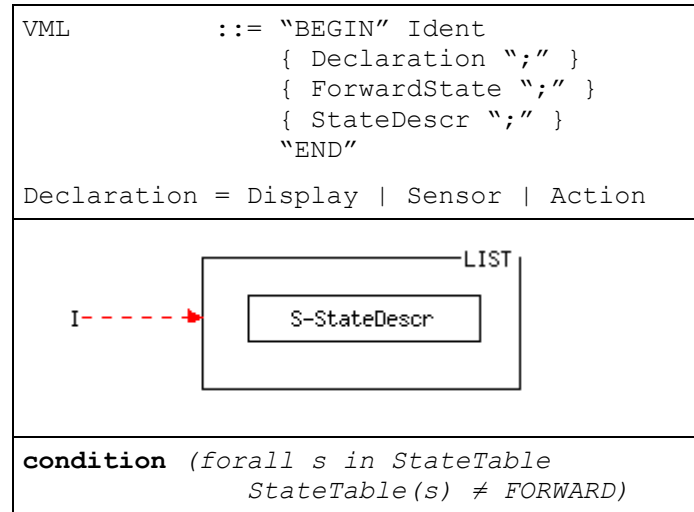
Figure 2: A “VML”-specification of the architecture of a beverage vending machine

The vending machine specified is able to dispense one kind of beverage. The customer presses the “BevButton” when the system is in its initial state. After that, the machine wait for an “active” signal from the “MoneySlot” sensor or the user may press the “CancelButton”. In the “Reset” state the system checks whether the “BevEmptySensor” signals that there are no more beverages in the vending machine and sets the value of the display components accordingly. For a more realistic modeling other components may be introduced.

5.1 Specifying VML using Montages

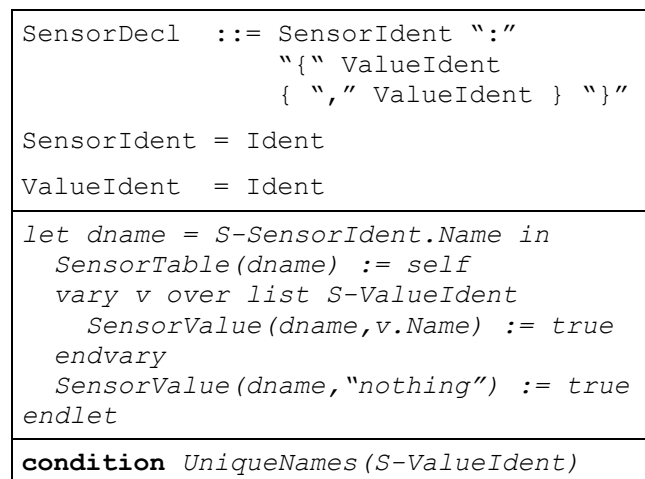
The Montages specification of VML will be sketched in the following. Note, that the specification parts shown here are an excerpt from a running version generated within the Gem-Mex support environment.

We will first have a look at the definition of the general structure of a VML program. This is given by the Montage for the nonterminal symbol “VML” as shown below.



The grammar rule define the syntax of an VML program: after the declaration of display, sensor, and action items, the states can be specified. A “FORWARD“ construct is introduced, in order to simplify the static analysis part. The static condition requires, that each entry in the „StateTable“ points to a actual node in the abstract syntax tree and doesn’t contain the special constant “FORWARD“. In the static analysis phase, all descendants of the VML node are visited, e.g. the lists of Declaration, ForwardState, and StateDescr, then the condition is checked. After the static analysis phase, the dynamic semantics can be executed by passing control to the initial node of the start symbol, e.g. the first StatDescr, as graphically specified by the I-arrow in the VML Montage.

The following figure contains the Montage for the nonterminal symbol “SensorDecl“. In the static part, the name of the sensor item together with the list of possible values for the sensor are stored into global tables named “SensorTable“ and “SensorValue“. These tables are declared at another place of the specification as one- resp. two-ary dynamic ASM functions.



The condition states that the values specified must be pairwise disjoint, where “UniqueNames“ is a macro defined as

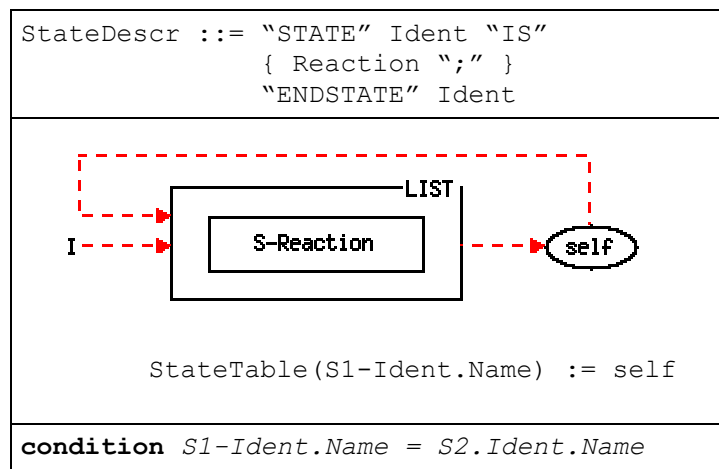
```

macro UniqueNames(IDENTLIST) is
  (forall v1 in list IDENTLIST:
    (forall v2 in list IDENTLIST:
      (v1_ind=v2_ind) or (v1.Name!=v2.Name)
    ))
endmacro

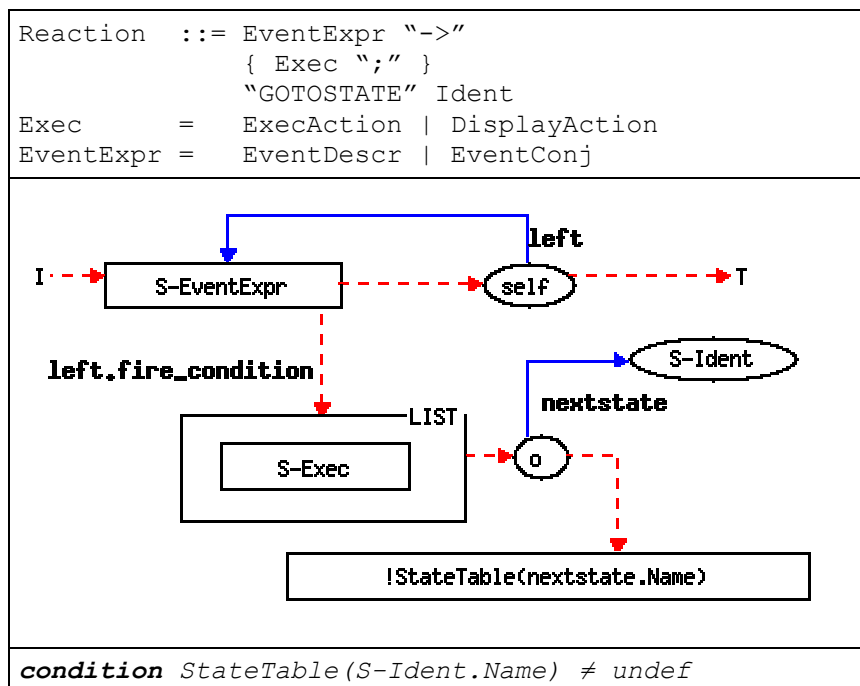
```

where “x_ind“ refers to the corresponding list index of the iteration variable x. The declaration of sensor values is specified accordingly.

Montage “StateDescr” shows the specification of the semantics of the “STATE“ construct. In the static analysis, the “StateTable“ is updated to contain the root node of the state in the syntax tree (“StateTable(S1-Ident.Name) := self“). This node serves as entry point when a “GOTOSTATE“ construct is executed. Reaching the end of the inner constructs of a state means, that none of the specified events have occurred. In this case, the state is not changed, the control flow returns to the initial task of the same state, which is specified by the dotted arrow in the figure below.

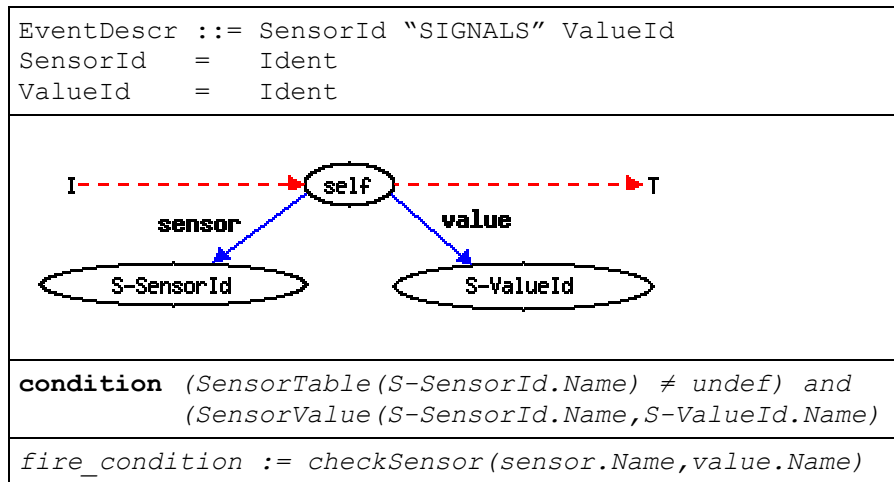


The following figure shows the “Reaction“-Montage. This construct is used in VML to specify the reaction of the system to an event described by the “EventDescr“. If this is the case, the actions specified in the “Exec“ list are executed and thereafter the system changes its state to the one specified after the “GOTOSTATE“ keyword.

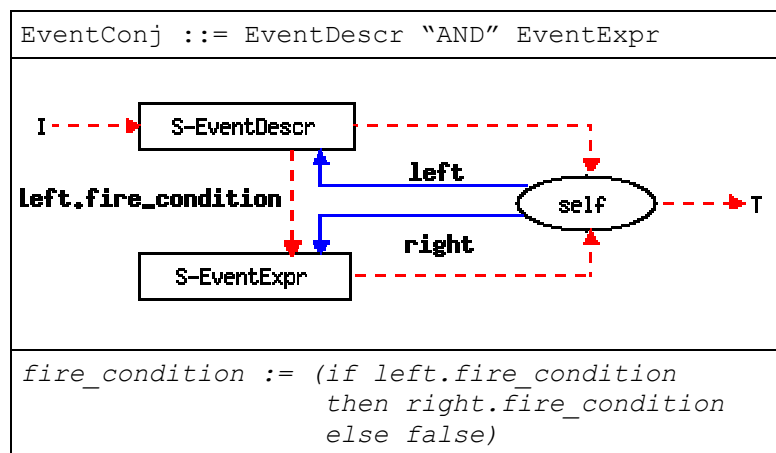


The control flow of this construct is completely specified graphically: if the fire condition of the event is fulfilled, then the “Exec” list is executed, otherwise this branch is skipped and the construct is left. After the actions of the “Exec” list have been processed, the control flow is directed to the next state the entry point of which is stored in the global “StateTable”. Note, that this can be done directly without checking whether a corresponding entry exists, because this has already been done by the static condition.

The following Montage contains the semantics of the “EventDescr” construct. The static condition checks whether the sensor value is defined for the given sensor item. In this Montage an additional dynamic semantics rule is specified, which updates the attribute “fire_condition” according to the actual values of the sensors.²



Finally, in the Montage “EventConj” an example is given how lazy evaluation of a conjunction can be expressed. The second part of the conjunction is only reached, if the fire condition of the first part is fulfilled, otherwise the construct is left via the self node. The actual value of the fire condition of the combined construct is given textually.



The Montages presented in this section together with a handful more are sufficient to formally specify the semantics of the DSL. The specification can be processed by the Gem-Mex system, which also automatically generates a visual debugging and animation interface for VML.

6. GEMMEX: TOOL SUPPORT FOR MONTAGES

² In the implementation “checkSensor” is an external function returning a random boolean value.

The development environment for Montages is given by the Gem-Mex tool. It is a complex 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:

- the Graphical Editor for Montages (Gem) is a sophisticated graphical editor in which Montages can be entered; furthermore high-quality documentation can be generated, see the Montages of Section 5 which have been generated this way;
- the Montages executable generator (Mex) which automatically generates correct and efficient implementations of the language;
- the generic animation and debugger tool visualizes the static and dynamical behavior of the specified language at a symbolic level; source programs written in the specified language can be animated and inspected in a visual environment.

A broad range of professionals may find interesting and convenient to use Gem-Mex. The whole development of a DSL can be supported with an effective impact on the productivity and robustness of the design. The designer can enter the specification, browse it and especially maintain it. Specifications may evolve in time even in a non-monotonic way since modifications can be localized within very neat boundaries. By doing so, different experimentation can take place with different versions of the syntax and semantics of the specified language in a very short time.

Besides the pure editing functionality, Gem can be used to generate documents suitable for specification presentation. Experience suggests how lack in documentation is a dangerous bottleneck for the consistency and coherence of a project. Both, paper and online presentation of the language specification are automatically generated by Gem:

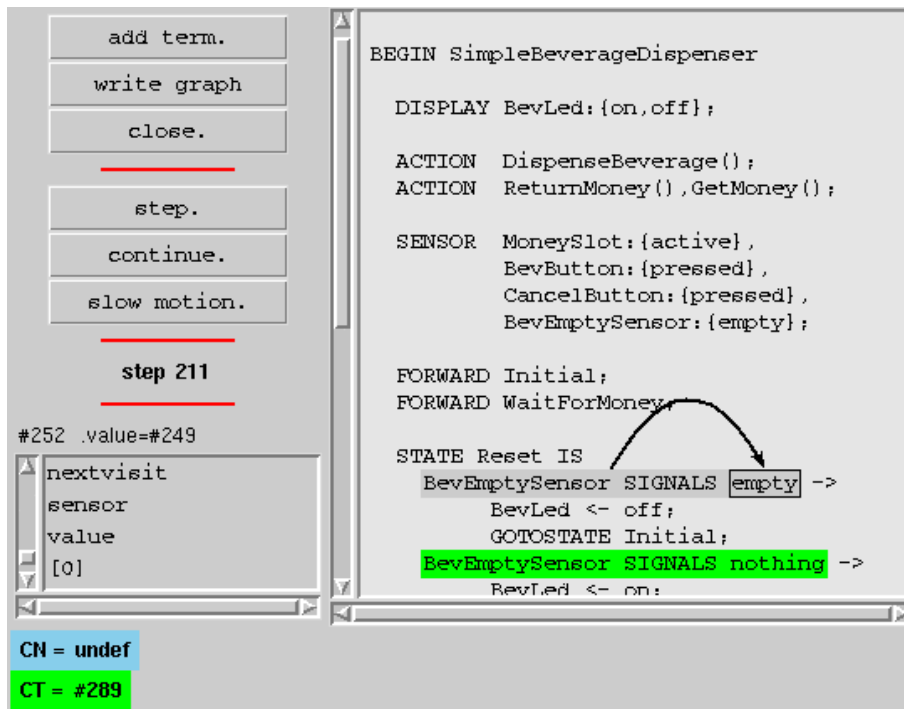
- LaTeX documents illustrate the Montages and the grammar; such documents are easily customizable for the non-specialist user;
- HTML versions of the language specification allows to browse the specification and retrieve pieces of specification.

Moreover, intelligibility is enhanced by means of “literate specification” techniques directly supported by Gem. Formal parts of the specification can be substituted with textual elements by means of a “literate programming” tool integrated in the system. “Literate specification” means that the Montages text fields may contain references to other parts of the formalization specified outside of the Montages modules. Thus, the readability and comprehension of a Montages specification results very much similar to those of language manuals.

6.1 Generation of Visual Animation Tools

Gem-Mex generates an executable version of the Montages specification by simply feeding the formal ASM-semantics of Montages to Aslan [An98]. Aslan is an efficient and flexible implementation of the ASM approach. Besides a plain textual version of the DSL interpreter, Gem-Mex also generates a visual debugging and animation tool for the specified DSL. The generated graphical user interface allows the language designer to browse the data structures of the formalisation and to trace the program run graphically.

The following figure shows the “view source” subwindow of the generated graphical user interface for VML. The movement from one task to the next one is animated by an arrow that starts from the previously active node and ends in the currently active node. In addition, there exist “textual” subwindows, where the values of globally defined functions can be inspected.



7. CONCLUSION

We have introduced a tool support environment that can be used for designing DSL to be used in software architecture. With this framework, one is able to intelligibly specify the syntax and semantics of the DSL based on a formal mathematical approach, Abstract State Machines. We pointed out, that domain experts should also be involved into the software architecture phase which is much easier, if DSLs are used in these phases rather than general purpose software architecture notations. On the other hand, these DSL must also be well-designed, so that domain knowledge is also needed during the design of an DSL. Our approach offers a possibility for a broader range of people to also participate in the design of an DSL, especially domain experts. Using a simple DSL named VML (vending machine language) we illustrated the design of DSLs. With Montages/Gem-Mex it is possible to specify the semantics of a DSL formally and in an intelligible way; a task that is very important, if DSLs should be used during software architecture.

References

- [Anl98a] M. Anlauff. Aslan: Efficient and flexible implementation of abstract state machines. August 1998
- [Anl98b] M. Anlauff. Gem-Mex: The Tool-Support Environment for Montages
- [AKP97] M. Anlauff, P. Kutter, A. Pierantonio. Formal aspects and development environments of montages. *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Amsterdam 1997
- [ADR95] B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language describing financial products. *IEEE Workshop on Formal Methods Application in Software Engineering*, 1995
- [Ben86] J. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711 721, 1986
- [Bja96] E. Bjarnason: Applab: A laboratory for application languages. In L. Bendix, K. Normark, and K. Osterby, editors, *Nordic Workshop on Programming Environment Research, Aalborg*. Aalborg University, 1996