From Ease in Programming to Easy Maintenance : Extending DSL Usability with Montages

Matthias Anlauff* **Alberto Bemporad†** **Samarjit Chakraborty‡** **Philipp Kutter‡**
Domenico Mignone† **Manfred Morari†** **Alfonso Pierantonio§** **Lothar Thiele‡¶**

Abstract

Montages is a formalism for giving executable specifications of programming languages. In contrast to related formalisms, it is state-based, and specifications can be given in an imperative or object-oriented style with which most programmers are comfortable. Hence, in the context of Domain Specific Languages (DSLs), it provides a framework in which domain engineers can themselves extend and maintain their language. It also uses visual notations, resulting in concise, intelligible specifications. Using the associated tool support Gem-Mex, such specifications can be used to automatically generate a programming environment with animation facilities. In this paper, we give an overview of the formalism and the tool support, focusing on its advantages in DSL maintenance and implementation. Towards this we also present a case study using a language called HYSDEL, developed by control engineers for specifying hybrid systems. A full implementation of this language is available and is currently being maintained by the Automatic Control Laboratory of ETH Zurich.

1 Introduction

With the increase in the diversity of applications in which computers are being used today it is becoming eminent for people working in different fields to use sophisticated computational tools. The background required to use such tools, however, often fall within the realm of core computer science and is seldom available with people specializing in other areas. Increasingly, within such contexts, the question arises whether the usability of such tools can be improved by using concepts which are closer to the domain in which the tool is used. Of late, Domain Specific Languages (DSLs) are being proposed as a possible solution to this.

A well designed DSL pertaining to a specific problem greatly helps the domain engineer to write concise programs, and has all the pragmatic qualities such as faster development and safer applications. But to incorporate large changes in the problem domain which may necessitate a change in the DSL, it requires skills (in compiler technology, for example) which are usually not available with the domain experts even though most of them possess a fair degree of programming expertise [33]. Associated with this is also the usually high startup cost, involved in implementing the language and the supporting tools [18].

At the first thought, a feasible solution to the maintenance problem is to predict the typical modification requests that are expected to arise, and then design the language so that such requests are expressible in the language itself and do not lead to modifications in the compiler [10, 33]. However, this is easier said than done. And trying to design the language so that it can accommodate possible future modifications might unnecessarily complicate the present form of the language, rather than keeping it small and compact, and thereby defeat the goals of using a DSL. To accommodate these two possibly conflicting goals, in this paper we advocate an implementation/prototyping scheme that is amenable to large changes as the language develops over time, while keeping both the language and the implementation compact at all points of time. Towards this we propose a framework

*GMD FIRST, Berlin. `ma@first.gmd.de`
†Automatic Control Laboratory, ETH Zürich. {`bemporad, mignone, morari`}`@aut.ee.ethz.ch`
‡Computer Engineering and Networks Laboratory, ETH Zürich. {`samarjit, kutter, thiele`}`@tik.ee.ethz.ch`
§Universitá di L'Aquila, Italy. `alfonso@univaq.it`
¶Author to whom all correspondence should be directed.

in which the domain engineers are able to maintain and extend their (domain specific) language as it evolves and changes with the requirements of their specific problem; and possibly also implement their language from scratch. This greatly extends the advantages of the DSL, beyond what is typically available currently.

Our framework called *Montages* [23] is a semi-visual meta language intended for giving executable specifications of programming languages. Using the associated tool support *Gem-Mex* [1], such specifications can be used to automatically generate a programming environment consisting of a collection of interactive tools, such as parsers, type-checkers, interpreters and symbolic debuggers, accompanied by animation facilities. The components of Montages used for describing the static aspects of semantics can be used for source-to-source translations as well, which is typical for many DSL implementations. Existing approaches for defining programming language semantics such as Centaur [19] and ASF+SDF [21] are advocated for this job as well. Since they are based on a functional framework and make use of the structure of the abstract syntax tree, it is natural to apply them for translations also [7, 30, 31]. However, the specification formalism used in each of these - algebraic specifications and conditional rewriting systems in ASF+SDF, and natural semantics [20] in Centaur, require a mathematical background which is not commonly available. Moreover, although such frameworks have typically a rich deductive proof system, their application to imperative languages (such as most DSLs) lead to long and complex specifications.

Attribute grammar systems on the other hand, are aimed directly on translations, typically from a high level to some machine language. Nevertheless, they are still outside the reach of general programmers without sufficient background in compiler construction [13]. Montages uses a version of attribute grammars which is embedded into a state based formalism. While specifying the semantics of general purpose programming languages the full features of attribute grammars are used, a reduced and simplified version is sufficient for source-to-source translations. It is easy for most practicing engineers to understand and allows short and concise specifications. Additionally, the specifications can be stated in an imperative or object-oriented style, with which most programmers are comfortable. We sacrifice the mathematical richness required for automated reasoning, in favor of more engineering qualities like easier specifications and maintainability. Nevertheless, the underlying framework for Montages is completely formal and hence the qualities of a formal specification like consistency and unambiguity do carry over.

A complete language specification is structured into specification modules called Montages. Each Montage specifying a language construct contains the context-free grammar rule using EBNF, a control and data flow graph using the *Montage Visual Language* (MVL), and the static and dynamic semantic rules of the construct using Abstract State Machines (ASM) [2, 15]. Such specifications can be entered, edited, debugged, and executed, using the Gem-Mex tool. A Montage structure depends directly on the syntax of the construct and has a strong notion of locality, i.e. a given behavior is limited within a small number of Montages, if not just one. Hence revising a construct or adding new constructs is extremely simple. This is in contrast to algebraic specification formalisms where a specification is given my means of a signature and a set of axioms which describe some behavior; and usually the behavior is distributed over the entire specification. Montages is partly based on the computational model of ASMs, whose goal is to simulate algorithms on their natural abstraction level. In an essence this also means that languages can be specified using Montages on the same abstraction level of the problem domain. This has important implications in the context of DSLs, if such specifications are to be maintained and modified by the domain engineers themselves. It carries over the same advantages of programming in a DSL to the work involved in implementing the DSL as well.

In this paper we make an attempt to illustrate these advantages using our experiences in implementing a DSL called *HYSDEL*. HYSDEL (HYbrid System DEscription Language) has been developed for control engineers to specify hybrid systems in a modelling framework called the *Mixed Logical Dynamical form* [8]. This language was designed at the Automatic Control Laboratory at ETH Zurich. As the language was based on ongoing research in hybrid control systems, its specifications were continuously updated by its designers. Hence it was important to choose a suitable implementation scheme which does not inhibit further developments of the language. Currently it is implemented using Montages and is being maintained by researchers in control engineering. For problems like controller synthesis, state estimation, and reconfiguration, systems are specified in the MLD form using HYSDEL. This specification has to be translated into MATLAB [25] code using techniques described in [8], which are then used for simulation and optimization purposes. Our implementation is thus a translator from HYSDEL to MATLAB code. However, we shall also point out in this paper that in many cases the full features of Montages (like specifying dynamic semantics) might be used to prototype a DSL directly rather than transforming it to another high level language. Such a scheme does not entail much additional complexity over that required for specifying the static aspects. Further, we believe that where applicable, it is much less error prone than source-to-source translations. This was not possible in the case of HYSDEL, for obvious reasons.

The work in this paper can hence be summarized as follows :

- We present a scheme for prototyping/implementing DSLs, which can be updated and maintained by domain engineers with only a moderate level of programming experience. No deep knowledge about compiler construction techniques or programming language semantics is required.
- A realistic case study involving a source-to-source translation from a DSL to a high-level language (MATLAB) is presented. The implementation is currently being maintained by control engineers.
- The possible advantages of using this framework for directly implementing DSLs (without translating into another high-level language) is discussed, along with the work done in this direction.

In the next section we give an overview of the Montages formalism and the associated tool Gem-Mex. Then we describe the language HYSDEL and its implementation using Montages. Following this, we briefly discuss the possibilities of directly implementing DSLs by specifying their dynamic semantics using Montages, and finally we conclude with the implications of this work and future directions.

2 An Overview of Montages

Montages is a formalism for the full specification of imperative and object-oriented programming languages. It serves as a vehicle for formally recording the design decisions taken during the development of a language, and automatically generate a prototype out of it. In this section we informally describe it and its associated tool support Gem-Mex. A detailed formal description can be found in [23].

A language is specified as a collection of abstract data types called *Montage*s. They are hierarchically arranged according to the context-free grammar rules of the language. Each Montage specifying a language construct consists of four parts, depending on the semantics of the construct.

1. The EBNF production rule specifying the syntax.
2. A control and data flow graph using a simple visual language devised for this purpose (the Montages Visual Language - MVL).
3. Static semantic rules.
4. Dynamic semantic rules.

As an example, Figure 1 shows a Montage specifying the construct *Sum*, having the usual semantics. It does not contain any static semantic rules, which would have otherwise appeared below the control-data-flow graph.

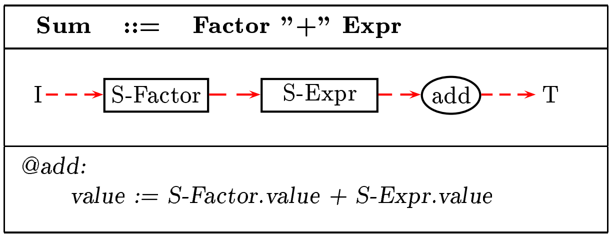


Figure 1: The Montage *Sum*

The semantics of a collection of Montages is an Abstract State Machine (ASM) [2, 15]. Such an ASM can be considered to be composed of two parts - the first modelling the static analysis and semantics, and the second defining the dynamic semantics. A program of a specified language, or rather the resulting parse tree, defines the initial state of the ASM. The static analysis phase decorates the leaves of the tree with the control and data-flow information specified in the Montages, and as a result defines a sequence in which the nodes of the parse tree are to be visited during the execution of the dynamic semantics. Additionally, the static semantic rules are also executed during static analysis. Now we state further details.

2.1 Mapping Concrete to Abstract Syntax

The EBNF production rule stated in the first part of each Montage specify the context-free syntax of a construct, and allow the generation of a parser from the language being specified. Additionally, all such rules define a mapping from the parsed program to the Abstract Syntax Tree (AST). The composition of different Montages, each based on a partial specification, is based on the structure of the AST.

EBNF rules. Without loss of generality, any EBNF rule can be assumed to be in one of the following two forms :

$$\begin{aligned} A &::= B\ C\ D \\ E &= F \mid G \mid H \end{aligned}$$

Rules of the first form are called *characteristic productions* and those of the second form are *synonym productions*. Obviously, for syntax rules stated in this form, each non-terminal appears exactly in one rule as the left-hand side. Non-terminals appearing on the left-hand side of characteristic productions are called *characteristic symbols* and those appearing on the left-hand side of synonym productions are called *synonym symbols*.

Composition of Montages Each characteristic symbol defines a Montage, which on an abstract level can also be considered as a *class* whose instances are the corresponding nodes in the AST. Such a class has attributes (e.g. *value* in the Montage *Sum*) and methods, which are the semantic rules. Symbols on the right-hand side of a characteristic production are called the (direct) *components* of the Montage containing the production, and symbols which are reachable as components of components are called *indirect components*. In order to access descendants of a given node in the abstract syntax tree, statically defined attributes called *selectors* can be used. In the example given above, the B, C, and D components of any instance of A can be accessed by the selectors *S-B*, *S-C*, and *S-D* respectively. Figure 2 illustrates these concepts.

The synonym productions introduce *synonym classes* and define subtype relations. The symbols on the right-hand side of such productions can be further synonym classes or Montage classes and each such class is a subtype of the class on the left-hand side. Thus, each instance of a class on the right-hand side is also an instance of the synonym class on the left-hand side.

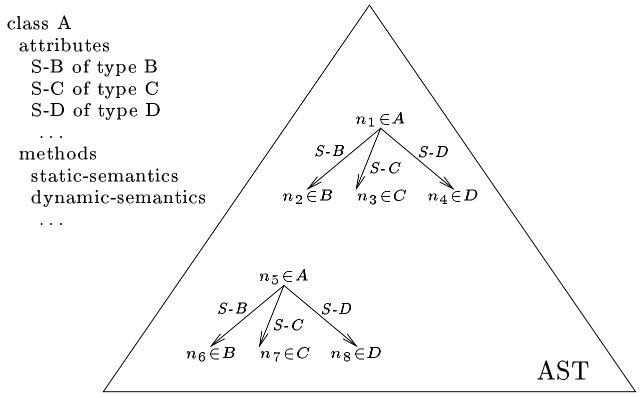


Figure 2: Montage class A, instances in the AST, and selectors S-B, S-C, S-D

Terminals, e.g. identifiers or numbers, do not correspond to Montages. They are accessed using a generic attribute termed as *Name*. This described treatment allows automatic generation of the AST from the concrete syntax [28]. Inside a Montage class, a generic term *self* denotes the current instance of the class and using the selectors and the generic attribute *Parent*, arbitrary nodes of the AST can be accessed. For example, *self.S-B.S-H.S-J* denotes a node of the class *J*, which can be reached following the selectors *S-B, S-H* and *S-J*. This obviously requires such a path to exist in all possible ASTs.

2.2 Control and Data-Flow Graphs using MVL

The elements of the MVL are rectangular boxes, ovals, and dotted and solid arrows. The boxes represent the direct components of a Montage and correspond to the symbols on the right hand side of the EBNF rule in the Montage. As mentioned in the last subsection, from any given instance of a Montage in the AST, the nodes corresponding to the direct components can be accessed by using the selector functions. The ovals represent dynamic semantic rules, and are labelled with the name of the rule (e.g. *@add* in the Montage *Sum*). Next, the dotted arrows connecting the boxes and the ovals represent the control-flow between the nodes of the AST corresponding to respective Montages, during the execution of the dynamic semantics. The entry and the exit points in the control-flow graph are indicated by the *I* (initial) and the *T* (terminal) arrows, which are used to plug this (local) graph into the global control-flow graph. Finally, considering the underlying state based framework, if the source of a control flow arrow is *active* and the predicate on the arrow evaluates to true, then control is passed to the target of the arrow. More generally, if the predicates on all the arrows originating from a node evaluate to false, then the control by default follows the unlabelled arrow. This is illustrated using the Montage *While* in Figure 3

Lastly, in the Montage *While*, the solid arrow marked *guard* indicates a data flow from the box marked *S-Expression* to the node *branch*. The semantics of this data flow arrow is that the attribute *guard* of *While* (all nodes in the AST corresponding to it) is assigned the value *S-Expression.Terminal*. Here *Initial* and *Terminal* represent the entry and the exit points in the control-flow graph of a Montage defined inductively. For example in an AST, the *Initial* of the node corresponding to the Montage *Sum* is the *Initial* of the node corresponding to *S-Factor*, so on and so forth. Hence, if an instance of *Expression* is a *Sum*, then *guard.value* can be used to access the *value* attribute of the node corresponding to *Sum*, which fully explains the behavior of *While*.

In addition to this basic MVL terminology, there are a few other possibilities. For example, a number of boxes can be nested to represent indirect components, and lists are depicted by a special *LIST*-box. These simple visual constructs can be combined in different ways to effectively represent constructs like lists and cases, details of which can be found in [23].

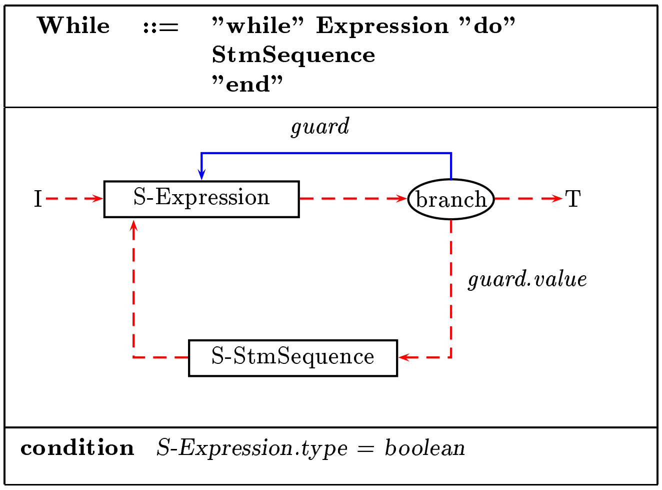


Figure 3: The Montage *While*

2.3 Static and Dynamic Semantics

The static analysis phase is responsible for decorating the leaves of the AST using the information provided by the control-data-flow graph using MVL. This defines the sequence in which the leaves of the AST are visited during the execution of the dynamic semantics. Additional static semantic rules can be stated below the control-data-flow graph which are executed at the corresponding nodes of the AST during static analysis. Such rules typically update some attribute or a global function. Lastly, the *condition* (e.g. in the Montage *While* in Figure 3) at each node is also checked during static analysis, which can be a full first-order logic predicate.

After the static analysis phase, the dynamic semantic rules at each node of the AST are executed, with the control flow defined by the graph constructed during static analysis. Both the static as well as the dynamic semantic rules are stated as ASM rules, the details of which can be found in [2, 15, 23]. An example of a dynamic semantic rule is in the Montage *Sum* in Figure 1, where the attribute *value* is updated.

2.4 Gem-Mex : The Development Environment for Montages

Gem-Mex [1] is the associated tool support for Montages. It is built up of a number of interconnected components. Gem is a graphical editor for Montages using which the specifications are entered, and Mex - the Montages executable generator, automatically generates an implementation of the specified language. In addition to these, Gem-Mex also consists of a generic debugger tool and animation facilities.

Generating Interpreters. Using the language specification given as a collection of Montages, Gem-Mex generates an interpreter for the language. The core of the Gem-Mex system is *Aslan* - the Abstract State Machine language, which is a full implementation of the ASM framework. The Montages specification is first transformed into Aslan code which is compiled by the Aslan compiler to generate C code, from which the final executable is obtained. This facilitates portability of the whole system.

Generating Visual Programming Environments. Besides the interpreter, Gem-Mex also generates a visual programming environment. This consists of an animation component and a debugger. During the compilation of Aslan code, special instructions are inserted which provide the necessary information required to visualize the execution of a source program. the animation and debugging features include step-wise execution, definition of break-points, interactive term evaluation, and replay of executions among others. Apart from using these facilities purely for debugging, they are also useful for the user to understand and learn about the Montages framework. Figure 4 shows screen-shots of the console window, portions of the debugger, the animations, and also a window of the editor showing the Montage *Expression* used in HYSDEL.

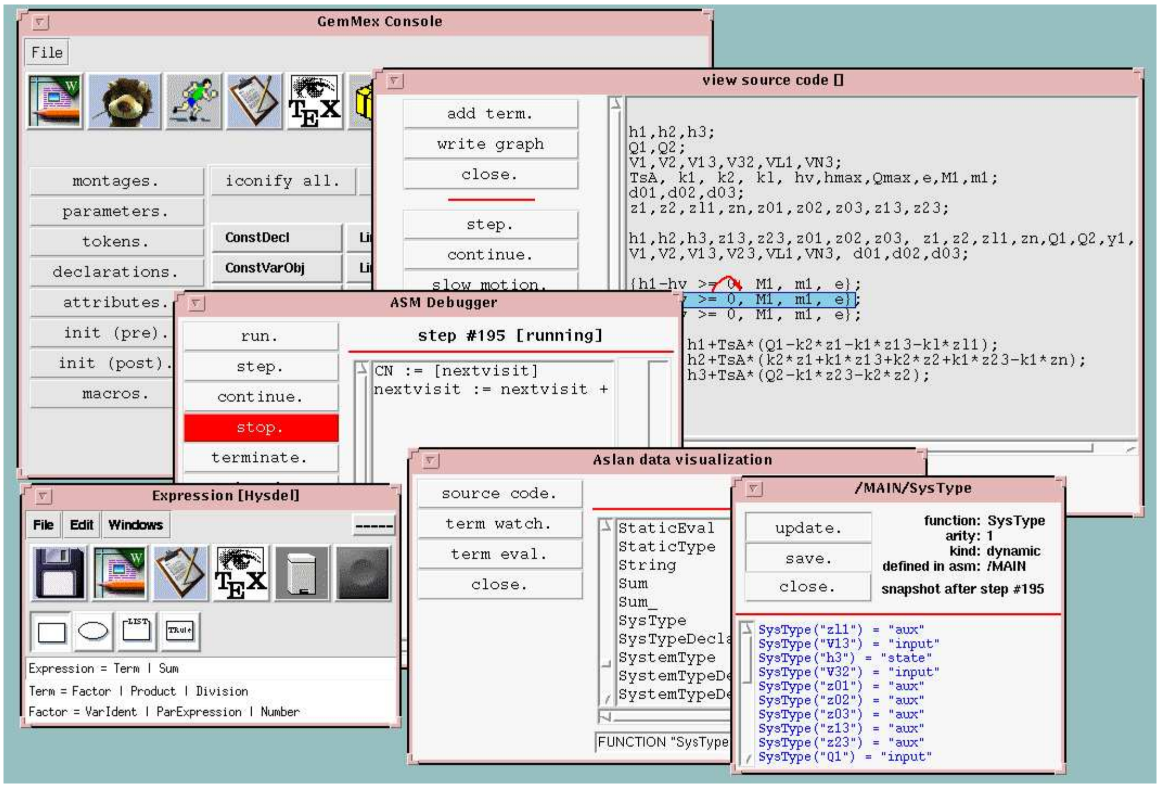


Figure 4: The Gem-Mex Environment

Generating Documentation. The Gem-Mex system also automatically generates files which can be used for documenting the language specification. For example LaTeX and HTML files for the full specification can be generated at the click of a button. All the Montages shown in this paper were generated using Gem-Mex.

Library of Programming Language Features. A concept for providing libraries of programming language features is currently under development. Since the different Montages are compositional, it is possible to reuse them in the design of a new language. Commonly usable features are arithmetic expressions, function calls, parameter passing techniques, etc.. The designer of a new language can import such already defined features and possibly customize them for particular needs. Such customizations might even be as simple as changing changing the names of some variables so that they better reflect the problem problem domain.

3 HYSDEL - A Control-Oriented Language for Describing Hybrid Systems

In this section we briefly introduce the domain of our case-study : hybrid systems in the context of control engineering, and introduce the Mixed Logical Dynamical form. Systems are specified using HYSDEL in this form. Some related software tools and languages are then mentioned, providing a motivation for the design of HYSDEL. Finally we describe the semantics of HYSDEL and its implementation in the Montages framework.

3.1 Hybrid Systems in Control and Systems Theory

In engineering, traditionally the concept of a *model* of a system is associated with differential or difference equations. Such equations are typically derived from the physical laws governing the *dynamics* of the system. Consequently, most of control theory and related tools have developed on the basis of such models. On the other hand, in many applications the system to be controlled or supervised is also constituted by parts described by *logic*. Examples are on/off switches or valves, and gears or speed selectors. Here the evolution of the system is dependent on *if-then-else* rules. Often, the control of these systems is left to schemes based on heuristic rules inferred from practical plant operations.

Recently, researchers are formally investigating *hybrid systems*, namely hierarchical systems constituted by dynamical components at the lower level, governed by logical/discrete components on the upper level [9, 14]. However, in some applications a precise distinction between the different levels is not possible, specially when the dynamical and logical facts are largely interdependent. In such cases it is not clear how such systems can be modelled systematically. To address this problem, in [8, 29] it was shown that expressing logical propositions in the form of linear constraints on integer variables leads to a powerful modelling framework - the *Mixed Logical Dynamical* (MLD) form.

A wide variety of systems commonly studied in control theory, such as discrete event systems, piece-wise linear systems, systems with discrete inputs, and many more [8], can be specified in the MLD form. Such MLD systems form a very natural problem domain. To describe such systems so that it is intelligible to control engineers, the language HYSDEL - HYbrid System DEscription Language, was developed by the Automatic Control Laboratory at ETH Zurich. A HYSDEL source code describes in a natural way the logical and dynamical interactions existing in a hybrid system in the form of propositional logic, difference equations, and constraints. This description has to be translated into MATLAB code which is then used for simulation and optimization.

3.2 Related Software and Languages

There are various languages and software packages for the description, simulation, and verification of hybrid systems. Among the more commonly used are HyTech [16], SHIFT [11], Simulink/StateFlow [26], and Spin [17]. However, these are focussed on simulation or verification problems and do not support systematic controller design or estimator design procedures as desirable in control systems practice. With systems specified in HYSDEL, the MLD framework allows such problems to be solved with a reasonable effort. Stated in this form, problems can then be posed as optimization problems like mixed integer-linear programs or mixed integer-quadratic programs. These mathematical problems have been well studied and efficient solvers for them exist.

3.3 The Syntax and Semantics of HYSDEL

A HYSDEL description of a system consists of a number of variables of the type *state*, *input*, *output*, *auxiliary*, and *fault*. Each of these *system variables* are independently either of the type *real* or *logic*. A program is a collection of different types of relations, constraints, and updates, using these variables and describe the behavior of a system. Any such program can be interpreted as a collection of three linear relations :

$$x(t+1) = Ax(t) + B_1 u(t) + B_2 \delta(t) + B_3 z(t) + B_4 f(t) \tag{1a}$$

$$y(t) = Cx(t) + D_1 u(t) + D_2 \delta(t) + D_3 z(t) + D_4 f(t) \tag{1b}$$

$$E_2 \delta(t) + E_3 z(t) \leq E_1 u(t) + E_4 x(t) + E_5 + E_6 f(t) \tag{1c}$$

These three linear relations define a MLD system. Here A, B_1-B_4, C, D_1-D_4, and E_1-E_6 are matrices and x, u, y, δ, z, and f are vectors representing the state, input, output, auxiliary boolean, auxiliary real, and the fault variables respectively. A translator from HYSDEL to MATLAB has to construct these matrices and vectors from a HYSDEL program based on the rules described in [8], and output them in MATLAB format. The rules used are based on propositional calculus and integer linear programming. In addition, it has to check the syntax, do type checking using both the system and the static types, check the degree constraints of the relations, check for possible constraint violations, and finally also create a log-file with the list of errors and warning messages.

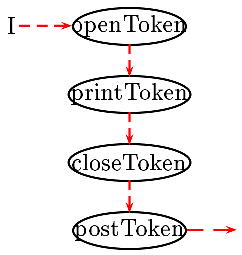
```
% Description of variables and constants
state h1,h2,h3;                            % Tank levels
input Q1,Q2;                               % Input flows
input V1,V2,V13,V32,VL1,VN3;               % Valves
const  A,Ts, k1, k2, kl, hv,hmax,Qmax,e;  % Constants

% Variable types
real  h1,h2, h3,z13,z32,z01,z02,z03, z1,z2,zl1,zn,Q1,Q2;
logic V1,V2,V13,V23,VL1,VN3,d01,d02,d03;

% Relations
d01 = {h1-hv >= 0, M1, m1, e};
z1  = V1*(z01-z03) {hmax-hv, hv-hmax, e};

% Other constraints
must h1 <= hmax;

% Update
update h1 = h1+Ts/A*(Q1-k2*z1-k1*z13-kl*zl1);
```

Figure 5: A toy HYSDEL program.

A toy HYSDEL program is shown in Figure 5. In this the state vector x, for example, is $\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$, and the *update* relation results in one row of the matrices occurring in the Equation (1a). Since on the right hand side of this update, the coefficient of the state variable h_1 is 1, and those of h_2 and h_3 are 0, the resulting row in the matrix A 1, 0, 0. A real HYSDEL program will contain such update statements for each *state* and *output* variable. The *relations* and *constraints* shown in the example are transformed into matrices in (1c) by applying the required rules. There can also exist products of variables. Since the MLD form described by (1) is linear, there are procedures to transform products into linear relations with the introduction of additional variables and inequalities. This whole procedure is executed by the translator and the newly introduced variables and inequalities also appear in the final MATLAB code.

In addition to the basic constructs mentioned, *sub-systems* can also be defined in a similar spirit in which procedures are defined in high level programming languages. The variables defined within a sub-system are local to it, and a number of sub-systems can be interconnected to specify a larger system. Moreover, there can be different ways in which sub-systems can be connected, for example cascade, parallel, or feed-back. There can also be predefined libraries of sub-systems such as piece-wise linear systems, hysteresis automata, and bilinear systems.

Program ::= **{Declaration ";"}**
{Relation ";"}
{Update ";"}



MatrixRowNo := 1
MatrixRowNoDueToNewVars := 1
–

@openToken:
_%openfile
@printToken:
!PrintMatrices
@closeToken:
_%closefile
@postToken:
_%postprocess

Figure 6: The Montage *Program*

3.4 Implementing HYSDEL with Montages

The translation process involved in implementing HYSDEL mainly makes use of the static aspects of Montages. The static analysis builds up the matrices and the vectors mentioned in the last subsection, and then a pretty printing to output these in MATLAB format is modelled as the dynamic semantics. This is illustrated in Figure 6 in the Montage *Program*, extracted from the HYSDEL specification. The dynamic semantic rule `PrintMatrices` is an ASM written in Aslan, and the other actions like opening and closing of files are written in C. In general, any external programs can also be called.

Our main driving force in implementing HYSDEL is to obtain a specification that can be understood by control engineers with a reasonable effort, which they can modify according to their needs, and have a working prototype at all points of time. Assuming a moderate level of programming expertise, the only additional concept that is required to be understood is that during a depth-first traversal (which is sufficient for HYSDEL) of the AST, the rules specified in the *static semantics area* of a Montage are executed at the nodes corresponding to the Montage. Typically such rules update some attributes or global functions. In general some of the attributes are evaluated when the node is first visited and the others in the second visit (when all the children have been visited).

As we have already mentioned in Section 2, both the static and the dynamic semantic rules are evaluated as ASM rules and have to be stated in Aslan - the programming language for implementing ASMs. However, from the users' point of view, it is not required to understand the implications of this underlying ASM framework. It is sufficient only to realize that at each node all the rules are evaluated in parallel. Moreover, the syntax of Aslan is extremely intuitive and anyone familiar with a high-level programming language like C, C++, or Pascal, should have no problems in adapting to it. For example, the Montage *Division* shown in Figure 7 evaluates the attributes `CoeffMap` and `Constant`. Here "+" denotes denotes string concatenation, `CoeffMap(v)` returns the coefficient of

Division ::= **Term "/" Factor**
– Evaluate the constant term *if (S-Term.Constant $\neq$ undef) then*   *Constant := "(" + S-Term.Constant + ")/(" + S-Factor.Constant + ")"* *endif* *– Evaluate the coefficients of the variables* *do forall v in SysTypeDeclared*   *if (S-Term.CoeffMap(v) $\neq$ undef) then*   *CoeffMap(v) := "(" + S-Term.CoeffMap(v)*   *+ ")/(" + S-Factor.Constant + ")"*   *endif* *enddo*
condition *not (S-Factor.Constant = undef)*

Figure 7: The Montage *Division*

the variable `v`, and `Constant` returns the constant term of an expression, both in the form of strings.

The rules specified in each Montage are on the same abstraction level of the problem domain, and the data structures used (the global functions and attributes) directly reflect the problem structure. As an example, Figure 8 shows the Montage *Declaration* which follows from the Montage for *Program*. The second box in this Montage shows the static semantic rules. Here, `SysTypeDeclared` is a unary relation, `SysType` is a unary function mapping variable names to their *system types* such as `state`, `input`, etc., and similarly for `TypeDeclared` and `StaticType`.

By using the powerful animation facilities of the Gem-Mex tool, it is possible to visualize during run-time the rules being executed at each node of the AST, and given any part of the MATLAB code it is possible to identify the HYSDEL code and the rules which are responsible for generating it. Additionally, it is also possible to track how any function or attribute evolves during execution.

It is natural to argue that this scheme of specifying translations using Montages is not new and uses simple, well established techniques from syntax-directed translation schemes and attribute grammars. However, the underlying framework which helps to structure the small subparts of the specification together in a smooth way is novel. This is helpful for the domain engineers to handle the whole specification without going into any details of compiler construction techniques. Because of the ongoing research on MLD systems, the design of HYSDEL is still unstable and new modifications are being introduced frequently. With the modular structure of the Montages specification, it is being possible to incorporate such changes leading to a stepwise refinement approach. A full specification of HYSDEL along with example problems is freely available [3].

4 Specifying Dynamic Semantics Using Montages

Although a large number of DSLs require a source-to-source translation there also exist applications where a DSL might be directly implemented instead of translating it to another high-level language. One of the major motivations for translating a DSL into a high-level language like C or COBOL is to provide portability over a large class of architectures while achieving efficient implementation through the use of optimizing compilers existing for these languages. However, such a scheme might be error prone.

Specifying a translation scheme from a DSL to another language usually involves a shift in the abstraction level naturally existing in the DSL. Hence, in situations where maintenance and easier specification has a preference over efficiency, using the Montages framework to directly implement a

Declaration	**::=**	**Type Ident {"," Ident}**
Type	=	SystemType |StatType |"const"
SystemType	=	"state" |"input" |"output" |"aux" |"fault"
StatType	=	"real" |"logic"

```
if S-Type.SystemType then
    vary i over list S-Ident
        SysTypeDeclared(i.Name) := true
        SysType(i.Name) := S-Type.Name
    endvary
else                    – StatType or constant
    vary i over list S-Ident
        TypeDeclared(i.Name) := true
        if S-Type.Name = "const" then
            StaticType(i.Name) := "real"
        else
            StaticType(i.Name) := S-Type.Name
        endif
    endvary
endif
```

Figure 8: The Montage *Declaration*

language will be helpful. Moreover, Montages uses simple graphical constructs to specify dynamic semantics which are very intuitive.

A real application where the dynamic semantic aspects of Montages have been used is in a data-warehousing application at the Union Bank of Switzerland (UBS) [24]. For connecting different heterogeneous databases, a driver-specification language called Corellation Modelling Language (CML) was designed. Here, one of the main requirements was an easy and transparent implementation which the database administrators could maintain. In this case the dynamic semantics of CML is specified using Montages and finally SQL code is generated as the output.

A second recent application is in providing tool support for Action Semantics (AS) [27]. Interestingly, Action Semantics has similar goals as that of Montages, like intelligible and pragmatic specifications. The tool support for AS is based on the ASF+SDF system [32] and is generally found to be difficult to maintain and extend. Recently it was shown that Action Semantics can be specified using Montages [22]. Using the tool support Gem-Mex the implementation is accessible to the user and provides a direct link (both formally as well as visually) between the specification and the execution of an AS program.

5 Conclusions and Future Directions

In this paper we have advocated the use of Montages for implementing Domain Specific Languages. It is associated with a number of pragmatic qualities which enable domain engineers to extend and maintain a language themselves - a feature which vastly improves the usability of DSLs. We believe that at least in the academia, and possibly also in industry, a number of applications demand the use of small, simple DSLs whose design constantly change based on upcoming needs. The maintenance of such DSLs are severely restricted because of dependence on specialized software personnel. Since any specialized background in semantics or compiler construction can not be generally expected from the domain engineers, an implementation scheme that uses minimal additional expertise, addresses this problem to a considerable extent. We have found Montages to be largely successful in such situations till now.

Specification formalisms with similar goals like ASF+SDF have also been used for implementing DSLs. There also exist several compiler construction tool-kits like KHEPERA [12] specifically targeted towards DSL implementation. However, our experience is that both such schemes require too much specialized background, compared to what is normally available.

Montages is an ongoing project [4]. Apart from its use in HYSDEL, it has also been used to implement a DSL for data warehousing applications and is being used by the Union Bank of Switzerland. Recently Montages was also used to specify Action Semantics with the goal of providing a flexible implementation of Action Semantic specifications.

ASF+SDF has been used in the design of DSLs like RISLA and RISQUEST. These languages are being successfully used in the Dutch banking industry for the last couple of years. Montages has still not been used in projects of this size. Although we do not foresee any bottlenecks either of the underlying theoretical framework or the tool support, it is yet to be seen how Montages scales in projects of this magnitude. In such cases issues like efficient implementation will also have to be taken into consideration, and are currently being investigated.

References

[1] The Gem-Mex Tool Homepage, `http://www.first.gmd.de/~ma/gem`.

[2] The Abstract State Machine Homepage, `http://www.eecs.umich.edu/gasm/`.

[3] HYSDEL - A Control-Oriented Language for Describing Hybrid Systems, `http://www.tik.ee.ethz.ch/~samarjit/HYSDEL.html`.

[4] The Montages Project Homepage, `http://www.tik.ee.ethz.ch/~montages/`.

[5] *Proc. Algebraic Methodology and Software Technology*. Springer Verlag, 1996. LNCS 1101, pages 9-18.

[6] *Proc. First USENIX DSL Conference*, October 1997.

[7] B. R. T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *Workshop on Formal Methods Application in Software Engineering*, 1995.

[8] A. Bemporad and M. Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica*, 35(3):407–427, March 1999.

[9] M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control : model and optimal control theory. *IEEE Trans. on Automatic Control*, 43(1):31–45, January 1998.

[10] J. C. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, 1988.

[11] A. R. Deshpande, A. Göllü, and L. Semenzato. *Shift Reference Manual*, 1998. `http://www.path.berkeley.edu/shift`.

[12] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. KHEPERA : A system for rapid implementation of domain specific languages. In *Proc. First USENIX DSL Conference* [6].

[13] Josef Grosch. Are Attribute Grammars used in industry ? In *Proc. Second Workshop on Attribute Grammars and their Applications*, 1999.

[14] R. L. Grossmann and H. Rischel, editors. *Hybrid Systems*. Springer Verlag, New York, 1993. LNCS 736.

[15] Yuri Gurevich. Sequential ASM thesis. Technical Report MSR-TR-99-09, Microsoft Research, February.

[16] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[17] G. H. Holtzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[18] Paul Hudak. Modular domain specific languages and tools. In *Proc. The Fifth International Conference on Software Reuse*, Canada, June 1998. IEEE CS Press.

[19] I. Jacobs. The Centaur 1.2 Manual. Technical report, INRIA, Sophia-Antipolis, 1992.

[20] G. Kahn. Natural semantics. In *Proc. Symp. on Theoretical Aspects of Computer Science*, Passau, Germany, 1987. Also available as Research Report RR-601, INRIA, Sophia-Antipolis, February 1987.

[21] Paul Klint. A meta-environment for generating programming environments. *ACM Trans. on Software Engineering and Methodology*, 2:176–201, 1993.

[22] P. Kutter and A. Pierantonio. Generating an action notation environment from montages descriptions. Invited Talk at AS'99: Second International Workshop on Action Semantics, Amsterdam, March 1999. To be published as BRICS Notes Series.

[23] P. W. Kutter and A. Pierantonio. Montages: Specification of realistic programming languages. *J.UCS, Springer*, 3(5):416 – 442, 1997.

[24] P. W. Kutter, D. Schweizer, and L. Thiele. Integrating formal Domain-Specific Language design in the software life cycle. In *Current Trends in Applied Formal Methods*, LNCS. Springer Verlag, October 1998.

[25] The MathWorks Inc., Natick, Mass. *MATLAB Reference Guide*, 1997.

[26] The MathWorks Inc., Mass. *Stateflow User Guide*, 1997.

[27] Peter D. Mosses. *Action Semantics.* Cambridge Tracts in theoretical Computer Science. Cambridge University Press, 1992.

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

[29] M. L. Tyler and M. Morari. Propositional logic in control and monitoring problems. *Automatica*, 1999. In print.

[30] 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. Algebraic Methodology and Software Technology* [5]. LNCS 1101, pages 9-18.

[31] A. van Deursen. *Executable Language Definitions : Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[32] A. van Deursen and P. D. Moses. ASD : The Action Semantic description tools. In *Proc. Algebraic Methodology and Software Technology* [5], pages 579–582. LNCS 1101, pages 9-18.

[33] Arie van Deursen and Paul Klint. Little languages: little maintenance ? In *Proc. First ACM SIGPLAN Workshop on Domain Specific Languages*, Paris, January 1997. Published as University of Illinois at Urbana Champaign Computer Science Report.