Using Domain-Specific Languages for the Realization of Component Composition

Matthias Anlauff[1], Philipp W. Kutter[2],
Alfonso Pierantonio[3], and Asuman Sünbül[4]

[1] GMD FIRST, Rudower Chaussee 5
D-12489 Berlin, Germany
ma@first.gmd.de
[2] Swiss Federal Institute of Technology
Gloriastr. 35, CH-8092 Zürich, Switzerland
kutter@tik.ee.ethz.ch
[3] Dipartimento di Matematica, Pura ed Applicata
Università di L'Aquila, I-67100 L'Aquila, Italy
alfonso@univaq.it
[4] Computer Science Department, Technical University Berlin
Einsteinufer 17, Sekr. EN-7, D-10587 Berlin, Germany
asu@cs.tu-berlin.de

Abstract. In recent years, component-based development has evolved to one of the key technologies in software engineering, because it provides a promising way to deal with large scale software developments. Due to this, the realization of component interactions has become an important task while implementing a system being assembled from (existing) components. Scripting languages, like Perl, Tcl, Unix-Shell, are often used for implementing this so-called *glue code*, because they provide a flexible way to process string-based input, the most common data structures used for component interactions. However, often it turns out that the algorithms of the component interactions are too sophisticated to be adequately expressed in this kind of languages. In this paper, we propose the use of language technology for that purpose: the strings passed between the components are treated as sentences in specialized component interaction languages (CIL). The syntax of such a language defines the input format of the components to be interconnected, and the semantics represents the functionality of the glue code. The appropriateness of this approach depends on the methodology and support functionality available for designing these languages. We report on the positive experiences using *Montages* as methodology. We will also describe the support functionality of the Gem-Mex tool containing its graphical animation and debugging facilities, which can be used as vehicle for the comprehension of the interplay between the components of the overall system.

1 Introduction

The need for designing new programming languages is generally restricted to those cases, where special applications require non-standard language constructs.

These so-called Domain-Specific Languages (DSL) are usually designed for being used by domain experts who need languages consisting of terms and notions they are familiar with. Often, a DSL also diminishes the amount of source code, because domain-specific knowledge is already contained in the semantics of the language constructs. However, the design overhead is often the reason, why the definition of a DSL is chosen as the last alternative for solving a given problem. This fact also prohibits specialized programming languages being exhaustively used in other situations than the processing of human-written source code. In this paper, we will present a "language design environment" which – as we think – decreases the threshold of defining DSLs for solving given problems. Especially, if the transformation from source to target is a non-trivial task, the use of DSLs often leads to much better and less ad-hoc solutions.

In other words, the proposal of using syntax and semantics descriptions of DSLs for implementing algorithms can be seen as a novel style of programming: *Programming On Syntax Trees (PoST)*. That means that the input data defines the locations of computation by the nodes of the parse tree, each of these nodes is connected with a set of actions representing the computation, and the program is given as the union of all these actions occurring in a parse tree.

The method, we are using for supporting this new paradigm is *Montages* [17, 3]. Unlike other methods for describing formally the syntax and semantics of programming languages, Montages is based strictly on techniques well known by programmers: EBNF, finite state machines, imperative code. We will show, how the Gem-Mex system automatically generates interpreters based on language definitions given as Montages specifications. Montages has a formal semantics based on Abstract State Machines [12, 15], which makes it also interesting when security aspects of the target system must be considered.

In this paper, we will use the application domain of component-based software development in order to show that designing new specialized languages in our framework can compete with the technologies usually chosen in this domain.

This paper is organized as follows: We will first briefly describe some aspects of the composition of software components and will thereby categorize our approach in the work already done in this field. Section 3 explains the running example of the paper. We will then introduce the Montages method and briefly describe the Gem-Mex system. Section 7 concludes the paper and provides an outlook on future work.

2 Composition of software components

A lot of work has been done regarding the composition of software components. This fact is expressed by the large number of publications related to this issue. We will try to position ourselves wrt. the state of the art in component composition technology using a very recent publication by Schneider and Nierstrasz [21] where they distinguish between five techniques that are used for the realization of software component composition, namely[1]

[1] citation from the mentioned paper

- *component frameworks* provide software components that encapsulate useful functionality;
- *architectural description languages* explicitly specify architectural styles in terms of interfaces, contracts, and composition rules that components must adhere in order to be composable;
- *glue* abstractions adapt components that need to bridge compositional mismatches;
- *scripting languages* are used to specify compactly and declaratively how software components are plugged together to achieve some desired result [19]; and
- *coordination models* provide the coordination media and abstractions that allow distributed components to cooperate [6, 9].

In this paper, we propose the use of domain-specific languages as a substitute for *scripting languages* for those cases, where it would be clumsy to use this kind of languages. We do *not* want to use DSLs for coordinating the architectural design of the composed software system, as proposed in [10]; we will focus on the use of DSLs for non-trivial component composition tasks on the implementation level. A more detailed discussion on aspects regarding the design of software systems that are assembled from components can also be found in [5] and [22].

2.1 Component interaction technologies

Important requirements for the technology that shall be used for realizing component interaction are *flexibility* and *adaptability* (see [11]). Flexibility in this context means the ability to process and generate any format of data occurring in a heterogeneous environment. Adaptability describes the possibility of the technology to adapt it for certain application domains. Therefore, scripting languages, like Unix-Shell-Script, Tcl/Tk, Python, Perl, are often used for the implementation of component interactions, because they meet these requirements. In [20], the importance of scripting language for the component interaction culminates in the statement: "The main purpose of a scripting language is to wire components in other languages".

In trying to analyze this elusive role of scripting languages for component interaction technology and knowing the drawbacks of this kind of languages concerning concepts of high-level programming languages, one might come to the conclusion, that the uncomplicated way of dealing with input and output data is the key factor for this rating. Most scripting languages provide very easy to use input/output primitives on a higher abstraction level than usually programming languages do. The data types that can be handled by scripting languages are restricted to one or two basic types (e.g. strings and integers); for the purpose of implementing component interaction this is usually sufficient.

2.2 The implementation of component interaction using DSLs

At the risk of oversimplifying somewhat, the functionality of the code implementing the component interaction can be characterized by two basic tasks:

processing data coming from and sending data back to components. The prerequisite for applying the "PoST" style of programming is therefore given for this kind of application: the execution of the code is mainly controlled by the structure of the input data. The format of this data defines the syntax of a *Component Interaction Language* (CIL), and the functionality is expressed by the language semantics given by the actions attached to the nodes of the resulting parse trees. Thus, the component interaction code can be regarded as an interpreter for the CIL. Figure 1 illustrates this approach: The components $C_{i_1} \ldots C_{i_n}$ send strings

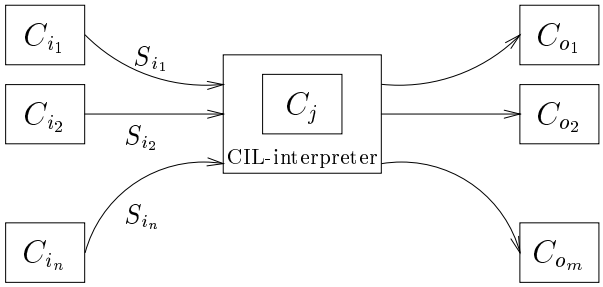


Fig. 1. Composition using a component interaction language

$S_{i_1} \ldots S_{i_n}$ to the component C_j, each of these strings is a sentence of the component interaction language CIL. The interpretation of these sentences in the CIL interpreter results in the output of data that is send back to the components.

The components on the left-hand-side and on the right-hand-side both represent subsets of the set of components occurring in the system.[2]

2.3 Domain-specific vs. scripting languages

The situations in which we propose to use DSLs as a substitute for scripting languages are those where non-trivial or at least non-linear transformations of the data exchanged between the components must be performed. On the one hand, we do not propose to get rid of scripting languages at all; this would be a big mistake because scripting languages have been proven to be a very good vehicle for a high percentage of component interaction problems. But on the other hand, often the expressive power of scripting languages does not suffice to provide an appropriate solution for a given composition problem. In these cases we propose to use domain-specific languages as an alternative component composition implementation techniques. Our running example introduced below will show a typical situation were we think that the overhead of designing an own language solely for the task of component interaction is justified. We will also show how this overhead can can diminished by using the Montages method,

[2] C_{i_j} may represent the same component as C_{o_k}, and both may coincide with C_j

a semi-visual framework for specifying syntax and semantics of programming languages, and its tool environment Gem-Mex.

3 Running Example: Transformation of Object Repository Decription Formats

As a running example for explaining our proposal we use the interaction of a object-based tool (e. g. a graphical editor) and a data repository. We assume that the tool wants to store its objects in the repository in order to make use of the configuration management functionality provided by the repository system. As it is common in practice, the meta-models of both components do not match so that the objects of the tool can not be stored in the repository as they are. In our special case, the tool has a notion of sub-classing while the repository system has not. In order to correctly reflect the interrelations of objects and their attributes, the inheritance relation should also be represented in the repository.

For illustration purposes, we assume, that the classes occurring in the object-based tool can consist of a list of attributes each of which has one of the basic types "int", "float", "string", or "bool". Furthermore, we assume, that a class has at most one super-class. For making things more concrete, we will further assume that the grammar for describing the object format of the object-based tool is given as follows:

Program	::= { ClassDefinition ";" }
ClassDefinition	= BasicClassDefinition | SubclassDefinition
BasicClassDefinition	::= "class" Ident "{" { Attribute ";" } "}"
SubclassDefinition	::= "subclass" Ident "of" Ident "{" { Attribute ";" } "}"
Attribute	::= Ident ":" AttrType
AttrType	= "Int" |"Float" |"String" |"Bool"

The repository component is only able to declare classes and attributes, no sub-classing is defined in the repository. The task of the DSL is to read the object definitions from the tool and generate for each object the corresponding entry in the repository. The attributes of an object's super-class must also be inserted for each object. Thus, the DSL must "flatten" the inheritance relationship introduced by the object-based tool, so that the corresponding objects in the repository contain the correct list of attributes.[3]

In the following, we will use this example to introduce the Montages approach, and demonstrate the functionality of the Gem-Mex system.

[3] This example is inspired by a problem that arose in the context of the "KobrA"-project which currently runs with industrial partners at GMD FIRST. The approach described in this paper is used to extend the repository system developed by one of the industrial partners to deal with sub-classing.

4 Visual Formal Semantics Descriptions: Montages

Montages [17, 2] 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.

In the same way Montages is used to describe the syntax and semantics of programming languages, it can be used to formulate application functionality based on the structure of input data. The difference lays on the conceptual level: While in the first case the *language* and its semantical description lays in the center of interest, in latter case one focuses on the *application* that processes data of a given format. Thus, the term "language" does not necessarily refer to a programming language, but may also refer to the description of the format of the input data of an application.

The mathematical semantics of Montages is given with Abstract State Machines (formally called Evolving Algebras) [13, 15]. 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 [14], Occam [7], C++ [23], Oberon [16], and Java [8] 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 ASMs 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 control and data flow, starting from the concrete syntax.

In terms of the "PoST"-programming style this means that during the first phase – the static analysis – the locations of the computation – associated with a subset of the nodes of the syntax tree – are linked with each other using control flow arrows. Data access to other parts of the syntax tree is defined by data arrows. In the second phase – the dynamic execution – these locations

are visited depending on firing condition on the flow arrows. In contrast to traditional programming languages, where the control flow graph is fixed by the source program, in our approach this graph is first constructed depending on the structure of the input data and then processed according to the rules and actions given for each node of that graph.

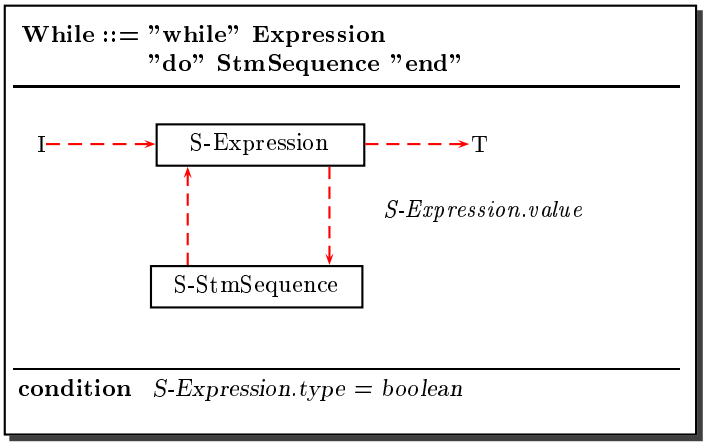


Fig. 2. The While Montage

In order to get an idea on how the the syntax and semantics is described for a typical language construct in an imperative language, Figure 2 contains the Montages specification of a "While" statement. 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 syntax tree and flow graph. In the While-Montage the tree with the expression and the statement sequence descendants (denoted by the selectors S-Expression and S-StmSequence) is mapped to a control cycle. The dotted control flow arrows may be labeled by means of firing conditions, i.e. predicates which determine through which edges the control must flow. If the firing condition S-Expression.value evaluates to true, control cycles, otherwise it leaves the construct through the exit point denoted by the T node (terminal). The entry point for the control is the I node (initial).

Unlike the While construct, most other Montages contain additional points of computation. Such points, or *actions nodes* are visualized as labeled ovals. The action to perform is given in the fourth part (not contained in Figure 2) of a Montage, by using the label as reference. We will see examples of this in the following section.

5 Implementing Component Interaction using Montages

In this section, we will illustrate how Montages and its support environment Gem-Mex can be used to implement the interaction of components using the examples introduced in Section 3. According to the grammar specified there we have specified a Montage for each language construct of the DSL implementing the component interaction between the object-based tool and the object repository.

In Figure 3 the Montage for the attribute definition is given. The action performed when an attribute node is reached is given by the lower part of the window: The name and the type of the attribute is written to stdout in a format that is accepted by the repository system.

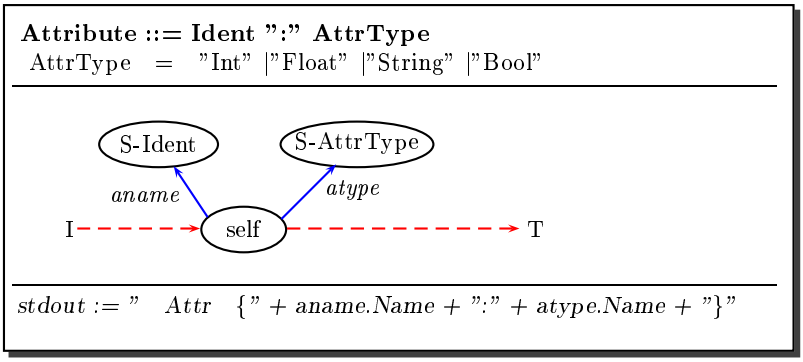


Fig. 3. The "Attribute"-Montage

In Figure 4, the Montage for a basic class definition is given. When the control flow reaches a basic class definition the return point is set to the "self" node. This is done, because the return point serves as register for storing the "next" class definition occurring in the input stream. The class definition header as it is expected by the repository system is printed to stdout. After the list of attributes has been processed, the control reaches the return point that has been set in the first action of this construct. This Montage represents the base case, only the local attributes are written to stdout.

The Montage in Figure 5 represents the most interesting part of this example, namely the case in which local and inherited attributes must be transfered to the repository system. The Montage is nearly identical to the "BasicClassDefinition"-Montage only that the control flow – after the local attributes has been processed – does not enter the return point immediately; instead the control flow is set to the "self" node of the super class. This node has been stored during the static analysis in the global table "ClassDef". From here, the inherited attributes are also reached and transfered to the repository component. The use of the "Return-Point" variable becomes clearer now: after processing the basic class definition

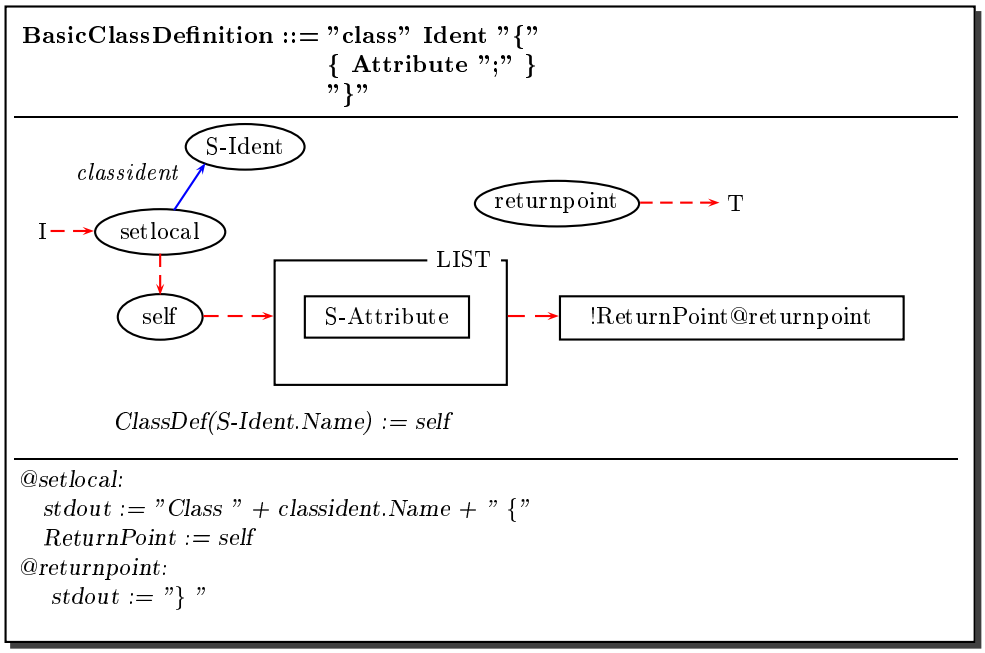


Fig. 4. The "BasicClassDefinition"-Montage

representing the root class for the current subclass, the control returns to the current class definition.

Example

In order to illustrate the functioning of the DSL described above, we have generated a language interpreter from the Montages described above using the Gem-Mex tool. Figure 6 contains an example input and the generated output. As one can see there, the attributes of the classes are flattened so that no information is lost in the repository.

6 Gem-Mex: The Development Environment for Montages

The development environment for Montages is given by the Gem-Mex tool [2, 4]. The intended use of the tool Gem-Mex is, on one hand to allow the designer to 'debug' her/his semantics descriptions by empirical testing of whether the intended decisions have been properly formalized; on the other hand, to automatically generate a correct (prototype) implementation of programming languages from the description, including visualization and debugging facilities.

Gem-Mex consists of a number of interconnected tools:

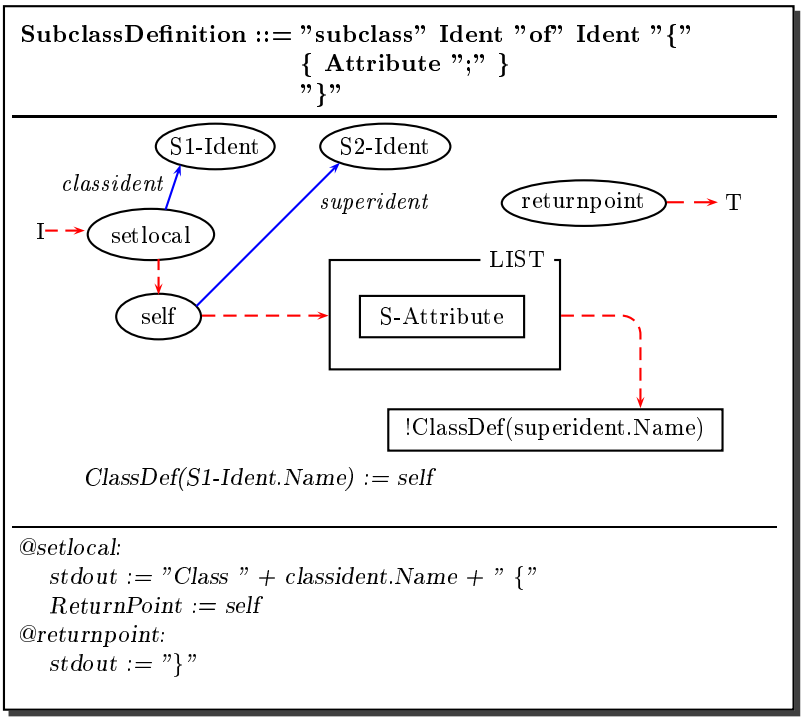
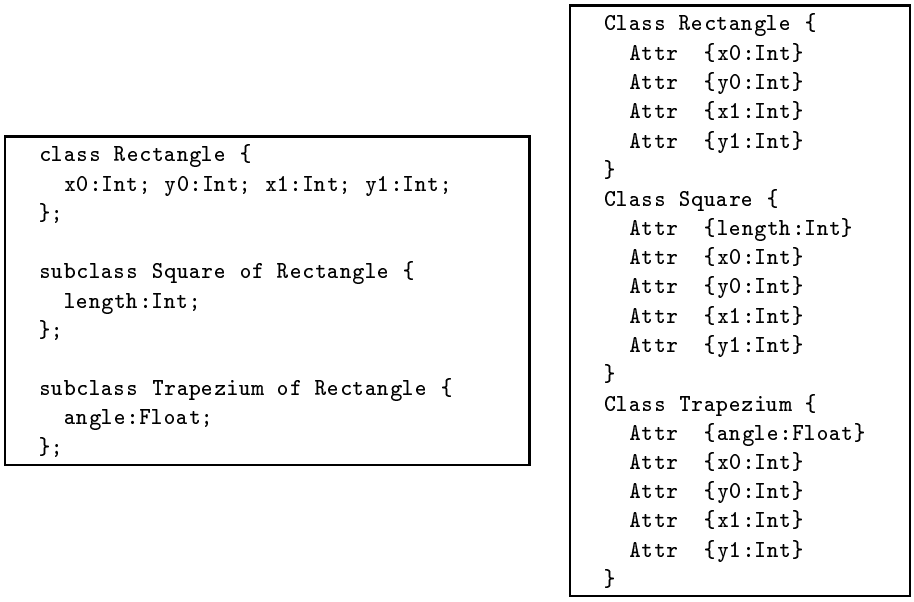


Fig. 5. The "SubclassDefinition"-Montage



```
class Rectangle {
  x0:Int; y0:Int; x1:Int; y1:Int;
};

subclass Square of Rectangle {
  length:Int;
};

subclass Trapezium of Rectangle {
  angle:Float;
};
```

```
Class Rectangle {
  Attr  {x0:Int}
  Attr  {y0:Int}
  Attr  {x1:Int}
  Attr  {y1:Int}
}
Class Square {
  Attr  {length:Int}
  Attr  {x0:Int}
  Attr  {y0:Int}
  Attr  {x1:Int}
  Attr  {y1:Int}
}
Class Trapezium {
  Attr  {angle:Float}
  Attr  {x0:Int}
  Attr  {y0:Int}
  Attr  {x1:Int}
  Attr  {y1:Int}
}
```

Fig. 6. Input (left) and output (right) of the DSL connector

- a specialized graphical editor allows to enter and manipulate Montages in a convenient way;
- frames for the documentation of the specified languages are generated automatically;
- the Montages executable generator (Mex) generates a correct and efficient interpreter of the language;
- the generic animation and debugger tool visualizes the static and dynamic behavior of the specified language at a symbolic level; source programs written in the specified language and user-defined data structures can be animated and inspected in a visual environment; a snapshot of the debugging process is shown in Fig.7.

6.1 Generation of Language Interpreters

Using nothing but the formal semantics description given by the set of Montages, the Gem-Mex system generates an interpreter for the specified language. The core of the Gem-Mex system is *Aslan*, which stands for *A*bstract *S*tate Machine *Lan*guage and provides a fully-fledged implementation of the ASM approach. Aslan can also be used as a stand-alone, general purpose ASM implementation. The process of generating an executable interpreter consists of two phases:

- The Montages containing the language definition are transformed to an intermediate format and then translated to an ASM formalization according to the rules presented in the previous sections.

– The resulting ASM formalization is processed by the Aslan compiler generating an executable version of the formalization, which represents an interpreter implementing the formal semantics description of the specified language.

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

6.2 Generation of Visual Programming Environments

Besides pure language interpreters, the Gem-Mex system is able to generate visual programming environments for the generated ASM formalization of the programming language semantics[4]. This is done by providing a generic debugging and animation component which can be accessed by the generated executable. During the translation process of the Montages/ASM code special instructions are inserted that provide the information being necessary to visualize the execution of the formalization. In particular, the visual environment can be used to debug the specification, animate the execution of it, and generate documents representing snapshots of the visualization of data structures during the execution. The debugging features include stepwise execution, textual representation of ASM data structures, definition of break points, interactive term evaluation, and re-play of executions.

Figure 7 shows an example of the graphical animation facility of the Gem-Mex system. On the right-hand-side of the window the source code program written in the specified programming language is displayed using position information generated during the compilation process of the Montages. This position information is used, for example, to highlight certain parts of the source code that correspond to values of data structures contained in the language formalization. In Figure 7, for example, the change of the value of the "current-task" function CT is animated by drawing an arrow from its old value to the new one. Similarly, after the rules of the static semantics are processed, the correspondence of the use occurrence of an identifier an its declaration position is visualized by drawing an arrow between the two positions in the "view source" window. Experiences show that especially this kind of animation is useful to explain and document the formal semantics as specified in the Montages.

6.3 Generation of Documentation Frames

The Gem-Mex system also generates files that can be used as frames for the documentation of the language specification. Both, paper and online presentation of the language specification are automatically generated:

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

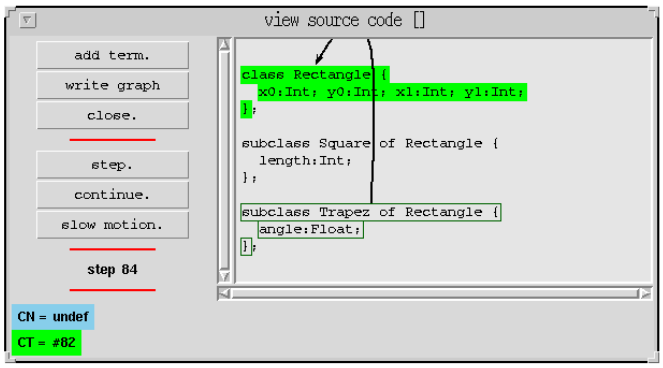


Fig. 7. Graphical animation in the Gem-Mex system

- LaTeX documents illustrate the Montages and the grammar; such documents are easily customizable for the non-specialist user; all Montages in this paper are generated by Gem-Mex;
- HTML versions of the language specification allows to browse the specification and retrieve pieces of specification.

7 Conclusion

The implementation of component interaction mainly consists of processing input data coming from other components, analyzing this data, starting internal computation and sending output data back. Because of their flexibility and adaptability, scripting languages are often used for this task. They usually provide regular expressions for analyzing incoming data streams. This technique is very powerful, if the transformations that must be applied to the data aren't too complex. Using scripting languages in these cases often leads to nearly unmaintainable code. In this paper, we propose an alternative for the implementation of component interaction for these cases. Our approach is based an experiences in the field of the formal description of programming languages semantics using the Montages method.

We explicitly do not want to get rid of scripting languages at all, because they have been proven to be very useful for most cases of component interaction problems. Nevertheless, we think that DSLs are appropriate for implementing component interactions requiring non-trivial or at least non-linear transformations of data that is exchanged by the components to be interconnected. This assumption only holds, if an environment like Montages/Gem-Mex is used diminishing the overhead of designing DSLs and generating interpreters for them.

Originally Montages are used to give syntax and semantics of programming languages: syntax defines the possible programs and semantics defines the static

and dynamic properties of programs. If used to implement component interaction, syntax defines the input format of components, while semantics is used to "code" the glue algorithms. We named this form of algorithm structuring the PoST programming style.

We have furthermore sketched, how the generated graphical tool environment, originally designed for debugging an animating the semantics of described programming languages, now serves as a tool for visualizing the component interactions.

In earlier work we designed two DSLs, which fall partly in the category of CIL. The Cubix DSL [18] is an interesting example for the proposed methodology. Cubix itself is used to initialize driver components. These components use as input CIL a query language for multi-dimensional data bases and generate as output corresponding SQL queries. The SQL-sentences are fed to a relational data base management system (DBMS). Hysdel [1] is designed in order to specify hybrid systems in a convenient way. Hysdel has two semantics: one is the direct simulation of the system for validation purposes. The second is the generation of the corresponding Matlab code. Hysdel together with the first semantics is used as a CIL for a simulation component, while the same language together with the second semantics is used as a CIL for a Matlab component.

As already mentioned in Section 3, we are applying our approach in the context of an research project with industrial partners. In this project, a commercial repository system should be connected with other components using an object definition language. First experiments with this industrial case study have shown, that our approach provides an efficient and elegant way to describe and implement component interaction, and that it represents a promising alternative to techniques traditionally used for the realization of component interaction.

References

1. M. Anlauff, A. Bemporad, S. Chakraborty, P. Kutter, D. Mignone, M. Morari, A. Pierantonio, and L. Thiele. From ease in programming to easy maintenance: Extending DSL usability with Montages, 1999. submitted for publication.
2. M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.
3. M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In A. D.Bjoerner, M.Broy, editor, *Perspective of System Informatics*, LNCS, 1999. to appear.
4. M. Anlauff, P. W. Kutter, and A. Pierantonio. The Gem-Mex tool homepage. `http://www.first.gmd.de/~ma/gem/`, 1997.
5. M. Anlauff and A. Sünbül. Software architecture based composition of components. In *GI-Workshop Sicherheit und Zuverlässigkeit software-basierter Systeme*, 1999.
6. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In Ciancarini and Hankin [9], pages 75–88.
7. E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: Simple Mathematical Interpreters. In U. Montanari and E. R.

Olderog, editors, *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.
8. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
9. P. Ciancarini and C. Hankin, editors. *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS. Springer Verlag, 1996.
10. C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10^{th} International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in LNCS, pages 170–194, Pisa, Italy, Sept. 1998.
11. F. Griffel. *Componentware*. dpunkt.verlag, 1998.
12. 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.
13. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
14. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.
15. J. Huggins. Abstract State Machines Web Page . http://www.eecs.umich.edu/gasm.
16. P. Kutter. Dynamic semantics of the programming language oberon. Technical report, ETH Zürich, 1997.
17. P. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5), 1997.
18. P. W. Kutter, D. Schweizer, and L. Thiele. Integrating formal domain-specific language design in the software life cycle. In *Current Trends in Applied Formal Methods*, LNCS. Springer, October 1998.
19. J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998.
20. J.-G. Schneider and O. Nierstrasz. Scripting: Higher-level programming for component-based systems. In *OOPSLA 1998*, 1998. Tutorial.
21. J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer, 1999.
22. A. Sünbül. Abstract state machines for the composition of architectural styles. In *Perspectives on System Informatics, PSI99*, 1999. to appear.
23. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.