Little Languages: Little Maintenance?

Arie van Deursen and Paul Klint

CWI, P.O. Box 94079, 1090 GB Amsterdam
http://www.cwi.nl/~{arie,paulk}/, {arie,paulk}@cwi.nl

December 16, 1996

Abstract

So-called *little*, or *domain-specific* languages (DSLs), have the potential to make software maintenance simpler: domain-experts can directly use the DSL to make required routine modifications. At the negative side, however, more substantial changes may become more difficult: such changes may involve altering the domain-specific language. This will require compiler technology knowledge, which not every commercial enterprise has easily available. Based on experience taken from industrial practice, we discuss the role of DSLs in software maintenance, the dangers introduced by using them, and techniques for controlling the risks involved.

1 Introduction

Little languages, tailored towards the specific needs of a particular domain, can significantly ease building software systems for that domain [Ben86]. To cite Hendon and Berzins [HB88],

> If a conceptual framework is rich enough and program tasks within the framework are common enough, a language supporting the primitive concepts of the framework is called for. (...) Many tasks can be easily described by agreeing upon an appropriate vocabulary and conceptual framework. These frameworks may allow a description of a few lines long to replace many thousand lines of code in other languages.

We will use the following terminology (see also Figure 1):

Domain-Specific Language (DSL) A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular problem domain [Wal96].

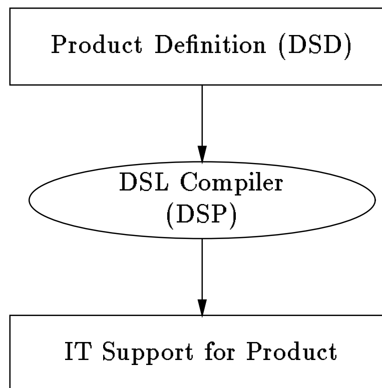

Figure 1: A DSL compiler.

Domain-Specific Description (DSD) A "program" (specification, description, query, process, task, ...) written in a DSL.

Domain-Specific Processor (DSP) A software tool for compiling, interpreting, or analyzing domain-specific descriptions.

A well-designed DSL will help the application builder to write short, descriptive, and platform-independent DSDs. Moreover, the good DSL will be effectively implementable, where the DSPs capture the stable concepts and algorithmic ingredients of the particular domain. Using such a DSL for constructing domain-specific applications, increases reliability and repairability, provides self-documenting and portable descriptions, and reduces forward (and backward) engineering costs [HB88].

In this paper, we elaborate on the advantages and problems of the use of domain-specific languages, emphasizing their role in software maintenance. Evidently, the attributes listed above will help reduce maintenance costs, and for that reason domain-specific approaches are investigated in order to arrive at "inherently maintainable software" [GB96]. However, using a domain-specific language can also make a system more difficult to maintain, for example if changes to the underlying domain model become necessary.

To discuss these issues, we first give an example of the commercial use of a DSL taken from the area of financial engineering (Section 2). We then cover the implications for software maintenance, and identify the risks and opportunities involved in the use of a DSL (Section 3). We conclude by describing two techniques (Sections 4 and 5) that will help to address two of the potential problems in the use of DSLs.

2 The Financial Engineering Domain

2.1 Interest Rate Products

Financial engineering deals, amongst others, with *interest rate products*. Such products are typically used for inter-bank trade, or to finance company take-overs involving triple comma figures in multiple currencies.[1] Crucial for such transactions are the protection against and the well-timed exploitation of risks coming with interest rate or currency exchange rate fluctuations.

The simplest interest rate product is the loan: a fixed amount in a certain currency is borrowed for a fixed period at a given interest rate. More complicated products, such as the *financial future*, the *forward rate agreement*, or the *capped floater* [Cog95, Chapter 12], all aim at *risk reallocation*. Banks can invent new ways to do this, giving rise to more and more interest rate products. Not surprisingly, different interest rate products have much in common, making financial engineering an area suitable for incorporating domain-specific knowledge in tools, languages, or libraries.[2]

2.2 Challenges

A software system supporting the use of interest rate products typically deals with the bank's financial administration (who is buying what), and — more importantly — provides management information allowing decision makers to assess risks involved in the products currently processed. Typical problems found in such systems are that it is:

- too difficult to introduce a new type of product, even if it is very similar to existing ones;
- impossible to ensure that the instructions given by the financial engineer are correctly implemented by the software engineer.

The first problem leads to a long time-to-market for new products;[3] the second leads to potentially incorrect behavior.

---

[1] As an example, Dutch PTT (KPN) recently bought the Australian company TNT for 2 billion Australian dollars. A clever cocktail of multi-currency loans, options, and swaps was used to finance this transaction, protecting KPN against interest rate differences and exchange rate fluctuations between the Australian and the Dutch financial markets.

[2] As an example, [EG92] describe the ET++ Swaps Manager, an object-oriented library for manipulating interest rate products.

[3] This can be very important: as an example, one Dutch bank decided mid-February 1996 to introduce a special one-day "leap year deposit" — a big success, but relying heavily on the flexibility of the bank's automated systems.

2.3 The Risla Language

Dutch bank MeesPierson, together with software house CAP Volmac saw the use of a specific language for describing interest rate products as the solution to the problems of long time-to-market and potentially inaccurate implementations. The language was to be readable for financial engineers, and descriptions in this language were to be compiled into COBOL. In this section we summarize earlier (and more detailed) accounts given in [Deu94, ADR95, BDK+96] of the development and use of this language.

The development of this language, called RISLA (for Rente Informatie Systeem Language — Interest rate information system language), started in 1992, and can be summarized as follows:

- MeesPierson had a very good library of COBOL routines for operating on cash flows, intervals, interest payment schemes, date manipulations, etc.;
- Using this library directly in COBOL did not provide the right level of abstraction, and cumbersome encoding tricks were needed to use, e.g., lists without a fixed length;
- An interest rate product can be considered as a "class": it contains instance variables to be assigned at creation time (the principal amount, the interest rate, the currency, etc.), information methods for inspecting actual products (when is interest to be paid), and registration methods for recording state changes (pay one redemption).

The language RISLA was designed to describe interest rate products along these lines. An instantiated product is called a *contract*, fixing the actual amount, rate, etc. of a particular product sold. The language is based on a number of built-in data types for representing cash flows, intervals, etc., and has a large number of built-in operations manipulating these data types (the operations correspond to the subroutines in the COBOL library). A product definition specifies the contract parameters, information methods, and registration methods.

RISLA is translated into COBOL. Other systems in the bank can invoke the generated COBOL to create new contracts, to ask information about existing contracts, or to update contract information. The initial version of RISLA was used to define about 30 interest rate products.

After a few years of working with RISLA, the users experienced the modularization features of RISLA as inadequate. A RISLA description defines a complete product; but different products are constructed from similar *components*. To remedy this situation, a project *Modular* RISLA was started. RISLA was extended with a small modular layer, featuring parameterization and renaming. Moreover, a *component library* was developed, and the most important products were described using this library.

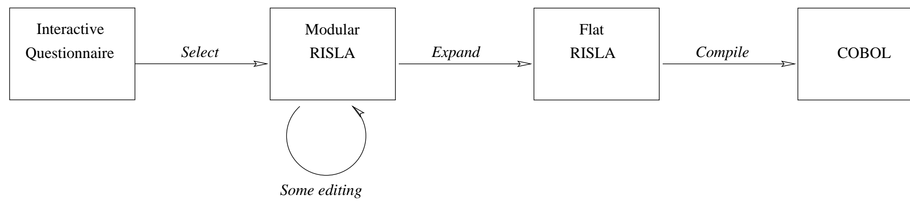

Figure 2: From questionnaire, via Modular and flat Risla, to COBOL

In addition to that, the RISLA development team made an effort to make the language more accessible to the financial experts. To that end, an inter-active *questionnaire* interface to the component library was developed. End-users can combine existing components into a new product by filling in the answers of a questionnaire.

This use of questionnaires and modular RISLA gives rise to the financial product life cycle as shown in Figure 2. An interactive questionnaire is filled in, and the answers are used to select the relevant RISLA components. This definition may contain some holes that are specific to this product, which can be filled by writing the appropriate RISLA code. The modular definition is then expanded to a flat (non-modular) definition, which in turn is compiled into COBOL.

As a last point of interest, the actual questionnaire used is defined using a second domain-specific language: RISQUEST. This is a language for defining questions together with permitted answers (choice from a fixed set, free text). Moreover, RISQUEST has constructs for indicating in which order questions are to be asked, and how this sequencing may depend on the actual answers given. Last but not least, RISQUEST can be used to associate library components with the possible answers. A RISQUEST definition is entered in textual form, and it is generated into a Tcl/Tk program. This program can be invoked by a financial engineer to fill in the questionnaire and to generate the corresponding modular RISLA.

2.4 Evaluation

At the positive side, the RISLA project has met its targets: the time it costs to introduce a new product is down from an estimated three months to two or three weeks. Moreover, financial engineers themselves can use the questionnaire to compose new products. Last but not least, it has become much easier to validate the correctness of the software realization of the interest rate products.

At the negative side, it is not so easy to extend the language. When a new data type or a new built-in function is required, the compiler, as well as the COBOL library, needs to be adapted. This requires skills in compiler construction technology, which is not the typical background of people working

mainly in a COBOL environment. Finally, the RISLA product definitions have become longer and longer. Whenever there was a new software system requiring information about products that was not provided in the existing methods, new methods had to be provided, sometimes requiring new data types or extensions to the RISLA language.

3 The Maintenance Perspective

3.1 Maintainability Factors

The literature on software maintenance (see, e.g., [Pig97] and the references cited there in) deals, among others, extensively with *maintainability*, defined as the ease with which a system can be kept in operation when modifications to the code become necessary [Pig97, p.274]. The factors affecting maintenance can be divided into three categories: the state the system is in (how reliable, understandable, testable, modular, and extensible is it?), the nature of the changes required (is the design prepared for the anticipated changes), and the skills of and process used by the maintenance team (procedures for recording modification and enhancement requests, use of steps that have future maintainability as an objective, etc).

Some of these maintainability factors (e.g., quality of the configuration management) are not affected by the use of a domain-specific language. Others are negatively influenced: The number of different languages used in the system [Pig97, p.283] increases, which in itself makes maintenance more difficult. Also, it may be difficult to find personnel fluent in this particular DSL (proper documentation of the use, design, and implementation of the DSL, will reduce this risk by making the DSL easy to learn).

The use of an explicit software maintenance model (who is performing which steps in what order) by the maintainers is considered an important factor for improving maintainability [Pig97, p.40]. Adopting a DSL affects only the actual steps taken: domain experts without much programming or maintenance experience now can inspect the consequences and quality of the modifications made by the maintainers (for the RISLA case this is particularly important: the interest rate product implementations should correspond to reality).

Positively impacted by the use of a DSL are the source code maintainability attributes (modularity, encapsulation, cohesion, portability, understandability, etc), which many regard as the predominant maintainability factors [Pig97, OH94, p.289]. The most important properties are that the DSDs are much smaller than their general purpose language counterparts, and that the DSDs are more descriptive, avoiding the need for many comment lines in the DSDs, and thus reducing the chance of obsolete comment lines.

Finally, increased maintainability will affect the principal maintenance cost indicator, the annual change traffic (ACT) — the fraction of code changed due

to maintenance each year. The maintenance effort ME is related to ACT and the initial development cost in man months DM as follows [Boe81]:

$$ME = F * DM * ACT$$

where F is a multiplication factor representing the system maintainability. When using a DSL, one should split this into the development costs and ACT of both the DSL compiler (the DSPs) and the actual set of DSL programs used (the DSDs):

$$ME = F_{DSD} * DM_{DSD} * ACT_{DSD} + F_{DSP} * DM_{DSP} * ACT_{DSP}$$

Typically, ACT_{DSP} should be close to zero (the compiler should be stable). The costs of a high change rate for the DSL programs (ACT_{DSD}) is only related to the development costs of the DSDs, not to the development costs of the compiler.

When deciding whether to use a DSL, estimates of the costs related to ACT in a traditional language and in a DSL setting will play an important role.

3.2 Benefits of DSLs

The single most important benefit of using domain-specific languages is that the domain-specific knowledge is formalized at the right level of abstraction. This, in turn, has the advantages that:

- Domain experts themselves can understand, validate, and modify the software by adapting the domain-specific descriptions (DSDs).
- Modifications are easier to make and their impact is easier to understand.
- Domain-specific knowledge is explicitly available, and not hidden into, e.g., COBOL code (the use of a DSL avoids the need for *business rule extraction*).
- The explicitly available knowledge can be re-used across different applications.
- The way the knowledge is represented is independent of the implementation platform; the DSPs hide whether the DSDs are translated into C, Fortran, COBOL, ...

Concerning the costs of using DSLs, there is empirical evidence suggesting that the use of DSLs increases flexibility, productivity, reliability, and usability [KMB+96]. As a way of reducing the costs of the initial development of the DSL and DSPs, the language and its tools can be sold as a product to competitors in the same field. In this way, it is possible to earn back initial development costs but at the same time keeping secret the suite of DSDs (DSL programs) that describe the company's proprietary products.

3.3 DSL Development

The development of a DSL requires a thorough understanding of the underlying domain. The steps to be taken include:

- Identify problem domain of interest.
- Gather all relevant knowledge in this domain.
- Cluster this knowledge in a handful of semantic notions and operations on them.
- Construct a library that implements the semantic notions.
- Design a DSL that concisely describes applications in the domain.
- Design and implement a compiler that translates DSL programs to a sequence of library calls.
- Write DSL programs for all desired applications and compile them.

3.4 DSL Design Questions

With respect to software maintenance, there are a number of considerations to be taken into account during the design of a DSL:

- Who is going to write the DSDs? What is the expected domain-specific background, and how much programming knowledge is required?
- How many DSDs will there be needed, and how long are they going to be? It may be possible to validate the correctness of three pages of DSL code, but who is going to predict the impact of a change in one out of 100 DSDs, each 25 pages long?
- Which (decidable) forms of static analysis and which integrity checks on DSDs are anticipated?
- What should happen if it turns out that the language requires new data types or new functionality?

  One approach could be to give the DSL sufficient expressive power to define new data types or data operations, but this complicates the construction of the DSPs. For example, some form of iteration or recursion increases expressive power, but making the language Turing complete will make the verification of important properties (termination) undecidable.
- Does the DSL support user-definable syntax for, e.g., naming procedures? This may increase the readability, an important issue in DSLs, but it seriously complicates the construction of DSPs, including analysis tools that are needed during later maintenance phases.

- Is the main library written in the DSL or written in the target language? Who will be responsible for maintaining the library?
- Is the interface (data representation) to other systems easily adaptable or is it hidden inside the implementation of the DSL compiler?
- Who is going to maintain the DSPs? Is the knowledge about the domain sufficiently stable such that changes in the design of the DSL or the DSP are not to be expected?

The actual trade-off to be made for each of these issues clearly depends on the domain and the application at hand, and on the prominence maintenance considerations take during the DSL design.

3.5 Risks

The maintenance risks involved in the use of DSLs can partly be related to making the wrong trade-offs in the design questions listed above. Other issues include:

- The use of a DSL involves a shift from maintaining hand-built applications towards maintaining (a) DSDs (DSL programs defining each application); (b) DSPs (the DSL compiler); (c) a DSL library of predefined objects. Especially maintaining the DSL compiler requires skills not available in every organization.
- For existing, widely used, languages one can profit from readily available manuals, tutorials, courses, and experienced people. For a new DSL one has to develop this all from scratch.
- For related, but different, application areas different DSLs are needed. How can applications based on them cooperate?

In the remaining sections, we will discuss two techniques to alleviate two of these risks.

4 Designing and Implementing DSLs

As mentioned above, the use of a DSL has important benefits, but moves part of the maintenance problems to the DSP level. In this section, we discuss the ASF+SDF Meta-Environment, and how it supports the development and maintenance of application languages. It was in fact used during the design of RISLA and RISQUEST, the languages described in Section 2.

It is the aim of ASF+SDF to assist during the design and further development of (domain-specific) languages [BHK89, Kli93, DHK96]. It consists of a formalism to describe languages and of a Meta-Environment to derive tools from

such language descriptions. Ingredients often found in an ASF+SDF language definition include the description of the (1) context-free grammar, (2) context-sensitive requirements, (3) transformations or optimizations that are possible, (4) operational semantics expressing how to execute a program, and (5) translation to the desired target language. The Meta-Environment turns these into a parser, type checker, optimizer, interpreter, and compiler, respectively.

4.1 The ASF+SDF Formalism

The language ASF+SDF grew out of the integration of the Algebraic Specification Formalism ASF and the Syntax Definition Formalism SDF [BHK89]. An ASF+SDF specification consists of a declaration of the functions that can be used to build *terms*, and of a set of equations expressing *equalities* between terms.

If we use ASF+SDF to define a language L, the grammar is described by a series of functions for constructing abstract syntax trees. Transformations, type checking, translations to a target language L', etc., are all described as functions mapping L to, respectively, L, Boolean values, and L'. These functions are specified using conditional equations, which may have negative premises. In addition to that, ASF+SDF supports so-called *default*-equations, which can be used to "cover all remaining cases", a feature which can result in significantly shorter specifications for real-life situations [DHK96]. Specification in the large is supported by some basic modularization constructs.

Terms can be written in arbitrary user-defined syntax. In fact, an ASF+SDF signature is at the same time a context-free grammar, and defines a fixed mapping between sentences over the grammar and terms over the signature. Thus, an ASF+SDF definition of a set of language constructors specifies the concrete as well as the abstract syntax at the same time. Moreover, concrete syntax can be used in equations when specifying language properties. This smooth integration of concrete syntax with equations is one of the factors that makes ASF+SDF attractive for language definition.

4.2 The ASF+SDF Meta-Environment

The role of the ASF+SDF Meta-Environment [Kli93] is to support the development of language definitions, and to produce prototype tools from these. It is best explained using Figure 3. A modular definition of language L, generates *parsers*, which can map L-programs to L-terms, *rewriters*, which compute functions over L-programs by reducing terms to their normal form, and *pretty printers*, which map the result to a textual representation. In the Meta-Environment, the generators are invisible, and run automatically when needed. The derived pretty printer can be fine-tuned, allowing one to specify compilers to languages in which layout is semantically relevant (e.g., COBOL) [BV96].

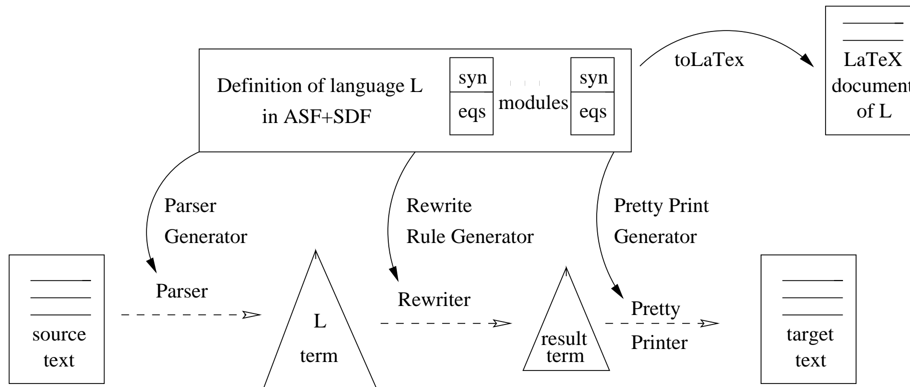

Figure 3: A language definition for L in the ASF+SDF Meta-Environment.

This pattern gives rise to a series of language processors, with a functionality as specified in the language definition. Basic user-interface primitives can be used to connect the processors to an integrated L-specific environment.

The ToLaTeX facility of the ASF+SDF Meta-Environment encourages the language designer to write his or her definition as a *literate* specification.

4.3 Industrial Applications

The typical industrial usage of ASF+SDF is to build tools for the analysis and transformation of programs in existing languages as well as for the design and prototyping of domain-specific languages. In this paper we will concentrate on the latter. The ASF+SDF formalism is used to write a formal language definition, and the Meta-Environment is used to obtain prototype tools. Once the language design is stable and completed successfully, the prototype tools can — depending on the needs of the application — be re-implemented in an efficient language like C, although there are also examples in which the generated prototype is satisfactory, and re-implementation is not even considered.

The underlying observation is that language design is both critical and difficult, and that it should not be disturbed by implementation efforts in a language like C. At the same time, prototype tools are required during the design phase to get feedback from language users. ASF+SDF helps to obtain these tools with minimal effort, by executing the language definitions, and by offering a number of generation facilities.

This requires an extra investment during the design phase, since ASF+SDF enforces users to write a thorough language definition. The assumption is that this investment will pay for itself during the implementation phase, an assumption confirmed by the various projects carried out so far, such as the ones discussed in [BDK+96].

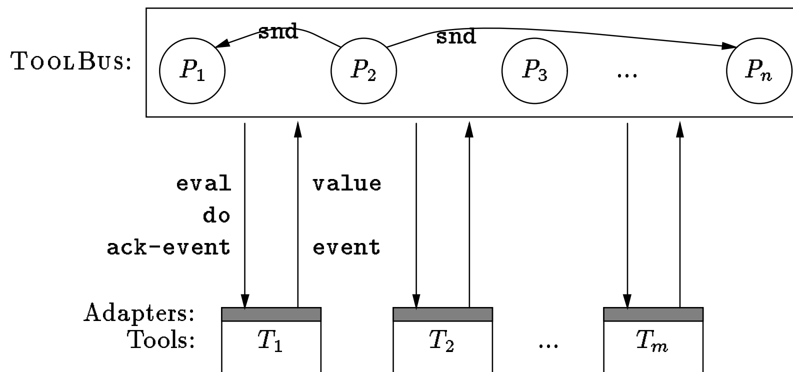

Figure 4: Global organization of the ToolBus

5 Coordinating different DSLs

So far we have seen how language technology can be applied to design and prototype a specific DSL and how to build supporting tools for DSL programs. In general, however, one will need a whole range of DSLs to cover the application areas that occur in a large organization. How can applications that have been built by means of different DSLs be coordinated? We answer this question in two steps: first we introduce the ToolBus coordination architecture and then we show how it solves the coordination issue just raised.

5.1 The ToolBus coordination architecture

In [BK96b, BK96a] the ToolBus coordination architecture has been proposed that facilitates the interoperability of heterogeneous, distributed, software components. To get control over the possible interactions between components ("tools") direct inter-tool communication is forbidden. Instead, all interactions are controlled by a "**T** script" that formalizes all the desired interactions among tools. This leads to a communication architecture resembling a hardware communication bus.

The global architecture of the ToolBus is shown in Figure 4. The ToolBus serves the purpose of defining the cooperation of a variable number of *tools* T_i ($i = 1, ..., m$) that are to be combined into a complete system. The internal behavior or implementation of each tool is irrelevant: they may be implemented in different programming languages, be generated from specifications, etc. Tools may, or may not, maintain their own internal state. Here we concentrate on the external behavior of each tool. In general an *adapter* will be needed for each tool to adapt it to the common data representation and message protocols imposed by the ToolBus.

The ToolBus itself consists of a variable number of processes P_i ($i =$

$1, ..., n$). The parallel composition of the processes P_i represents the intended behavior of the whole system. Tools are external, computational activities, most likely corresponding with operating system level processes. They come into existence either by an execution command issued by the ToolBus or their execution is initiated externally, in which case an explicit connect command has to be performed by the ToolBus. Although a one-to-one correspondence between tools and processes seems simple and desirable, this is not enforced and tools are permitted that are being controlled by more than one process as well as clusters of tools being controlled by a single process.

At the implementation level, the **T** script is executed by an interpreter that makes connections with tools via TCP/IP. In various case studies tools for user-interfacing, data storage and retrieval, parsing, compiling, constraint solving, scheduling, simulation and game-playing have been successfully integrated in various combinations yielding seamlessly integrated applications although the building blocks used are heterogeneous and may even execute in a distributed fashion.

5.2 Exchanging data

When coordinating distributed, heterogeneous, components, two key questions should be answered:

- How do components exchange data?
- How is the flow of control between components organized?

The former is discussed here, the latter is postponed to Section 5.3. There are two alternatives for exchanging data between components. One can either provide a direct mapping between the machine/language-specific representations of data in the various components or one can provide a common representation to which all machine/language-specific representations are converted.

In the case of the ToolBus the latter approach has been chosen and simple prefix terms are used as common data representation. Terms may consist of integers, strings, reals, function applications (e.g., `f(1,2)`) and lists (e.g., `[1, "abc", 3]`). For most applications this suffices, but as a general escape mechanism, terms may contain so-called binary strings that can represent arbitrary binary data such as, for instance, object files and bitmaps.

At the implementation level, terms are compressed before they are shipped between components, thus enabling fast exchange of large amounts of data.

5.3 T scripts

A **T** script describes the overall behavior of a system and consists of a number of definitions for processes and tools and one ToolBus configuration describing the initial configuration of the system. A process is defined by a process

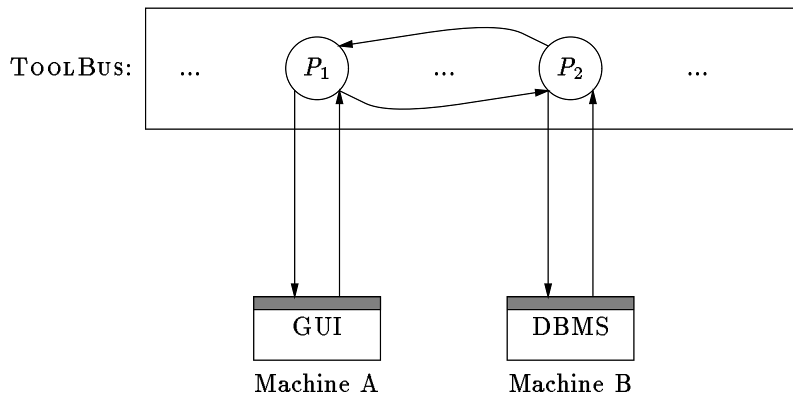

Figure 5: A Typical distributed application.

expression, and a tool by the name of its executable. Process behavior is based on a variant of Discrete Time Process Algebra and provides primitives for

- synchronous, binary, communication ("messages");
- asynchronous, broadcasting communication ("notes");
- tool-related actions such as creation/connection, communication, and termination/disconnection;
- process composition operators such as sequential composition, choice, iteration, parallel composition, and conditional;
- remote monitoring of processes and tools;
- delay and timeout.

5.4 Examples

A typical application of the ToolBus approach is shown in Figure 5. From the user's perspective, a database management system (DBMS) can be queried through a graphical user-interface (GUI). From an architectural perspective, the GUI and the DBMS are completely decoupled and they are even running on different machines. The key issue here is that there is no fixed connection between the components; both only communicate with the ToolBus and the processes running there (e.g., P_1 and P_2) determine the routing of GUI requests to the DBMS. This is achieved using the various communication primitives available in **T** scripts. The routing may even be changed dynamically, without disturbing the overall operation of the application.

Other examples are a distributed auction (where one auction master and a variable number of bidders cooperate in an auction, each working via his/her

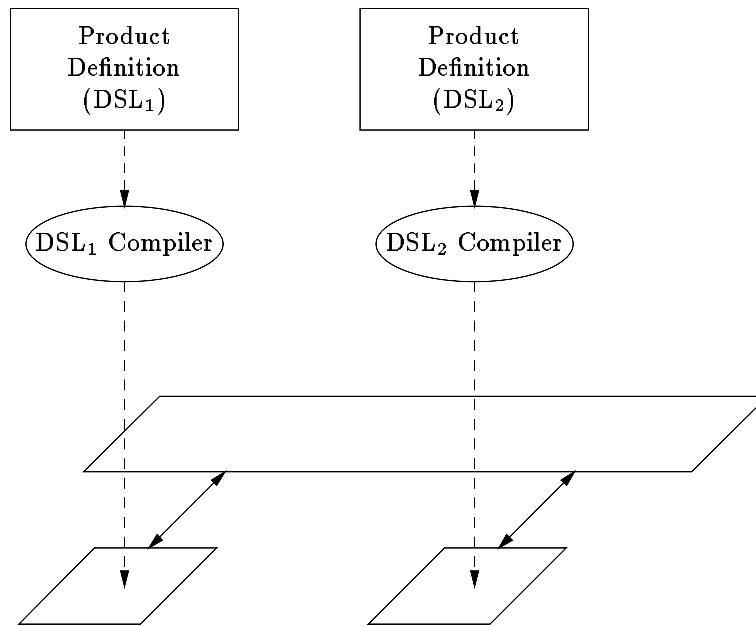

Figure 6: Coordination of DSLs using the ToolBus.

own workstation), distributed multi-user games, multi-user distributed programming environments and the like.

In all these examples, the **T** script defines the global architecture of each application and a wide variety of components based on a range of implementation technologies can be fitted into this architecture provided that they obey the protocol imposed by the **T** script.

5.5 Coordinating DSLs with the ToolBus

Applications that have been constructed by means of different DSLs can be coordinated using the ToolBus technology as well. Recall from Figure 1 the case of a product definition in some DSL and its compilation to the desired IT support for that product. Next, we sketch in Figure 6 the case where two different products are being defined using two different DSLs and how they can be coordinated. Typically, all DSL compilers will generate ToolBus compatible components and an overall script will describe the cooperation of all (generated) components.

There are several issues involved here related to maintenance, renovation, and gradual evolution:

- The ToolBus acts as a form of “middleware” that can connect new and old software components. It enables the gradual transition from a system based on traditional, hand-crafted, components to a system based on generated components using DSLs.
- Maintenance of a specific DSL or its compiler does not affect the whole system.
- Different DSLs can use different technology (when relevant). This enables transitions to new technology during the evolution of a system.
- For flexibility and ease of maintenance, each DSL compiler can also be based on a private ToolBus (not shown in Figure 6).

6 Concluding remarks

6.1 DSL is not a panacea

DSLs are no panacea for solving all software engineering problems, but a DSL designed for a well-chosen domain and implemented with adequate tools may drastically reduce the costs for building new applications as well as for maintaining existing ones.

On the positive side, in a DSL-based approach one concentrates all knowledge about an application in the DSL and its supporting component libraries, while all implementation knowledge is concentrated in the DSP (DSL compiler). From the perspectives of flexibility, quality assurance, maintenance, and knowledge management this is a highly desirable situation.

On the negative side, an application domain may not yet be sufficiently understood to warrant the design of a DSL for it or adequate technology may not be available to support the design and implementation of the DSL. Under such circumstances a more traditional approach to system design and maintenance should be preferred.

6.2 Future directions

We have already mentioned that the usability of DSLs by application domain experts (as opposed to programmers) is a decisive factor for their acceptance and success. There are several directions for increasing the ease of use of DSLs:

- Visual DSLs in which visual/iconic user-interfaces are used to compose library components.
- Natural language DSLs in which stylized natural language sentences are used to compose applications.

- Interactive DSLs in which domain experts are guided through a list of queries in order to select and assemble an application from library components.

- Prototyping environments for DSLs that support the realistic simulation of applications.

Regarding the design and implementation of DSLs we see the following needs:

- Further development of tools for designing and implementing DSLs. Typical issues: (a) modular structure of the DSL; (b) static checking of DSDs (DSL programs); (c) correctness of the translation rules used by the DSP.

- Tools for designing and implementing supporting component libraries. Typical issues: (a) modular structure and design of the component library; (b) implementation of the modular structure in given implementation languages, e.g., how to implement parameterized modules in COBOL? There is a relation here with current work on designing so-called *business objects*.

- Tools for connecting different DSLs. Typical issue: while coordination architectures as described in Section 5 provide basic connectivity and interoperability, a more abstract, application level, model of coordination is needed.

- Collection of empirical data concerning maintenance costs (ACT, cost of maintenance per line of code, cost per enhancement request, ...) in systems built using domain-specific languages.

Domain specific languages ("little languages") introduce an appropriate abstraction level for packaging domain-specific knowledge and technology. "Little maintenance" is becoming feasible for applications using them, provided that state-of-the-art techniques are used like the ones discussed in this paper.

References

[ADR95] B. R. T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[Ben86] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[BK96a] J. A. Bergstra and P. Klint. The Discrete Time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 288–305. Springer-Verlag, 1996.

[BK96b] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION '96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[BDK+96] 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 M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[BV96] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[Cog95] Ph. Coggan. *The Money Machine: How the City Works*. Pinguin, 1995. Third edition.

[Deu94] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[EG92] Th. Eggenschwiler and E. Gamma. ET++ SwapsManager: Using object technology in the financial engineering domain. In *OOPSLA'92 Seventh Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–177. ACM, 1992. SIGPLAN Notices 27(10).

[GB96] S. J. Glover and K. H. Bennet. An agent-based approach to rapid software evolution based on a domain model. In *Proceedings International Conference on Software Maintenance ICSM'96*, pages 228–237. IEEE Computer Society Press, 1996. Monterey, CA.

[HB88] R. M. Herndon and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, SE-14:803–809, 1988.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KMB+96] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering ICSE-18*, pages 542–553. IEEE, 1996.

[OH94] P. Oman and J. Hagemeister. Constructing and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, 1994.

[Pig97] T. M. Pigoski. *Practical Software Maintenance – Best Practices for Managing Your Software Investment.* John Wiley and Sons, 1997.

[Wal96] L. Walton. Domain-specific design languages, 1996. URL http://-www.cse.ogi.edu/~walton/dsdls.html.