Reliable software implementation using domain-specific languages

Diomidis Spinellis
Department of Information and Communication Systems, University of the Aegean, Greece

ABSTRACT: We describe the use of domain-specific languages for expressing critical design values and constraints in a civil engineering CAD software application. Through the use of these specialised languages information that is critical to the correct operation of the software can be expressed in a form that can be easily drafted, verified, and maintained by domain experts (civil engineers) thus minimising the risk inherent from the mediation of software engineers. Although domain-specific languages can offer increased expressiveness, runtime efficiency, and reliability at a modest implementation cost, system architects should take into account the issues of training costs, tool support, and software process integration.

In G. I. Schuëller and P. Kafka, editors, *Proceedings ESREL '99 — The Tenth European Conference on Safety and Reliability*, pages 627–631, Munich-Garching, Germany, September 1999. ESRA, VDI, TUM, A. A. Balkema.

1 INTRODUCTION

The safety and reliability of products, processes, equipment, and installations is often critically affected by the software that controls their design and operation (Winter et al. 1998). Software engineering stands out among other engineering disciplines because of its tight, haphazard, and fluid coupling with other elements of its operating environment. Mechanical, civil, or electrical engineers can base their designs on standardised and well understood specifications and constraints, and can design their implementations based on materials and components of known modalities and properties. In contrast to that, software engineers have to rely on specifications often expressed in the notation most suited to the application domain, design an artefact whose application changes significantly the environment it was designed for (Lehman 1991), and rely on implementors whose output quality can vary up to a factor of 10 (Sackman et al. 1968). These problems are of particular importance for safety critical applications where human life can be put at risk from malfeasant software designs and implementations.

Our approach for overcoming those problems is a software process architecture based on *domain specific languages* (DSLs). These allow the specification of critical parts of a software subsystem in the most appropriate formalism for the application do-

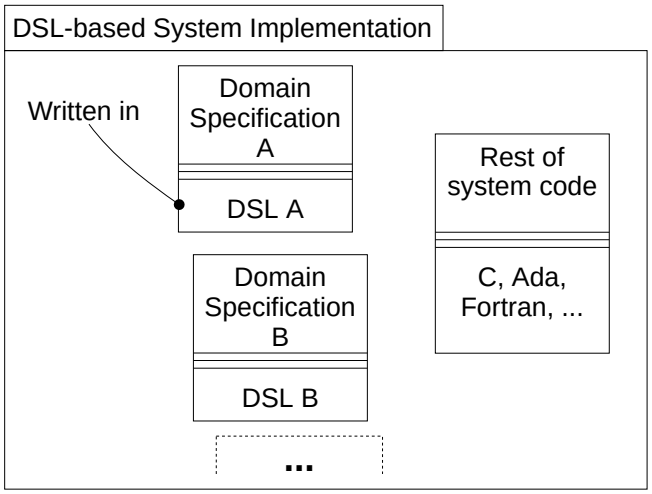

Figure 1: UML diagram of a DSL-based system architecture.

main, thus bridging the gap between the domain expert specifications and the software implementation, providing maintainability, and — once a particular DSL has been correctly implemented — ensuring consistent quality throughout the system's lifecycle. In the following paragraphs we illustrate this concept by detailing the usage of DSLs in the design and implementation of a civil engineering CAD application. The application is typical for the class described so far: it embodies a large amount of domain knowledge and its failures can lead to reliability and safety problems.

2 DOMAIN SPECIFIC LANGUAGES

A domain-specific language (DSL) (Ramming 1997) is a programming language tailored specifically for an application domain: rather than being general purpose it captures precisely the domain's semantics. Examples of DSLs include *lex* and *yacc* (Johnson and Lesk 1987) used for program lexical analysis and parsing, HTML (Berners-Lee and Connolly 1995) used for document mark-up, and VHDL used for electronic hardware descriptions. Domain-specific languages allow the concise description of an application's logic reducing the semantic distance between the problem and the program (Bell et al. 1994; Spinellis and Guruprasad 1997).

DSLs are, by definition, special purpose languages. Any system architecture encompassing one or more DSLs is typically structured as a confederation of modules; some implemented in one of the DSLs and the rest implemented using a general purpose programming language (Fig. 1). As a design choice for implementing safety-critical software systems DSLs present two distinct advantages over a "hard-coded" program logic:

Concrete Expression of Domain Knowledge
Domain-specific functionality is not coded into the system or stored in an arcane file format; it is captured in a concrete human-readable form. Programs expressed in the DSL can be scrutinised, split, combined, shared, published, put under release control, printed, commented, and even be automatically generated by other applications.

Direct Involvement of the Domain Expert The DSL expression style can often be designed so as to match the format typically used by the domain expert. This results in keeping the experts in a very tight software lifecycle loop where they can directly specify, implement, verify, and validate, without the need of coding intermediaries. Even if the DSL is not high-level enough to be used as a specification language by the domain expert, it may still be possible to involve the expert in code walkthroughts far more productive than those over code expressed in a general purpose language.

3 PLATFORM DESCRIPTION

FESPA *for Windows* (LH Software 1998) is an integrated software system used to analyse, dimension, display, verify, and draw three dimensional building structures (Fig. 2). A building is designed along the following steps:

1. specification of the building in terms of slabs, pillars, bars, and other nodes,
2. template-based production of additional floors,
3. calculation of pillar loads and creation and dimensioning of the building foundations, footings, and connecting rods,

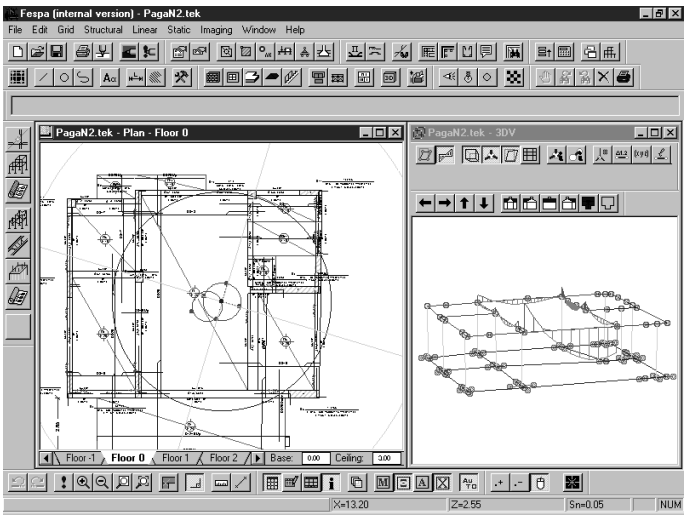

Figure 2: FESPA *for Windows* featuring a building overview and a three dimensional model displaying beam tension moments.

4. solution of the space structure and plotting of member distortions, forces, and moments,

5. calculation of section reinforcements, and

6. derivation and plotting of the wood mould.

All calculations described above are based on hundreds of parameters that determine inter alia the material and ground properties, the expected seismic environment, and the building's intended usage.

4 METHODOLOGY

Around 5% of the 210000 lines of code representing the user interface of FESPA are written in one of the ten DSLs designed for concretely expressing important elements of the system's specification.

This concept can be illustrated considering the specification of the system's entity properties. Every one of the 42 different entities (such as slabs, beams, pillars, etc.) has a set of properties associated with it. In total 1181 properties have to be specified. The type of each property, the permitted value range, and the set of allowable values form important parts of the specification, have little meaning for the software implementor, and are critical for the reliable and safe operation of the system; a situation which calls for the application of a DSL. A language was designed to

```
#define SL_STEEL 220:500:220;400;420;500

big_pressure:Rod stir-
rup steel:rod_steel_stirrup:-
            :regulation_new:SL_STEEL
```

Figure 3: DSL specification of building parameter properties.

Name	Type	Min	Max
Slab steel	pressure	220	500
Rod stirrup steel	pressure	220	500
Pillar stirrup min intersections	int	2	
Beams' min top bars	int	2	
Steel safety factor	nounit		
Friction angle β	angle	0	36
Brickwork fck	pressure	0	50000
Shear wall top γ Rd	len	1	1.4
Static beam load Z	load	-1000	1000

Figure 4: Domain-expert view of building parameter properties.

bind together the type of every parameter, the name presented to the user, the associated variable and function call-backs within the program, the value range, explanatory diagrams, and the selection values.

Thus, DSL expressions of the form shown in Figure 3 are presented to the domain expert (civil engineer) in tabular form as exemplified in Figure 4 and are compiled into efficient C++ code for incorporation into the CAD system by a small DSL compiler coded in Perl (Wall and Schwartz 1990). The presentation of values illustrated above allows the domain expert to directly specify and verify important aspects of the system's implementation without relying on intermediaries.

A similar methodology was followed for expressing other elements of the system's operation. Representative examples include the specification of the system's:

Commands The grouping, functionality, and explanatory details of the 450 available commands.

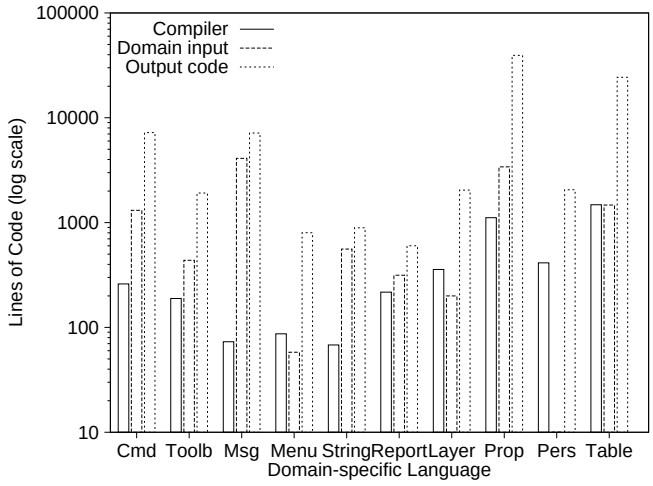

Figure 5: Size metrics of the system's DSL-based implementation

Toolbars The functionality, icons, and explanatory details of the system's toolbars.

Menus The names, associated commands, allowable execution context, and initialisation sequence for the system's menu structure.

Reports The specification of the report generator.

Layers The grouping, functionality, and interactions between the system's 68 layers of entities.

Persistency Data allowing the automatic persistent storage of the user property selections within the data files. The input source for this DSL is actually a suitably annotated version of the system's source code.

For every one of the above DSLs we implemented a specialised compiler which read its domain-specific input source and transformed it into efficient C++ code. The relationship between the lines of code needed to implement the compiler, the domain-specific data, and the resulting target code is illustrated in Figure 5.

5 APPRAISAL

The object of a DSL-based software architecture is to minimise the semantic distance between the system's specification and its implementation. Although the concept bears similarity to executable specification languages (Sommerville 1989, p. 125), (Turski and Maibaum 1987, p. 135) such as (Paryavi and Hankley 1995) the DSL approach exhibits some important advantages:

Expressiveness Executable specification languages taking a Swiss army knife approach towards the problem of specification offer facilities for specifying all types of systems, but often at a cost of clearness of expression. As an example OBSERV (Tyszberowicz and Yehudai 1992) provides a multiparadigm environment allowing the system specification using object-oriented constructs, finite state machines, and logic programming. In contrast, DSLs being tailored towards a narrow, specific domain can be designed to provide the exact formalisms suitable for that domain.

Runtime Efficiency The possible interactions between different elements of a general purpose specification language such as its type system and its support for concurrency result in runtime inefficiencies. A narrowly focused DSL can employ the most efficient implementation strategy and specialised optimisations for satisfying the expressed specification.

Modest Implementation Cost DSLs are typically implemented by a translator that transforms the DSL source code into source or intermediate code compatible with the rest of the system. All DSL translators we used in the FESPA *for Windows* implementation transform DSL source code into C++ source code. Such an approach can often be implemented using string processing languages such as *awk* (Aho et al. 1979) and Perl, language development tools such as *lex* and *yacc*, specialised systems such as TXL (Cordy et al. 1991) and KHEPERA (Faith et al. 1997), or declarative languages such as Prolog and ML. The DSL implementation cost is — and should always be — modest; in our case we implemented ten DSL translators in a total of 4000 lines of Perl.

Reliability As described in the previous paragraph, the limited scope of a DSL often allows a source-

to-source transformation type of implementation. The small scale of the required implementation effort often results in a translator whose correctness can be trivially verified. The size of typical executable specification languages means that the implementor must often take the correctness of the language's implementation on trust.

On the other hand, the system architect contemplating the use of a DSL architecture should also have in mind the following potential shortcomings of this approach:

Tool Support Limitations CASE and integrated software development tools offer only limited support for integrating DSLs into the development process. Ad hoc solutions are often required to smoothly integrate DSL code with existing revision control systems, compilers, editors, source browsers, and debuggers.

Training Costs In contrast to established specification languages such as Z (Potter et al. 1991) system implementors and maintainers will by definition have no prior exposure to the DSL being used. This problem is somehow mitigated by the fact that a correctly chosen DSL will be familiar to other participants of the implementation effort such as those involved in the specification, beta testing, and final use. These participants will be able to perform DSL code walkthroughs — a task normally reserved for experienced software engineers.

Design Experience DSL-based system architectures are not widely adopted within the software industry. As a result, there is an evident lack of design experience, prescriptive guidelines, mentors, design patterns, and supporting scientific literature. Early adopters will need to rely more on their own judgement as they adopt the approach in a stepwise fashion.

Software Process Integration The use of DSLs is not yet an integral part of established software processes. Therefore, the software process being used has to be modified in order to take into account the design, implementation, integration, debugging, and maintenance of the adopted DSLs.

6 CONCLUSIONS

DSLs offer an additional level of abstraction in the implementation of software-based systems. In the example we described, 11000 lines of DSL specifications written, reviewed, and maintained by domain experts are automatically translated into 87000 lines of C++ code. A DSL-based architecture can be utilised in all areas where there is a formal standardised specification particular to the application domain that can be relatively easily translated into a standard programming language. Where this part of the specification is large or fluid, significant increases in software reliability, maintainability, and cost can be achieved through the use of the proposed DSL approach.

REFERENCES

Aho, A. V., B. W. Kernighan, and P. J. Weinberger (1979). Awk — a pattern scanning and processing language. *Software: Practice & Experience 9*(4), 267–280.

Bell, J., F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou (1994). Software design for reliability and reuse: a proof-of-concept demonstration. In *Conference on TRI-Ada '94*, pp. 396–404. ACM: ACM Press.

Berners-Lee, T. and D. Connolly (1995, November). RFC 1866: Hypertext Markup Language — 2.0. Status: PROPOSED STANDARD.

Cordy, J. R., C. D. Halpern-Hamu, and E. Promislow (1991, January). TXL: A rapid prototyping system for programming language dialects. *Computer Languages 16*(1), 97–107.

Faith, R. E., L. S. Nyland, and J. F. Prins (1997, October). KHEPERA: A system for rapid implementation of domain specific languages. See Ramming (1997), pp. 243–255.

Johnson, S. C. and M. E. Lesk (1987, July-August). Language development tools. *Bell System Technical Journal 56*(6), 2155–2176.

Lehman, M. M. (1991, September). Software engineering, the software process and their support. *Software Engineering Journal 6*(5), 243–258.

LH Software (1998). *FESPA for Windows*. Athens, Greece: Kleidarithmos. In Greek.

Paryavi, M. N. and W. J. Hankley (1995). OOSPEC: an executable object-oriented specification language. In *ACM 23rd annual computer science conference. CSC '95*, pp. 169–177. ACM: ACM Press.

Potter, B., J. Sinclair, and D. Till (1991). *An Introduction to Formal Specification and Z*. Prentice-Hall.

Ramming, J. C. (Ed.) (1997, October). *USENIX Conference on Domain-Specific Languages*, Santa Monica, CA, USA. USENIX.

Sackman, H., W. J. Erikson, and E. E. Grant (1968, January). Exploratory experimental studies comparing on-line and off-line programming performance. *Communications of the ACM 11*(1), 3–11.

Sommerville, I. (1989). *Software Engineering* (Third ed.). Addison-Wesley.

Spinellis, D. and V. Guruprasad (1997, October). Lightweight languages as software engineering tools. See Ramming (1997), pp. 67–76.

Turski, W. M. and T. S. E. Maibaum (1987). *The Specification of Computer Programs*. Addison-Wesley.

Tyszberowicz, S. and A. Yehudai (1992, July). OBSERV — a prototyping language and environment. *ACM Transactions on Software Engineering and Methodology 1*(3), 269–309.

Wall, L. and R. L. Schwartz (1990). *Programming Perl*. Sebastopol, CA, USA: O'Reilly and Associates.

Winter, V. L., J. M. Covan, L. J. Dalton, L. Alkalai, A. T. Tai, R. Harper, B. Flahive, W.-T. Tsai, R. Mojdehbakhsh, S. Rayadurgam, K. Mori, and M. R. Lowry (1998, April). Key applications for high-assurance systems. *Computer 31*(4), 35–45.