On the Use of *Declarative Meta Programming* for Managing Architectural Software Evolution

Position Statement for the 2nd Workshop on Object-Oriented Architectural Evolution
ECOOP 2000, Sophia Antipolis, France, June 13, 2000

Tom Mens, Kim Mens, Roel Wuyts
{tommens | kimmens | rwuyts}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium

Abstract. *When looking at existing tools that provide support for architectural software evolution, we can distinguish between support for run-time, pre-execution time and design-time evolution; between support for unconstrained and constrained evolution; and between proactive, reactive and retroactive support for evolution. Current tools that support architectural evolution can only deal with a subset of the above issues. Moreover, they typically address only some particular aspects of software evolution (e.g., impact analysis, reverse engineering, etc). We propose* declarative meta programming *as an expressive and uniform medium for building tools that support architectural evolution. Amongst others, this enables the integration and combination of such tools, and allows the construction of new tools by sharing and reusing earlier codified knowledge.*

1. The Architectural Decision Space

Current tools for architectural evolution are restricted in the *way* they provide support for evolution, as well as in the *kind* of evolution they provide support for. The limitations of each tool depend on a number of decisions that have been made during development of the tool. To clarify this, we have set up a decision space for architectural evolution that allows us to express three orthogonal decisions: **evolution-time**, **action-time** and **kind of evolution**.

The first decision, proposed by [12], concerns the time of evolution. This can be *design time*, when the architecture is still under development; *pre-execution time*, when the architecture has already been specified and the software has been implemented, but the software system is not yet running; or *run time* where the architecture can be modified dynamically while the software is running. Tool support for design-time evolution is considerably simpler than for run-time or pre-execution-time evolution, because at design time no existing (or even running) implementation needs to be taken into account.

A second decision concerns the different ways in which a tool can undertake actions to support evolution. *Proactive* tools try to identify the potential effect of making a change even before the change has actually been made. They might also prohibit certain changes that would otherwise lead to inconsistencies or other problems. *Reactive* tools start to work at the moment the actual changes are being made, and typically try to resolve potential evolution problems interactively. Finally, *retroactive* tools only detect problems after the fact [10].

The third decision has to do with the restrictions that are imposed on the evolution itself. With *unconstrained evolution*, the software architecture is allowed to evolve in any conceivable way, without any restriction whatsoever. *Constrained evolution*, on the contrary, imposes a number of constraints or recommendations on the possible ways in which the architecture can evolve:

Restrictions are negative constraints, in the sense that they prohibit certain kinds of evolutions that would breach these constraints. Examples are well-formedness constraints and integrity constraints that must be preserved during evolution.

Obligations are positive constraints that evolutions must satisfy in order to be valid. Examples are style conventions, provided that a tool is able to verify or enforce these conventions.

Recommendations can be seen as a kind of safety precautions that should preferably be followed. Experienced software architects, however, may decide to ignore them if needed (at their own responsibility). Like constraints, recommendations can be negative or positive. Examples of positive recommendations are *cookbooks* that document how an object-oriented framework should be customized [7]. Cookbooks only give *guidelines* of what to do, but do not *impose* these guidelines on the software architect.

Clearly, constrained evolution demands more reasoning capabilities from supporting tools than unconstrained evolution. For example, a means should be provided to ensure that imposed constraints are satisfied, and to help a software architect to follow suggested recommendations.

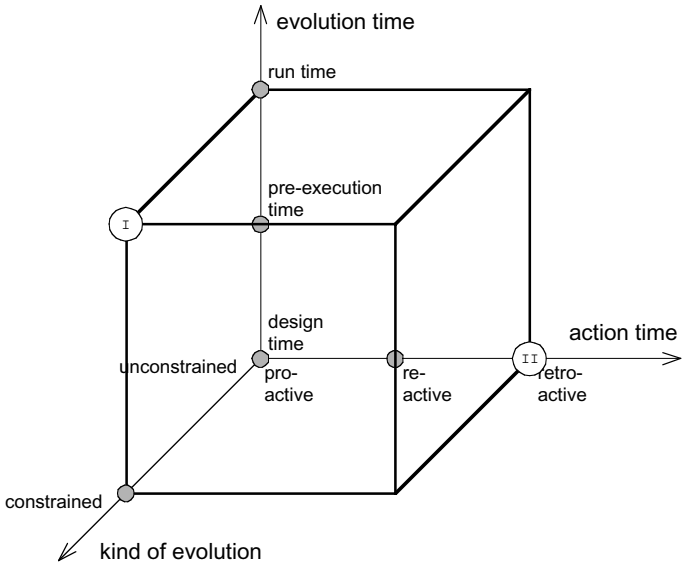

Figure 1: The Architectural Decision Space.

The three-dimensional grid of Figure 1 schematically illustrates the different decisions that tools for architectural evolution need to make. All existing tools can be situated somewhere in this architectural decision space. Let us take a look at two typical examples that take completely opposite positions in the decision space depicted in Figure 1.

(I) Tools for *run-time* architectural evolution (position `I` in Figure 1) are typically *constrained* and *proactive*, because one cannot afford to introduce inconsistencies when the architecture is running. The imposed constraints are *restrictions* on the evolution because only those changes are allowed that do not compromise application integrity.

(II) A tool like reuse contracts for detecting architectural evolution conflicts [17] is situated at the opposite side of the decision space (position `II` in Figure 1). It deals with architectural evolution at *design time*, and is well suited to deal with *unconstrained evolution*. Moreover, it is currently used *retroactively*.

The position that is taken in the decision space can have a large impact on the kind of tools that can be conceived. To illustrate this, Section 2 will discuss some evolution problems that need to be addressed, and will position different tools that solve these problems in the architectural design space. Although every tool focuses on a particular problem and may cover only a certain range in this design space, this does not imply it is not useful to have a uniform medium in which to build such tools. In Section 3, we propose *declarative meta programming* as such a medium.

2. Required Support for Architectural Evolution

Below we present a number of tools that address a variety of problems related to architectural evolutions. For some of these tools, we explain where they are situated in the decision space.

In Section 3 we will explain why they can easily be constructed using declarative meta programming.

- *Architectural drift* or *architectural erosion* arises when changes that break the constraints of the original architecture, are made to the implementation. **Conformance checking tools** [11,18,9,10] try to verify whether an implementation (still) matches the architecture. When different architectural views on the same implementation are provided, we also need support for **maintaining consistency** between these different views. Tool support can vary according to the level of synchronization one wants to obtain. Retroactive tools only start to work once all modifications have been made, while reactive tools can start taking appropriate actions as soon as an architectural breach is detected. In some cases, the tools can even be proactive, by prohibiting certain changes to the implementation that would breach the architecture and by providing help to resolve such breaches.
- In practice, due to deadline pressure or for reasons of efficiency, it is difficult to make an implementation fully conform to the intended software architecture. To be able to deal with such situations, we need **support for architectural *deviations*** [5].
- Tools that check whether a given architecture conforms to certain **architectural styles or patterns** are also useful.
- Tools for **architectural extraction or reconstruction** try to (re)construct the architecture from a certain implementation. These tools can be very helpful in the case of legacy code (extraction), or in the case of architectural drift (reconstruction).
- Tools for **impact analysis** [1,2], **effort estimation** and **change propagation** [13,15] reason about the potential consequences of making a change. These tools are typically *proactive*, because they intend to calculate the effect of a change beforehand.
- Tools that **detect the need for architectural evolution** find out when, where and how the architecture should evolve or be restructured [8,16].
- We also need tools that **provide disciplined support for architectural evolution**, and try to **detect and resolve conflicts** during evolution [4,17].
- The need for **co-evolution** arises when the architecture, design and implementation can evolve independently, but need to be kept synchronized in some way. Dealing with co-evolution is far from trivial, especially if there is a many-to-many mapping between architecture and implementation, and if multiple architectural views are allowed. [6] proposes a formal framework for co-evolution. In [3], declarative meta-programming is proposed as a novel but promising way to deal with co-evolution.

3. Declarative Meta Programming

At our lab, research is being conducted on how the emerging technique of declarative meta programming (DMP), can be used to build state-of-the-art software development support tools. As a particular instance of this research, in this position paper, we promote the use of DMP as a medium which is particularly suited for building tools that support *architectural evolution*. DMP is an instance of hybrid language symbiosis, merging a declarative meta-level language with a standard object-oriented base language. DMP requires that the symbiosis between the logic meta language and the object-oriented base language is made explicit, allowing base-level programs to be expressed as terms, facts or rules in the meta level.

As summarized in [3], a number of experiments have been performed to use DMP to qualify implementation-level artifacts with enforceable design and architectural concerns [20,19,9]. It seems intuitively clear that design information, and in particular architectural concerns, are best codified declaratively as constraints or rules. Such rules and constraints can be used to enforce or check architectural constraints in the source code, to search or browse for certain architectural constructs in the source, or even as a process for code generation and transformation based on this architectural information. In particular, we think DMP is an ideal medium for building architectural evolution support tools.

One of the advantages of using DMP for building architectural evolution is that it can be used to provide tool support throughout the entire architectural design space discussed in Section 1.

- **Evolution time.** First of all, it is clear that DMP is independent of the evolution-time. Support for *design-time evolution* can straightforwardly be achieved by expressing and reasoning about architectural descriptions in the declarative language. To support *pre-execution-time* evolution, an explicit symbiosis between the declarative meta-language and the OO base language is needed. The technique has not yet been tried out to support *run-time* evolution, but, in principle, everything is in place to do so. In a base language such as Smalltalk, with a completely open development environment, it is possible to capture any implementation or architectural modification, and declaratively manipulate this information.
- **Action-time.** A *retroactive* tool can be defined, for example, in terms of a logic query that is evaluated after the evolution has occurred. *Proactive* and *reactive* tools could be defined in terms of constraints that are triggered when certain changes or made to the source code, or when certain preconditions, postconditions or invariants are invalidated.
- **Predictability.** Declarative programming has long been identified as very suited to meta programming and language processing in general. In particular, both constrained and unconstrained evolution can easily be expressed using DMP, for example, by declaring architectural descriptions and constraints in terms of logic facts, rules and constraints.

One of the advantages of using a uniform medium for building tools that support architectural evolution, is that it enables the integration of all these tools. In the remainder of this section, we provide evidence for the fact that declarative meta programming is effectively suited for building such tools.

At our lab, prototype tools have been built to support: conformance checking of source code to architectural descriptions [9,10]; conflict detection during architectural evolution [17]; generating the source code of an application by describing it at design level as a configuration of architectural components [14]; checking, browsing for and enforcing design patterns and styles in source code [20]; doing source to source transformation from implementations with a clean design and architecture to more efficient ones [19]; and so on. In addition, [3] discusses extensively how DMP can be used for supporting *co-evolution*.

A declarative approach is equally suited to express most other tools described in Section 2. First of all, as mentioned earlier and as illustrated by our experiments, DMP seems to provide the desired level of abstraction to codify architectural knowledge. In addition, the multi-way querying, unification and backtracking facilities of typical DMP environments enable the powerful reasoning required by tools for architectural evolution. Thirdly, because all tools are built using the same declarative medium, many rules can readily be reused, thus significantly increasing the time and effort needed for building new tools.

4. References

[1] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Press, 1996.

[2] S. A. Bohner and R. S. Arnold. *An Introduction to Software Change Impact Analysis*. In [1], pp.˚1-26.

[3] T. D'Hondt, K. De Volder, K. Mens and R. Wuyts. *Co-evolution of object-oriented design and implementation*. Int. Symposium on Software Architectures and Component Technology: The State of the Art in Research and Practice. Enschede, The Netherlands, January 2000.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[5] G. Cugola, E. Di Nitto, A. Fuggetta and C. Ghezzi. *A Framework for Formalizing Inconsistencies and Deviations in Human-Centered Systems.* 1996.

[6] T. Katayama: *A Theoretical Framework of Software Evolution* . Proc. Int. Workshop on Principles of Software Evolution, pp. 1-5. Kyoto, Japan, 1998.

[7] G. E. Krasner and S. T. Pope., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming, pp. 26-49, August/September, 1988.

[8] M. Lanza. *Combining Metrics and Graphs for Object Oriented Reverse Engineering.* Thesis, University of Bern, October 1999.

[9] K. Mens, R. Wuyts and T. D'Hondt. *Declaratively Codifying Software Architectures Using Virtual Software Classifications*. Proceedings of TOOLS 29 Conference on Technology of Object-Oriented Languages and Systems, pp. 33 45, IEEE Press, 1999.

[10] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming.* PhD Dissertation (under preparation), Vrije Universiteit Brussel, 2000.

[11] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution.* Ph.D. dissertation, University of Washington, 1996.

[12] Peyman Oreizy. *Issues in the Runtime Modification of Software Architectures*. Technical Report UCI-ICS-TR-96-35, University of California, 1996.

[13] Dewayne E. Perry and Gail E. Kaiser. *Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems*. Proc. ACM Computer Science Conf., pp. 292-299, ACM Press, February 1987.

[14] M. J. Presso. *Generic Component Architecture Using Meta-Level Protocol Descriptions.* Master's dissertation, Vrije Universiteit Brussel, 1999.

[15] V clav Rajlich. *A Methodology for Software Evolution*. Journal of Software Maintenance, Vol. 9, 1997, pp. 103-125.

[16] Tamar Richner, and St phane Ducasse. *Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information*. Proc. Int. Conf. Software Maintenance. IEEE Press, September 1999.

[17] Natalia Romero. *Managing Evolution of Software Architectures with Reuse Contracts*. EMOOSE Dissertation, Vrije Universiteit Brussel, 1999.

[18] R. W. Schwanke, V. A. Strack and T. Werthmann-Auzinger. *Industrial Software Architecture with Gestalt*. Proceedings of IWSSD-8, pp. 176-180, IEEE Press, 1996.

[19] T. Tourw and W. De Meuter. *Optimizing Object-Oriented Languages Through Architectural Transformations*. 8th International Conference on Compiler Construction, pp. 244-258, Springer-Verlag, 1999.

[20] R. Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. Proceedings of TOOLS USA'98, pp. 112-124, IEEE Press, 1998.