Software Evolution through Dynamic Adaptation of its OO Design

Walter Cazzola[1],*, Ahmed Ghoneim[2], and Gunter Saake[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg, Germany
{ghoneim|saake}@iti.cs.uni-magdeburg.de

Abstract. In this paper we present a proposal for safely evolving a software system against run-time changes. This proposal is based on a reflective architecture which provides objects with the ability of dynamically changing their behavior by using their design information. The meta-level system of the proposed architecture supervises the evolution of the software system to be adapted that runs as the base-level system of the reflective architecture. The meta-level system is composed of cooperating components; these components carry out the evolution against sudden and unexpected environmental changes on a reification of the design information (e.g., object models, scenarios and statecharts) of the system to be adapted. The evolution takes place in two steps: first a meta-object, called *evolutionary meta-object*, plans a possible evolution against the detected event then another meta-object, called *consistency checker meta-object* validates the feasibility of the proposed plan before really evolving the system. Meta-objects use the system design information to govern the evolution of the base-level system. Moreover, we show our architecture at work on a case study.

Keywords: Software Evolution, Reflection, Consistency Validation, Dynamic Reconfiguration, UML, XMI.

1 Introduction

In advanced object-oriented information systems related to engineering applications, classes and, therefore, their instances are subjected to frequent adaptations during their life cycle. This applies to the structure of class definitions and the behavior of their objects. In the last decade, several object-oriented systems were designed to address the problem of adapting object structure and behavior to meet new application requirements (see for example [14, 9]).

* Walter Cazzola's work has been partially supported by Italian MIUR (FIRB "Web-Minds" project N. RBNE01WEJT_005).

Nowadays a topical issue in the software engineering research area consists of producing software systems able to adapt themselves to environmental changes by adding new and/or modifying existing functionalities. *Computational reflection* [15, 4] provides one of the most used mechanisms for getting software adaptability.

A software system with a long life span, must be able to dynamically adapt itself to face unexpected changes in its environment avoiding a long out-of-service period for maintenance. The evolution of the design of a software system is determined by the evolution of the behavior of its components and of the interactions among them. Several design elements govern such aspects: *class/object diagrams statecharts* and *sequence diagrams*. Run-time system evolution also involves such aspects, therefore, design information should be used to concert run-time evolution as well. Design information provide the right mechanism to grant the consistency of the evolved system against the system requirements.

A reflective architecture represents the perfect structure that allows running systems to adapt themselves to unexpected external events, i.e., to consistently evolve. In [6] we described a reflective architecture for system evolution at run-time. In such a framework, the system running in the base-level is the one prone to be adapted, whereas software evolution is the nonfunctional feature realized by the meta-level system. Evolution takes place exploiting design information concerning the running systems.

To correctly evolve[1] the base-level system, the meta-level system must face many problems. The most important are: (1) to determine which events cause the need for evolving the base-level system (2) how to react on events and the related evolutionary actions (3) how to validate the consistency and the stability of the evolved system and eventually how to undo the evolution, (4) to determine which information allows system evolution and/or is involved in the evolution.

In [5], we introduced a pattern language modeling the general behavior of the meta-level components and their interactions during the evolutionary process.

The rest of the paper is organized as follows: section 2 provides a brief overview of the tools we have adopted in our work; section 3 describes our reflective architecture (the evolutionary mechanism) and the evolutionary engine related with the architecture and its rules; section 4 describes the application of our reflective approach to software evolution by an example. Finally, in sections 5 and 6 we survey some related work, draw our conclusions and present some future work.

[1] By the sentence *correctly evolve a system* we mean the fact that evolution takes place only when the system remains consistent and stable after evolution.

2 Background

2.1 Computational Reflection

Computational reflection or *reflection* for short is the ability of a system to watch its own computation and possibly change the way it performs. Observation and modification imply an "underlay" that will be observed and modified. Since the system reasons about itself, the "underlay" modifies itself, i.e. the system has a *self-representation* [15].

A *reflective architecture* logically models a system in two layers, called *base-level* and *meta-level*[2]. The base-level realizes the *functional* aspect of the system, whereas the meta-level realizes the *nonfunctional* aspect of the system. Functional and nonfunctional aspects discriminate among features, respectively, *essential or not* for committing with the given system requirements. Security, fault tolerance, and evolution are examples of nonfunctional requirements[3]. The meta-level is *causally connected* to the base-level, i.e., the meta-level has some data structures, generally called *reification*, representing every characteristic (structure, behavior, interaction, and so on) of the base-level. The base-level is continuously kept consistent with its reification, i.e., each action performed in the base-level is *reified* by the reification and vice versa each change performed by the meta-level on the base-level reification is *reflected* on the base-level. More about the reflective terminology can be learned from [15, 4].

Reflection is a technique that allows a system for maintaining information about itself (meta-information) and using this information to change (adapt) its behavior. This is realized through the casual connection between the base- (the monitored system) and the meta-level (the monitoring system).

2.2 Design Information

Our approach to evolution uses *design information* as a knowledge base for getting system evolution. Design information consists of data related to the design of the system we want to evolve. UML is the adopted formalism for representing design information.

The *unified modeling language* (UML) [3, 11] has been widely accepted as the standard object-oriented modeling language for modeling various aspects

[2] In the sequel, for simplicity, we refer to the "part of the system working in the base-level or in the meta-level" respectively as base-level and meta-level.

[3] The borderline between what is a functional feature and what is a nonfunctional feature is quite confused because it is tightly coupled to the problem requirements. For example, in a traffic control system the security aspect can be considered nonfunctional whereas security is a functional aspect of an auditing system.

of software systems. UML is an extensible language, it provides mechanisms (stereotypes, and constraints) that allow introducing new elements for specific domains if necessary, such as web applications, database applications, business modeling, software development processes, and so on. A *stereotype* provides an extensibility mechanism for creating new kinds of model elements derived from existing ones, whereas *constraints* can be used to refine the behavior of each model element.

The design information we consider are related to two categories: system structure and behavior. *Structural design information* is an explicit description of the structure of the base-level objects. This includes the number of attributes and their data type. *Behavioral design information* describes the computations and the communications carried out by the base-level objects. It includes objects behavior, collaboration between objects, and the state of the objects. Structure and behavior of the system are modeled by class diagrams, sequence diagrams and state diagrams.

3 Software Evolution through Reflection

The main goal of our approach consists of evolving a software system to face environmental and requirement changes and validate the consistency of such an evolution. This goal is achieved by:

- adopting a reflective architecture which dynamically drives the evolution of the software system through its design information when an event occurs; this has been made possible by moving design information from design- to run-time.
- using two sets of rules which respectively describes how evolution takes place and when the system is consistent; these rules are used by the decisional components of the reflective architecture but not by the system that must be evolved;
- replanning the design information of the system and reflecting the changes on the running system.

In the rest of this section we give a brief overview of the reflective architecture and its components, we show how these components work and the manipulation of the design information.

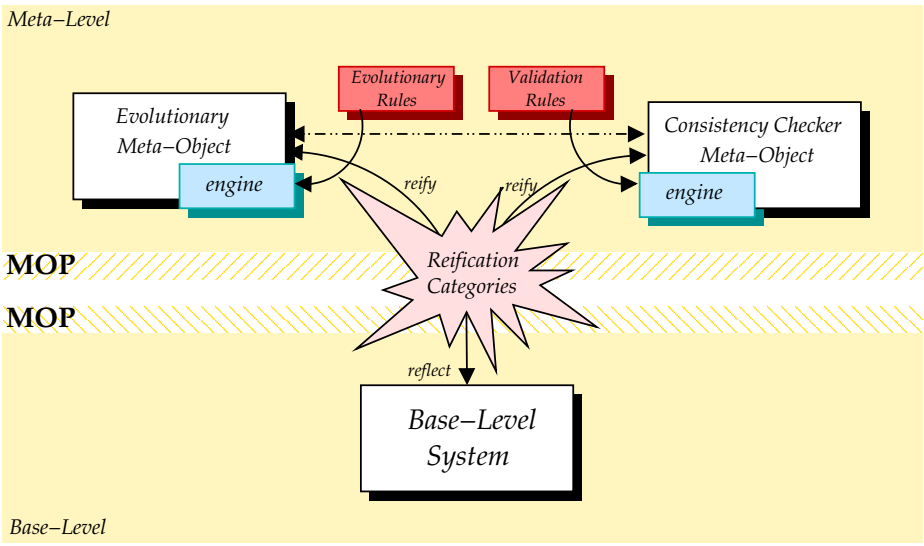

Fig. 1. Reflective architecture designed for the evolution of software systems.

3.1 The Reflective Architecture

To render a system self-adapting[4], we encapsulate it in a two-layers reflective architecture as shown in Fig. 1. The base-level is the system that we want to render self-adapting whereas the meta-level is a second software system which reifies the base-level design information and plans its evolution when particular events occur. Reflective properties as transparency [20, 21] and separation of concerns [13] provide the meta-level with the mechanism for carrying out the evolution of the base-level code and behavior without having previously foreseen such an adaptation for the system.

At the moment, we just take in consideration two kinds of software adaptation: *structural* and *behavioral* evolution. This limitation is due to the fact that, at the moment, we only reify the following design information related to the base-level:

– *object and class diagrams*, which describes classes, objects and their relations; this model represents the structural part of the system;
– *sequence diagrams*, which trace system operations between objects (inter-object connection) for each use case at a time; and

[4] By the sentence *to render a system self-adapting* we mean that such a system is able to change its behavior and structure in accordance with external events by itself.

– *statecharts*, which represent the evolution of the state of each object (intra-object connection) in the system.

The approach can be easily extended to observe and manipulate all the other diagrams provided by UML such as *use case*, *activity diagrams*.

The meta-level is responsible of dynamically adapting the base-level and it is composed of some special meta-objects, called *evolutionary meta-objects*. There are two types of evolutionary meta-objects: the *evolutionary* and the *consistency checker* meta-objects. Their goals consists of consistently evolving the base-level system. The former is directly responsible for planning the evolution of the base-level through adding, changing or removing objects, methods, and relations. The latter is directly responsible for checking the consistency of the planned evolution and of really carrying out the evolution through the causal connection relation.

The base-level system and its design information is reified in *reification categories* in the meta-level (see section 3.3 for more details). Classic reflection takes care of reifying the state and every other dynamic aspect of the base-level system, whereas the design information provides a reification of the design aspects of the base-level system such as its architecture and the collaborations among its components. The reification categories content is the main difference of our architecture with respect to standard reflective architectures. Usually, reifications represent the base-level system behavior and structure not its design information. Reification categories can be considered as representatives of the base-level system design information in the meta-level. Both evolutionary and consistency checker meta-objects directly work on such representatives and not on the real system, this allows a safe approach to evolution postponing every change after validation checks. As described in [5] when an external events occur, the evolutionary meta-object proposes an evolution as a reaction to the consistency checker meta-object which validates the proposal and schedules the adaptation of the base-level system if the proposal is accepted.

3.2 Decisional Engines and Evolutionary Rule Sets

Adaptation and validation are respectively driven by a set of rules which define how to adapt the system in accordance with the detected event and the meaning of system consistency.

To give more flexibility to the approach, these rules are not hardwired in the corresponding meta-object rather they are passed to a sub-component of the meta-objects themselves, respectively called *evolutionary* and *validation engines*, which interpret them. Therefore, each meta-object has two main components: (i) the core which interacts with the rest of the system (e.g., detect-

ing external events/adaptation proposals, or manipulating the reification categories/applying the adaptation on the base-level system) and implementing the meta-object's basic behavior, and (ii) the engine which interprets the rules driving the meta-object's decisions.

In this paper, for sake of simplicity, we express both evolutionary and validation rules by using the formalism for *event-condition-action (ECA)* [1, 8] rules. Rules are usually written in the following form:

on *event* **if** *conditions* **do** *actions*

where *event* represents the event which should ignite the evolution of the base-level system, *conditions*, and *actions*, respectively, represent the conditions the engine must validate when the *event* occurs and the actions the engine must carry out for adapting the system against the occurred *event*. Both *events* and *conditions* involve the base-level reification (see section 3.3 for details on reifications). The engines (both evolutionary and consistency checker) interpreting these rules are simply state machines indexed on events and conditions.

Both rules and engines working on them are tightly bound but completely unbound from the rest of the reflective architecture. Therefore, to adapt our approach to use rules specified with a different formalism is quite simple; we have just to substitute the engine with one able to interpret the chosen formalism. Of course, the engines must be able to interact with the rest of the architecture as described in the following algorithm. More complex and powerful approaches are under development.
In general, adaptation takes place as follows:

- the meta-level reifies the base-level design information and the system itself into reification categories;
- the evolutionary meta-object waits for an event that needs the adaptation of the base-level system; when such an event occurs it starts to plan evolution:
  - through the design information of the base-level system, it detects which base-level components might be involved in the evolution; then
  - it informs its engine about the occurred event and components involved in the evolution;
  - the evolutionary engine decides which evolutionary rule (or which group of evolutionary rules) is better to apply; then
  - it designs the evolutionary plan by applying the chosen evolutionary rule (or group of rules);
- the evolutionary meta-object passes the evolutionary plan to the consistency checker meta-object which must validate the proposed evolutionary plan before rendering the adaptation effective:

    - the consistency checker meta-object demands the validation phase to the validation engine;
    - the validation engine validates the proposed evolutionary plan by using its validation rules and the base-level system design information.
- if the proposed evolutionary plan is considered sound the consistency checker meta-object schedules the base-level system adaptation in accordance with such an evolutionary plan; otherwise the consistency checker meta-object returns an error message to the evolutionary meta-objects restarting the adaptation phase.

The evolutionary plan proposed by the evolutionary meta-object is a manipulation of the design information of the base-level system. The causal connection is responsible of modifying the model of the base-level system accordance with the proposed evolution. The adopted mechanism for transposing design information in the real system is based on the UML virtual machine [17].

The most important side-effect of this approach is represented by the fact that adaptation can take place also on nonstoppable systems because it does not require that the base-level system stops during adaptation, but it only needs to define when it is safe to carry out the adaptation.

3.3 Reification and Reflection by Using Design Information

We have talked about reifying and reflecting on design information of the base-level system whereas such design information simply feed the meta-level system during system bootstrap and drive its meta-computations during the evolution of the base-level system.

When an event occurs, the design information related to the base-level entities, that can be involved by the event, are used by the evolutionary and the consistency checker meta-objects for driving the evolution of such base-level entities (as described in the previous algorithm).

Design information identifies which entities are involved by the event (object/class and state diagrams), their behavior (sequence diagrams) and how the event can be propagated in the base-level system (collaboration diagrams). Therefore introspection and intercession on large systems become simpler than using standard reflective approaches because the design information provide a sort of index on the base-level entities and their interactions.

Moreover design information is the right complement to the base-level system reification built by the standard causal connection. Meta-objects consult and manipulate the design information in order to get information that otherwise are not easily accessible from the running system, e.g., the collaboration among objects. Design information is also used as a testbed for manipulation because they

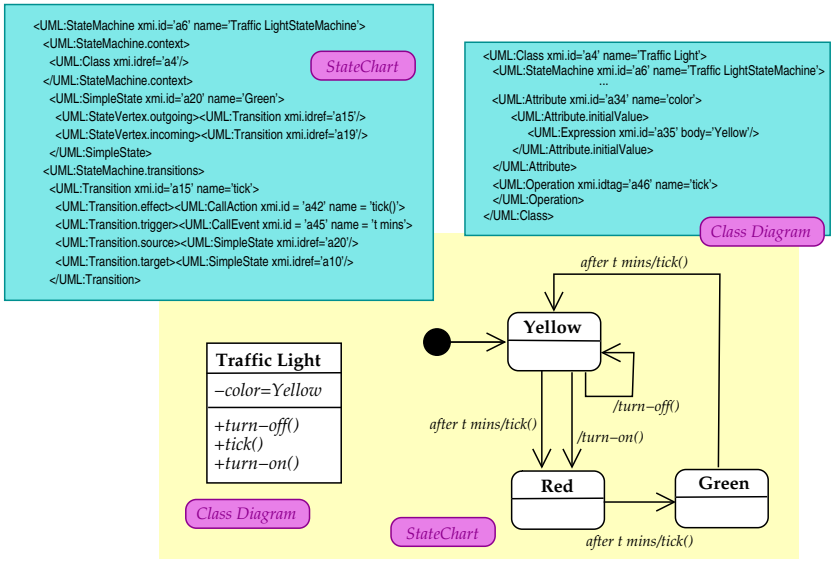

Fig. 2. Poseidon4UML's XMI translation of a simply class diagram and the related statechart

give a easily accessible overview of global features as inter-objects collaborations.

UML specifications provide graphical data that usually exist at design-time and are difficult to manage at run-time. Whereas, our meta-objects require such a specifications at run-time for driving the evolution. We chose to overcome this problem by encoding the design information in XML, in particular with the XMI standard [16]. XMI provides a translation of UML diagrams in a text-based form more suitable for run-time manipulation.

The XMI standard gives a guideline for translating each UML diagram in XML. Each diagram is assimilated to a graph whose nodes are the diagram's components (e.g., classes, states and so on), and arcs represents the relation among the components. The graph is decorated with XML tag describing the properties of the corresponding UML component. In our architecture, we use the XMI code generated by Poseidon4UML [2]. Poseidon4UML provides us with a tool for drawing UML diagrams and for generating the corresponding XMI code. Figure 2 shows a simple class diagram, a possible statechart for that and the corresponding XMI code (shortened for the sake of space) generated by Poseidon4UML.

At system bootstrap, the meta-level reifies the design information of the base-level, that is, the meta-level loads into the reification categories (each cate-

gories is devoted to an aspect of the base-level) the XMI representation of design information of the base-level. In this way we render accessible UML data-model to the meta-objects.

The XMI schemas are tightly linked with the base-level components. The evolutionary meta-object modifies these schemas introducing some specific XMI elements, providing the consistency checker with the necessary pieces of information for validating the evolutionary plan and for effectively modifying the base-level. Some of these elements are:

- `XMI.difference` is the element used to describe differences from the initial model;
- `XMI.delete` represents a deletion from the base model;
- `XMI.add` represents an addition to the base model; and
- `XMI.replace` represents a replacement of a model construct with another model construct in the initial model.

As described in [12], we can create new UML models from XMI schemas, therefore the evolutionary plan, which is a group of modified XMI schemas, can be reverted into UML diagrams. Basically, this reciprocity between UML diagrams and XMI schemas allows us maintaining the causal connection between base- and meta-level.

4 Urban Traffic Control System: a Case Study

When designing *urban traffic control systems* (UTCS), the software engineer will face many issues such as distribution, complexity of configuration, and reactivity to the environment evolution. Moreover, modern cities have to face a lot of unexpected hard to plan problems such as traffic lights disruptions, car crashes, traffic jams and so on. In [18] these issues and many others are illustrated.

The evolution of complex urban agglomerates have posed significant challenges to city planners in terms of optimizing traffic flows in a normally congested traffic network. Simulation and analysis of such systems require modeling the behavioral, structural and physical characteristics of the road system. This model includes mobile entities (e.g., cars, pedestrians, vehicular flow, and so on) and fixed entities (e.g., roads, railways, level crossing, traffic lights and so on).

Of course, the UTCS, due to its complexity, cannot be considered as a whole case study. In this section, we describe our approach to evolution involving just three components of the UTCS: *road*, *traffic light* and *traffic*. Figure 3 shows how this fragment of the UTCS is integrated with our reflective approach to

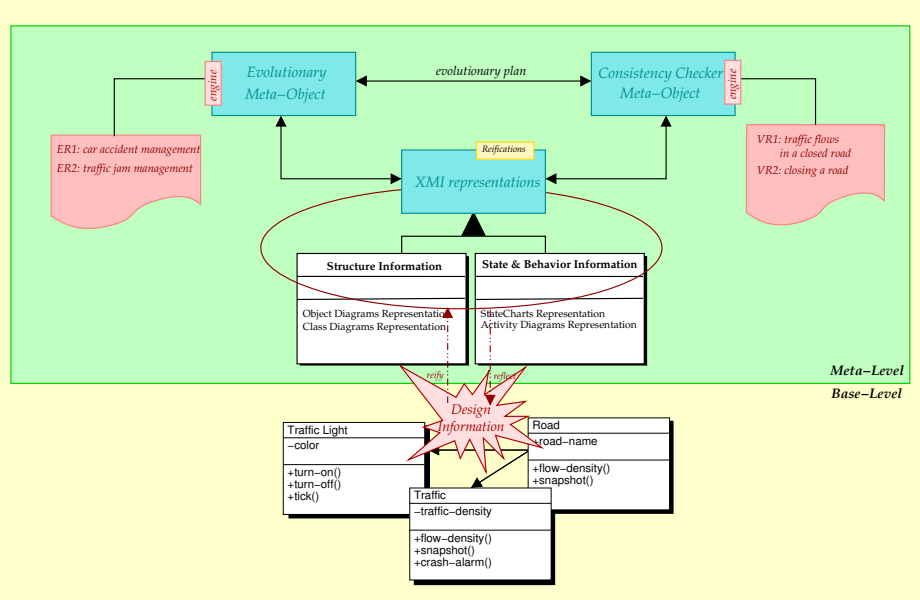

Fig. 3. XMI reification data and UML UTCS flow density.

evolution, and how the UTCS design information is managed by using the XMI encoding.

In this example we consider as unexpected events only car accidents. Therefore, we will show some evolutionary and validation rules that, used in conjunction with our approach, force the UTCS evolution for dealing with problems due to car accidents, e.g., traffic jams.

4.1 Evolutionary and Validation Rules against Car Accidents

For UTCS to deal with traffic jams must monitor vehicular flow density[5], and the status of every involved entity (both fixed and mobile). The evolutionary meta-object detects when the vehicular flow augments in a specific street and plans to adapt the current traffic schedule (i.e., the UTCS behavior) to face the problem.

[5] UTCS is supported by CCD-Cameras and movement sensors installed in every important nexus [18]. CCD-cameras take a photo every second and by comparing these photos, we can estimate the traffic flowing density. Sensors will notify anomaly events that cannot be detected by CCD-cameras like traffic light disruptions or damages to the road structure.

Evolutionary Rules

ER_1 (car accident management): when there is a notification of a car accident in a specific road and the traffic in that road is stuck then such a street must be closed and the traffic flow hijacked towards the adjacent streets.

> **on** *(a car accident in street R_i has been detected)*
> **if** *(flow density of R_i is too elevate)*
> **do** *closes R_i and reschedules the traffic lights so that cars avoid to pass through R_i.*

Obviously, the traffic lights rescheduling implies a change in the statechart of the traffic lights, whereas closing the road means to add a new traffic light in the system. The evolutionary meta-object will render effective these considerations manipulating the corresponding XMI schemas and forming the evolutionary plan to pass to the consistency checker meta-object.

ER_2 (traffic jam management): the density of the vehicular flow is constantly monitored road by road thanks to automatic cameras installed at the entrance of each road. These cameras take a snapshot of the traffic entering in the road every few seconds (delta that can be changed to have a more timely reaction), consecutive snaps are compared to establish if the vehicular flow has overcome a given tolerance threshold, if so the temporization of the traffic lights in the corresponding area are modified to allow a more fluid circulation in the road.

> **on** *(comparing two consecutive snapshots)*
> **if** *(the traffic flow has increased overcoming the tolerance threshold)*
> **do** *control the traffic lights and modify their temporization.*

As for **ER_1**, changing the traffic light temporization implies a modification of the statechart associate to the involved traffic light.

Validation Rules

VR_1 (traffic flows in a closed road): when the evolutionary meta-object proposes an evolutionary plan in which traffic is inhibited to a certain road, the consistency checker must verify that there is not any road whose traffic flows in the inhibited road.

> **on** *(traffic has been inhibited to road R_i)*
> **if** *(there is a road R_j whose traffic still flows into R_i)*
> **do** *inconsistency has been detected, reject the plan.*

The checking for consistency implies a complete scan of the object models and statecharts of the roads whose traffic usually flows in the closed road.

Note that we have decided of rejecting the evolutionary plan but another strategy should consider to fix the evolutionary plan against the detected problems instead of rejecting it.

VR_2 (closing a road): another important aspect that must be validated before rendering effective the planned evolution is to determine when it is safe to schedule the changes. In our example this mean to wait that all cars are halted at the entrance of the road to close before changing the direction of the vehicular flow.

> **on** *(evolutionary plan has been authorized)* $\wedge$ *(road* R_i *must be closed)*
> **if** *(no car is entering in* R_i*)*
> **do** *turns red the traffic lights in* R_i *and applies the evolutionary plan*.

This rule monitors the traffic light statecharts and intervenes on that when it is feasible before applying the evolutionary plan.

The rules showed in this section do not pretend to cover every aspect of the evolution but they want only to give a glance at the possibilities offered by our approach. Moreover the rules are expressed by using the natural language because the scope of this paper consists of describing our approach to evolution and we prefer to avoid complicated formalisms that would have obscured the simplicity of the mechanism.

5 Related Work

Several other researchers have proposed a mechanism for dynamic evolution by using a reflective architecture and design information. The system we consider in this short overview are UML virtual machine [17], The K-Component Architecture [10], Architectural Reflection [7] and design enforcement [19].

In [17] has been presented the architecture for a UML virtual machine. The virtual machine has a logical architecture that is based on the UML four-level modeling architecture and a physical architecture that realizes the logical architecture as an object-oriented framework.

Dowling and Cahill [10] have proposed a meta-model framework named *K-components*, that realizes a dynamic, self-adaptive architecture. It reifies the features of the system architecture, e.g., configuration graph, components and connectors. This model presents a mechanism for integrating the adaptation code in the system, and a way to deal with system integrity and consistency during dynamic reconfiguration.

Cazzola et al. [7] have presented a novel approach to reflection called *architectural reflection* which allows dynamic adaptation of a system through its design information. This has been possible moving the system software architecture from design-time to run-time. Software architecture manipulation allows adaptation in-the-large of the system, i.e., we can add and remove components but we cannot add functionalities to a component.

In [19] has been presented a method for design enforcement, based on a combination of reflection and state machine diagrams. Combining concepts of concurrent object-oriented design, finite state diagrams, and reflection leads to increase the reliability of the systems, by insuring that objects work in accordance with their design.

6 Conclusions and Future Work

The main topic of our work concerns with software adaptability. In this paper we have presented: i) a reflective architecture for dynamically and safely evolving a software system; and ii) the decisional engines and their rules which govern such an evolution. Finally, we have shown on a case study how to instruct our reflective architecture to adapt itself to unexpected events and how the evolution takes effect.

Our approach to software evolution has the following benefits:

- evolution is not tailored to a specific software system but depends on its design information;
- evolution is managed as a nonfunctional features, therefore, can be added to every kind of software system without modifying it; and
- evolution strategy is not hardcoded in the system but it can be dynamically changed by substituting the evolutionary and validation rules.

Unfortunately there are also some drawbacks: (i) we need a mechanism for converting UML diagrams in the corresponding XMI schemas (problem partially overcome by using *Poseidon4UML* [2]); (ii) decomposing the evolution process in evolution and consistency validation could be inadequate for evolving systems with tight time constraints.

In future work, we plan to overcome the cited drawbacks and to implement a prototype of the described architecture by using OpenJava for supporting the causal connection among meta-level representation and the base-level system and a scripting language such as Ruby or Python for specifying and interpreting the rules.

References

1. James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for XML. In *Proceedings of the 11th International World Wide Web Conference, WWW2002*, pages 486–495, Honolulu, Hawaii, USA, May 2002. ACM Press.
2. Marko Boger, Thorsten Sturm, and Erich Schildhauer. *Poseidon for UML Users Guide*. Gentleware AG, Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany, 2000.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
4. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.
5. Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.
6. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Reflective Analysis and Design for Adapting Object Run-time Behavior. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, Lecture Notes in Computer Science 2425, pages 242–254, Montpellier, France, on 2nd-5th of September 2002. Springer-Verlag. ISBN: 3-540-44087-9.
7. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 263–266, Cocoa Beach, Florida, USA, on 12th-15th October 1999.
8. Stefan Conrad and Can Türker. Prototyping Object Specifications Using Active Database Systems. In A. Emre Harmancı, Erol Gelenbe, and Bulent Örencik, editors, *Proceedings of the 10th International Symposium on Computer and Information Sciences (ISCIS X)*, Volume I, pages 217–224, Kuşadası, Turkey, October 1995.
9. Jim Dowling and Vinny Cahill. Building a Dynamically Reconfigurable Minimum CORBA Platform with Components, Connectors and Language-Level Support. In *Proceedings of the IFIP/ACM Middleware 2000 Workshop on Reflective Middleware*, New York, NY, USA, April 2000. Springer-Verlag.
10. Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.
11. Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Massachusetts, 1997.
12. Timothy J. Grose, Gary C. Doney, and Brodsky Stephan A. *Mastering XMl: Java Programming with XMl, XML, and UML*. John Willy & Sons, Inc., April 2002.
13. Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

14. Jeff Kramer and Jeff Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEEE Proceedings Software*, 145(5):146–154, October 1998.
15. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.
16. OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.
17. Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In Linda Northrop and John Vlissides, editors, *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 327–341, Tampa Bay, Florida, USA, October 2001. ACM Press.
18. Andrea Savigni, Filippo Cunsolo, Daniela Micucci, and Francesco Tisato. ESCORT: Towards Integration in Intersection Control. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8^{th} Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 2000.
19. Shaul Simhi, Vered Gafni, and Amiram Yehudai. Combining Reflection and Finite State Diagrams for Design Enforcement. 2(4):269–281, 1997.
20. Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.
21. Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, Inc., 2000 Corporate Blvd.,N.W., Boca Raton, Florida 33431, 1996.