Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02), *pages 35–54, Højstrupgård, Helsingør, Denmark, September 2002. Microsoft Business Solutions.*

Framework Patterns for the Evolution of Nonstoppable Software Systems

Walter Cazzola[1], James O. Coplien[2], Ahmed Ghoneim[3], and Gunter Saake[3]

[1] Department of Informatics and Computer Science,
Università degli Studi di Genova
Via Dodecaneso 35, 16146, Genova, Italy
cazzola@disi.unige.it

[2] University of Manchester Institute of Science and Technology, United Kingdom, and
Computer Science, North Central College, Naperville, Illinois
jocoplien@cs.com

[3] Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D-39016 Magdeburg, Germany
{ghoneim|saake}@iti.cs.uni-magdeburg.de

Patlet. *The fragment of pattern language proposed in this paper, shows how to adapt a nonstoppable software system to reflect changes in its running environment. These framework patterns depend on well-known techniques for programs to dynamically analyze and modify their own structure, commonly called computational reflection. Our patterns go together with common reflective software architectures.*

Keywords: Pattern, Framework Pattern, Pattern Language, Reflection, Software Evolution, Dynamic Reconfiguration, Reconfiguration of Nonstoppable System.

1 Context Overview and Case Study

A nonstoppable software system with long life span — that is, a software system whose execution can not be halted for allowing system reconfiguration —, has to be able to dynamically adapt itself to changes to its environment, i.e., to evolve itself. To render a nonstoppable software system *self-adapting* to changes to its environment is a topical issue in the software engineering research area. *Computational reflection* [4, 15] is one of the most used mechanisms for getting software adaptability [8, 18, 19]. Two aspects control the evolution of such kind of systems: *behavior*, and *dependencies*. Both of them can be involved in system evolution to comply with changes to system requirements.

When designing *urban traffic control systems* (UTCS), the software engineers must face many issues such as distribution, complexity of configuration, and reactivity to the environment evolution. Moreover, modern urban agglomerates provide a lot of unexpected hard to plan problems such as traffic lights disruptions, car crashes, traffic jam and so on. In [17] all these issues and many others are illustrated.

The evolution of complex modern cities has 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).

Figure 1, shows a possible object model for the UTCS described by using the UML formalism [2]. Moreover, Fig. 1 shows how the UTCS can be integrated with our reflective approach to evolution [5], i.e., the approach we are expressing as pattern language. This model includes two types of objects:

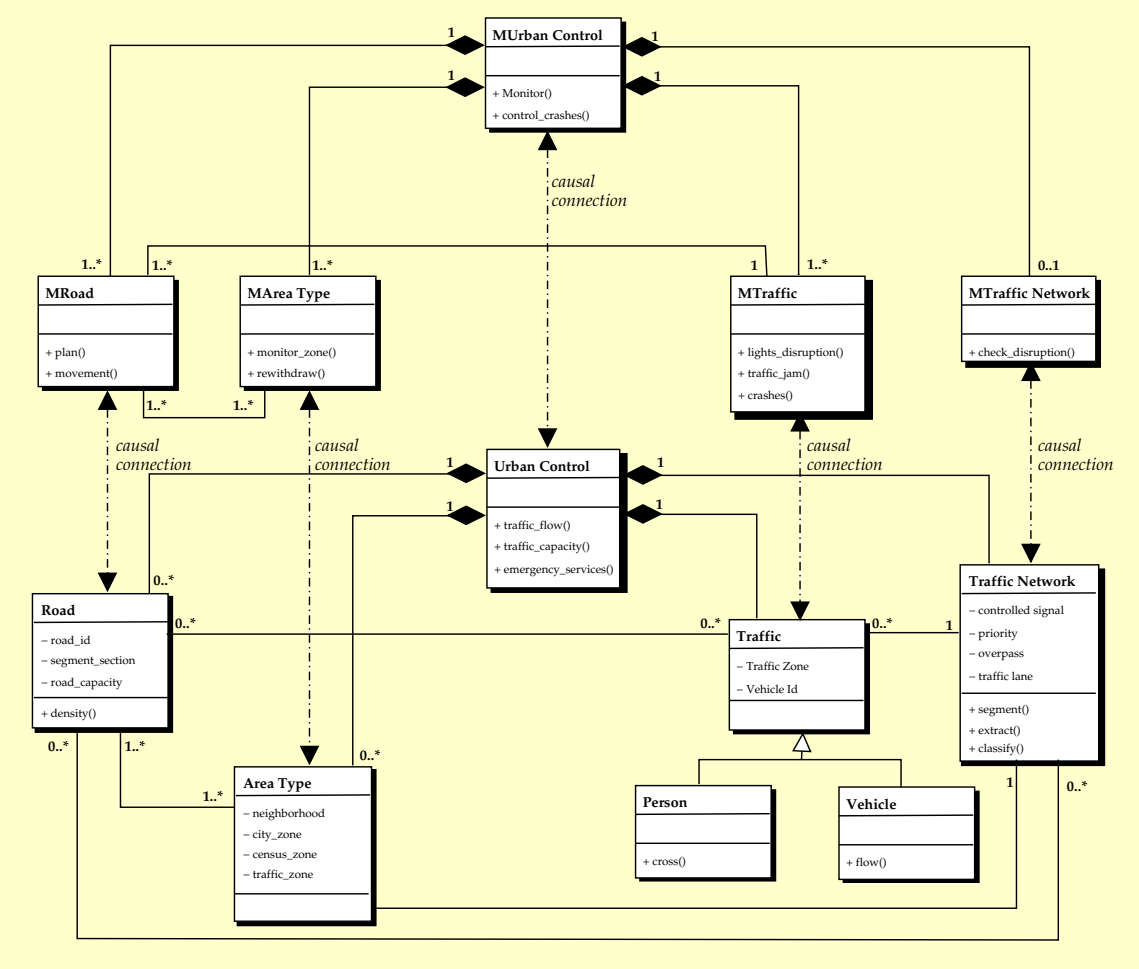

Fig. 1. Urban reflective object model.

- base-objects, e.g., urban control, roads, and so on, whose interactions describe UTCS' structure and its behavior in expected situations;
- meta-objects, associate to the base-objects, whose work consists of evolving the base-objects to deal with unexpected situations.

As mined from the ESCORT project [17], UTCS to deal with all the unexpected problems has to monitor vehicular flowing density[4], and the status of every involved entity (both fixed and mobile). The meta-level detects every anomaly of the UTCS and adapts the current traffic schedule (i.e., the UTCS behavior) to face the problem.

It is fairly evident that modeling and developing a urban traffic control system is an hard job for software engineers. The most important issues they have to deal with are: slowly evolving road situation, that the model must reflect accurately at all times, changes to the road situation that happens with no warning (accidents, broken traffic lights etc.) and that the system must take into account immediately, and of course the ever changing flow of people and vehicles and the dire consequences of restarting the system during rush hour or at all.

[4] UTCS is supported by CCD-Cameras and movement sensors installed in every important nexus [17]. 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.

There are many other nonstoppable systems that have problems similar to the urban traffic control system. Air traffic control, assembly line and nuclear station power are some examples of this kind of systems. Their problems are related to the fact they are nonstoppable and need a higher reactivity to sudden environmental changes. We do not further face these systems but the pattern language we propose can also be used for modeling them as well as the urban traffic control system.

2 Reflection, Reflective Architectures and Evolution of Nonstoppable Systems

Reflection is the ability of a system to watch its own computation and possibly change the way it is performed. Observation and modification imply an "underlay" that will be observed and modified. Since the system reasons about itself, the "underlay" is 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*. 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. 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[5]. 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 [4, 15].

A reflective architecture represents the perfect structure that allows a nonstoppable system to easily evolve. In [5] we have described a reflective architecture for the evolution of nonstoppable systems. In this framework the system running in the base-level is the nonstoppable system prone to be adapted, whereas the nonfunctional feature realized by the meta-level is the system evolution. Evolution takes place exploiting design information concerning the nonstoppable system.

To correctly adapt the base-level system, the meta-level has to face many problems. The most important are: (1) to keep consistent the base-level with its representative in the meta-level, when the base-level evolves, (2) to adapt the reification of the base-level to sudden changes through the design information of the base-level, (3) to verify whether the proposed adaptation would leave the base-level coherent and then to schedule its realization, finally (4) to reflect the modified reification on the base-level. Both reflection and adaptation involve several aspects of a system, the most important are: structure (objects, methods and so on), behavior (state and semantics of the objects) and collaboration (roles, exchanged messages, interfaces and so on).

In this work we describe a fragment of the framework pattern language for dynamically evolving a software system. We talk about framework pattern language because our patterns describe a framework and each of them entrusts part of its execution to other patterns in the

[5] 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.

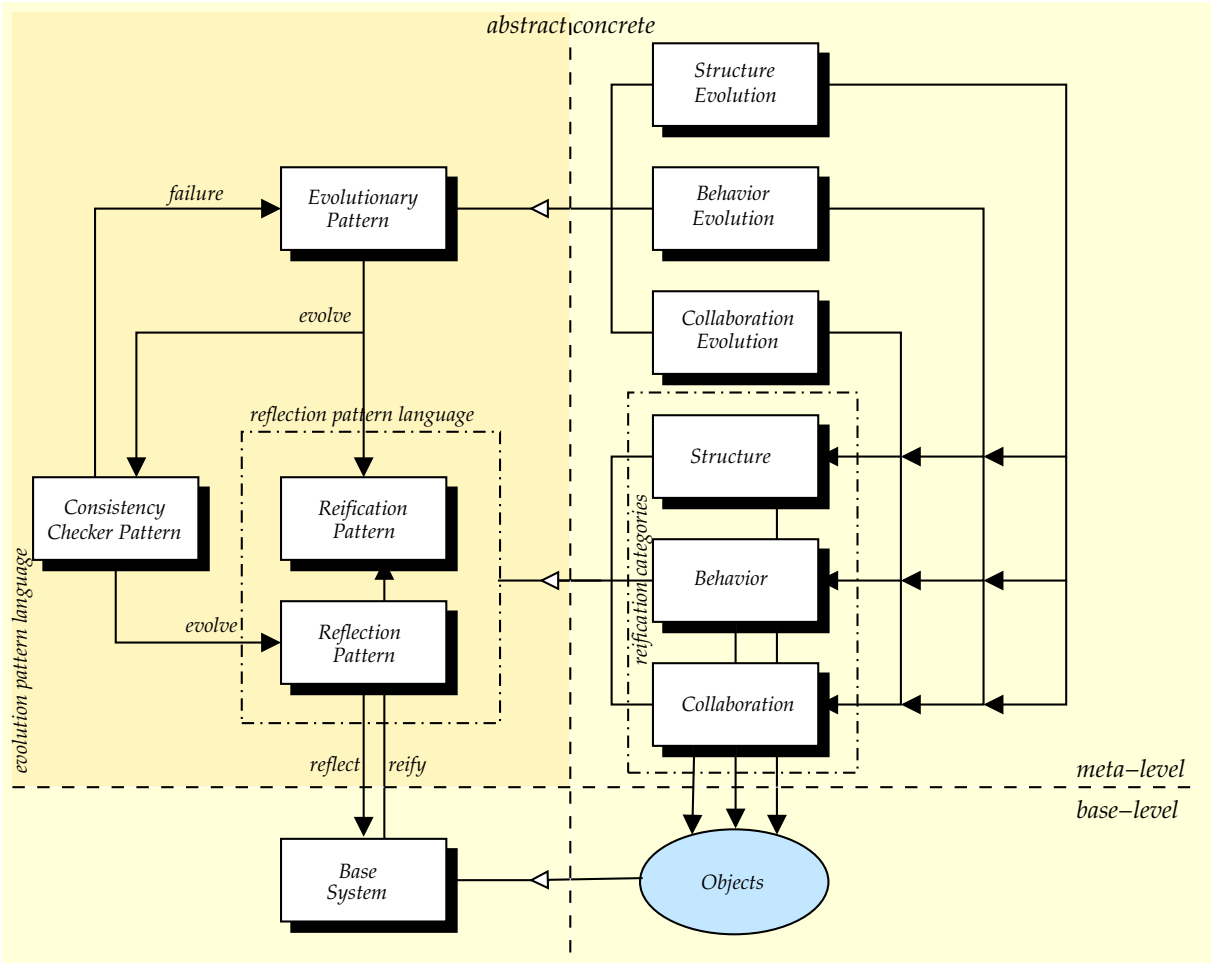

Fig. 2. Framework patterns for the evolution of nonstoppable software systems.

language, i.e., these patterns are tightly coupled. Points (1) and (4) in the above list are entrusted to the pattern language for computational reflection composed of *Reification* and *Reflection Patterns*, the evolution (point (2) in the list above) is entrusted to the *Evolutionary Pattern*, while the *Consistency Checker Pattern* takes care of checking the feasibility of the adaptation and of scheduling its realization (point (3)). Figure 2 sketches our framework, only patterns in the darker box on the left will be explored in this work.

The pattern language we are going to introduce can be used to adapt a nonstoppable system to sudden changes to its requirements or to its environment. This approach to evolution must be integrated with both the adaptation scheme and the consistency constraints used by the *Evolutionary* and *Consistency Checker* patterns; an possible adaptation scheme based on design information is described in [18]. Note that, reflection patterns does not force to choose a programming language with reflective features but its implementation can take advantage from it.

3 Pattern Language for System Evolution

In this section, we define the fragment of pattern language for the evolution of nonstoppable software systems. We have four main patterns: *Evolutionary*, *Reification*, *Reflection* and *Con-*

sistency Checker Pattern. Evolutionary patterns are responsible of adapting the system to incoming requirements. They work on the system reification supplied them by the reification patterns. The consistency checker pattern verifies the feasibility of the adaptation provided by the evolutionary patterns. Finally, the reification and reflection patterns, these patterns are part of the pattern language for computational reflection, are responsible for reifying the base-level system and, for reflecting the adaptation on the base-level, when the consistency checker pattern approves such an adaptation. Both evolutionary and reflection patterns describe a general behavior that can be applied to many domains, as shown in Fig. 2, whereas the consistency checker pattern collaborates with every application of the reflection pattern.

EVOLUTIONARY PATTERN

Intent. *To enable a nonstoppable software systems to adapt itself to dynamic changes to its requirements.*

Problem

Several times a (nonstoppable) software system must evolve to adapt itself to the evolution of the environment it is modeling. For example, in a software system as the UTCS (Sect. 1), this involves changes in the overall structure and behavior of the system, i.e., new components to interact with and a reorganization of the components interactions. If changes are planned a lot of time in advance, it is not a problem to take advantage of a moment when the traffic is low, for stopping the UTCS for a while, just the time for the reconfiguration. This is not a feasible solution when the change suddenly happens such as in case of a road interruption due to a road accident or something similar. In a similar case, we cannot stop the normal execution of the UTCS, creating many problems in other parts of the system, to face the unexpected situation.

Forces

Several forces are involved in the dynamic evolution of a nonstoppable system, obviously the most important is represented by the fact that:

- keeping still for a while (during the reconfiguration) a nonstoppable system could have dire consequences up to and including death.

Other not so important forces are:

- the system has to change whenever the environment it is modeling changes;
- changes to the environment can happen at all times, they are outside the control of the system and can not be foresee during system designing;
- reconfiguring the system in accordance with the changes in the environment is not easy and always feasible; moreover the cost of errors can be very high;
- to limit the problems, system stoppings must be planned in advance (e.g., roads impracticability is notified to drivers weeks in advance) and have to be scheduled in noncritical moments (e.g. signals maintenance is performed during night).

Solution

It is fairly evident that to render a nonstoppable system in compliance with the above forces a specific mechanism for adapting the system to environmental changes is needed. Adaptation takes place on a representative of the system, i.e., a group of *reification categories* [5]. In this

way, the adaptation mechanism does not interfere with the current execution of the system it is adapting, preserving the nonstoppable property of the system. Moreover, working on a representative rather than directly evolving the system provides an implicitly fault tolerance of the adaptation mechanism. Once the adaptation has been completed, the synchronization of the representative with the original system is delegated to the reflection pattern, before really modifying the original system, the soundness of the adaptation is verified by the consistency checker pattern. Examples of this approach can be found in [18].

Implementation

Here we show an algorithm illustrating the basic steps carried out by the evolutionary pattern. This algorithm is realized by using the C++ language.

```
template<typename aspect> class evolutionary {
  public:
    evolutionary(reification_category<aspect> a) : _representative(a) {}
    void do_adaptation(event e) {
          // it retrieves from the plan the rule to face with the event.
      void (*adaptation)(reification_category<aspect>, event);
      adaptation = _plan.get_action(e);
      adaptation(_representative, e);          // it carries out the adaptation.
      _representative.changed();               // it notifies the attempt of evolution
    }
    void inconsistency_detected() {
        // when called an inconsistency has been detected, it tries to solve such an inconsis-
        // tency exploiting its plan.
    }
  private:
    reification_category<aspect> _representative; // representative of the base-level aspect
    plans<aspect> _plan;                          // the rules it follows for adaptations
};
```

The evolutionary class is parametric on the representative it has the intention of adapting. This means that a class describing the aspect to adapt has also to be provided and to be used to instantiate the evolutionary class (see the second row of the code below).

The evolutionary pattern works on a reification category, i.e., on a representative of the software system (more on the representative is explained in section 3.1 when we describe *reification/reflection* patterns). The representative, of course, depends on the aspect of the system this class will deal with.

Another important element are the rules adopted by the algorithm for adapting the representative. These rules are represented by an instance `_plan` of the `plans` class. The `do_adaptation()` of the evolutionary object asks `_plan` for the adaptation rule to apply when an event happens (see row 7 in the code above). Then the evolutionary applies the adaptation rule to the representative.

```
reification_category<structure> str; evolutionary<structure> structure_evolution(str);
reification_category<behavior> beh; evolutionary<behavior> behavior_evolution(beh);

bool on_external_event(event &e) {
     // when an external event has occurred returns true then its argument refers to the
     // occurred event.
}
```

```
int main() {
  event e;
  while (true)
    if (on_external_event(e)) {
       structure_evolution.do_adaptation(e);
       behavior_evolution.do_adaptation(e);
    }
}
```

The evolution takes place when an external event, i.e., an event that has not been generated by the nonstoppable system, happens. The `inconsistency_detected()` method is invoked by the implementation of the consistency checker pattern.

In a complex system, as shown in the code above and in Fig. 2, there will be as many instances of the evolutionary pattern as many aspects of the system have to be adapted.

Applicability

The evolutionary pattern can be used to dynamically reconfigure a system (not necessarily a nonstoppable system) as a reaction to external events, such as anomalies detected by electronic devices. Moreover, the evolutionary pattern can be used to dynamically extend a running system with new features, components, and relations between them. The evolutionary pattern can be used to adapt the behavior as well the structure of the system as well the components interaction.

Known Uses

The evolutionary pattern is applied in the traffic control system realized in ESCORT [17] for controlling the traffic lights in accordance with the density of the vehicular flow as shown in Figure 3. This pattern interacts with elements of the object category like roads, traffic lights, urban control, traffic network, and elaborates the photography survey at real-time. In this case, the adopted evolutionary plan consists of comparing images of the traffic flow taken at different (adjacent) times (t_k). This comparison is done by pruning noises from images by using filters, by segmenting and classifying the images, and estimating the motion for images. Then, evolutionary pattern estimates the density of vehicular flow in a certain road in accordance with these values the traffic lights stay red or green for more or less time.

Collaborations

The evolutionary pattern, as shown in the applicability section, can be used stand alone, but in our work is only a part of a larger overall, so it implicitly interacts with: reflection, reification, and consistency checker patterns.

As explained, the evolutionary pattern observes the environment changes and adapts the base-level representative. The consistency of the representative is validated by the consistency checker pattern when the evolutionary pattern finishes the adaptation. If the validation fails the control returns to the evolutionary pattern for fixing the problem otherwise the base-level is conformed to the representative by the reflection pattern. The representative adapted by the evolutionary pattern is kept up to date by the reification pattern.

Consequences

The evolutionary pattern provides the following advantages:

– provides an implicit mechanism for dynamically evolving a system;

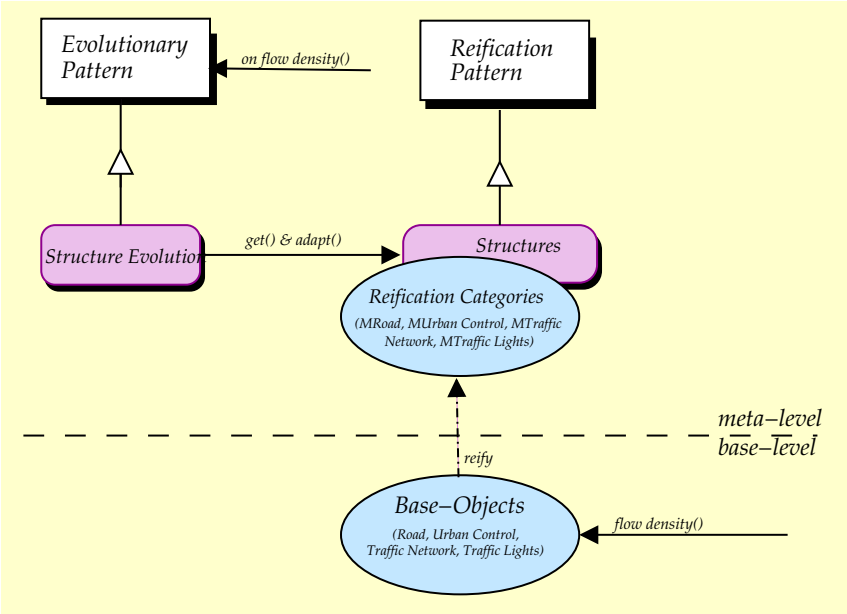

Fig. 3. The application of the evolutionary pattern in the UTCS

– provides a uniform way to evolve every aspect of a system, they could also be evolved separately;

and drawbacks:

– the nonstoppable system has an overhead when external events occurs and adaptation is needed;
– the system need extra code and data structures representing the system, its behavior and the evolutionary rules.

A critical role is played by the adaptation rules, they are the core of the evolutionary pattern and their realization is very hard because badly designed or applied at the wrong time adaptations could have very dire consequences, e.g., consider the chaos generated by stopping the traffic lights in a very busy area during the rush hour.

Related Patterns

The dynamic object model [16], allows the types of a system objects to change at run-time. This has been done by adding new types, changing existing ones, and changing the relationships between them. The work by Foote and Yoder [9], present three evolution patterns, *software tectonics* shows how continuous evaluation can be achieved without failure. *Flexible foundations* presents the need for continual and incremental evolution for systems. *Metamorphosis* presents mechanisms that allow the system to evolve dynamically.

The reflective state pattern [14], that is a refinement of state design pattern [10] based on the reflection architectural pattern [3]. The work by Yacoub and Ammer [20] represents the statecharts patterns and their relation to finite state machine patterns. This is done through defining these patterns: *basic statechart*, *hierarchical statechart*, *orthogonal behavior*, *broadcasting*, and *history-state*.

Intent. *To verify the feasibility and the soundness of the changes "proposed" by evolutionary patterns. That is, to check if it is possible to apply such changes without rendering inconsistent the base-level system.*

Problem

The delicacy of dynamically changing (part of) a component of a system is fairly evident. Usually changes directly affect only (part of) a component rendering simple to verify the effects of the changes. In complex systems each component cooperates with, integrates/is integrated in, uses/is used by other components, therefore, the effect of changes performed on a component are propagated to many other components not directly involved by the modification. Hazardous changes to a component will conflict with the overall behavior of the system and such conflicts are quite difficult to be detected. This problem is further amplified by the fact that the system can not be stopped hindering an easy reconfiguration and validation of the complete system. For example, in the UTCS, at a crossroads we can not turn green a traffic light without considering the state of the correspondent traffic light for pedestrians.

Forces

The forces involved by the consistency checker are:

- changes to the system are proposed as a reaction to changes in the environment;
- changes to the system are made on a component basis whereas their impact usually affects more than a single component;
- changes to the environment can frequently occur and can impact on many system components;
- inconsistencies due to hazardous changes are difficult to be detected.

Therefore, it is important to verify that environmental changes impacting on many components do not generate conflicts in the overall system and the effect of these environmental changes has only to be propagated to the system components when it is safe, i.e., when the propagation do not leave the system in an inconsistent state. Moreover, it is important to schedule the adaptation before its effects become obsolete or unnecessary.

Solution

It is fairly evident from the problem description that every "proposed" change to a component of the system has to be well planned and validated against inconsistency. Hence we need a mechanism that applies the changes to the system only when the "proposed" changes are proved to leave consistent the system. Moreover, such a mechanism has not only to guarantee against inconsistencies due to erroneous updates but also to choose the right moment for applying the "proposed" changes before their effects become obsolete.

The basic idea consists of gathering many "proposed" changes on representatives (the corresponding reification categories) of the system, checking step by step that replacing such a pool of representatives with the corresponding aspects of the system will leave the system in a consistent situation. Then, the replacement will take effectively place when the system is in a state that can safely be carried out and as long as such a replacement is necessary.

The effective updating of the system is delegated by the consistency checker pattern to the reflection pattern, as shown in the corresponding section this pattern will also choose the right

moment for render effective the update. Whereas, changes, that the consistency checker pattern considers that could render the system inconsistent, are returned to the evolutionary pattern for fixing.

Implementation

The code example below illustrates the basic steps carried out by the consistency checker pattern for verifying and maintaining consistent the base-level system after evolution. This algorithm is realized by using the C++ language.

```
class consistency_checker {
     // consistency checker working only on two reification categories: behavior and struc-
     // ture.
  public:
    consistency_checker(reification_category<structure> s,
       reification_category<behavior> b) : beh(b), str(s) {}
    bool check_consistency() {
       str.reset(); beh.reset(); // reset the notification of a change received by the evolutionary
       return plan.check_consistency(str, beh);
    }
  private:
    reification_category<structure> str;
    reification_category<behavior> beh;
    plans<consistency> _plan; // consistency plan
};
```

The consistency checker works on the whole system. It does not work only on a specific aspect but rather it has to maintain the consistency among every reification category of the system. Therefore, the C++ class describing the consistency checker must access to all the system representatives.

Consistency rules are an important element managed by the consistency checker. These rules are represented by an instance `_plan` of the `plans<consistency>` class and are used to determine if the representatives (in our code are represented by a behavior: `beh` and a structure: `str`) are a consistent snapshot of the system. The method `check_consistency()` of the consistency checker delegates `_plan` for such a check on the representatives (see row 8 of the code above).

```
reification_category<structure> str; evolutionary<structure> structure_evolution(str);
reification_category<behavior> beh; evolutionary<behavior> behavior_evolution(beh);
consistency_checker cc(str, beh);
reflection<structure> obj(str); reflection<behavior> state(beh);

int main() {
  while (true)
    if (str.is_changed() || obj.is_changed())
      if (!cc.check_consistency()) {
        structure_evolution.inconsistency_detected();
        behavior_evolution.inconsistency_detected();
      } else {
        obj.reflect();
        state.reflect();
      }
}
```

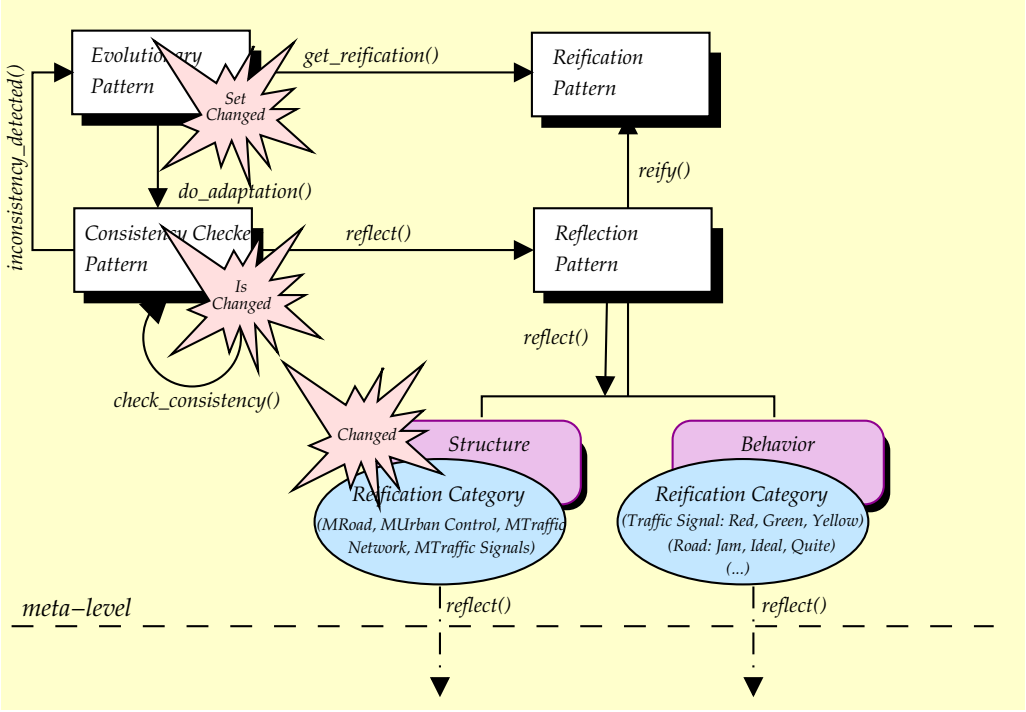

Fig. 4. An example of consistency checking against evolution of the system structure in UTCS.

Both the evolutionary and the consistency checker patterns work on system representatives. Evolutionary objects carry out their work when external events occur whereas the consistency checker performs its work when one of the representatives that it is monitoring is modified, that is, when an evolutionary object proposes a change. If evolutionary objects notify that they have carried out a change to the consistency checker, it is able to simply detect such a change in the representatives.

In Figure 4 we show the integration of the consistency checker pattern with the evolutionary and the reification/reflection patterns. The figure sketches consistency checker role in the UTCS by considering only two reification categories.

Applicability

The consistency checker pattern has to be used when we have to check the consistency of dynamic changes carried out by a system on a representative of another system before effectively performing such changes. The consistency checker pattern has to be used in critical environments to avoid the dire consequences of erroneous and inconsistent updates.

Known Uses

A feasible use of the consistency checker consists of checking the consistency of the base-level of a reflective system against changes performed by the meta-level system before reflection takes place. However this pattern is not mined from existing systems.

Collaborations

The consistency checker pattern can be used stand alone for checking the consistency of a sys-

tem or in collaboration with reification/reflection and evolutionary patterns for safely evolving a system:

- it compares the consistency of the reification categories embodied by the reification pattern. It uses a set of predefined rules;
- it interacts with evolutionary patterns to fix potential inconsistencies between the "proposed" adaptation and the referents;
- it delegates/authorizes the reflection patterns to update the corresponding aspects after the validation of the "proposed" adaptation.

Consequences

The consistency checker pattern provides the following advantages:

- It checks the consistency of the reification categories after evolution and before updating the base-level system in accordance with the adaptation. That is, it checks that carrying out the adaptation proposed by evolutionary patterns will not render the base-level system inconsistent.
- The control flow returns to the evolutionary pattern for fixing the proposed evolution if the consistency check fails. Otherwise, the control flow passes to the reflection pattern which carries out the proposed evolution.
- It looks, in collaboration with the reflection pattern, for the right moment to allow system evolution, i.e., the moment which guarantees that evolution leaves the evolved system working and consistent. It also associates an expiry date to the "proposed" adaptations that must still be applied and applies them only if such a date is not expired yet.

The consistency checker pattern has a few drawbacks:

- It augments the run-time overhead due to its checking and to its cooperation with evolutionary patterns for fixing the inconsistencies.
- Its work is based on a rigorous set of rules establishing when the system can be considered inconsistent. To write this set of rules is a delicate and difficult job.
- Adaptation does not immediately occur, could be postponed for long time and could never occur.

An important point is represented by the quality of the rules composing the validation system. This requirements is a very delicate point which requires a highly skilled software architect because all the efficacy of the consistency checker pattern is based on the quality of the validation system and a bad designed validation system can have dire consequences.

Related Patterns

CHECKS [6] is a pattern language for information integrity, presents two family of patterns. The first family of patterns considers quantities used by the domain model. They check your business logic capturing minimal variations in behavior (*Whole Value* pattern), non-applicable or exceptional quantities (*Exceptional value* pattern), and inappropriate combinations of values (*Meaningless Behavior* pattern). Whereas, the second family enables the direct and transparent manipulation of the domain model. This has been done by using these patterns: *Echo Back*, *Visible Implication*, *Deferred Validation*, *Instant Projection*, and *Hypothetical Publication*.

3.1 Pattern Language for Computational Reflection

This little pattern language has an intrinsic dualistic nature and it is composed of two patterns: *Reification* and *Reflection*. Their combined work allows the system to *export* (to *reify* by using a reflective parlance) a reification of a specific aspect of the system to another system for manipulation and to *import* (to *reflect* by using a reflective parlance) such a reification after the manipulation in the system again. After the terminology we have adopted in [5] reifications are called *reification categories*. These two patterns are not stand-alone but their work has to be integrated with the work of the other system. By the way, the sequence of application is reification then reflection (we have to export before manipulating and importing again).

The pattern language for computational reflection manages the interface between base- and meta-level composing the reflective system architecture. In the framework language for evolution we are describing, this pattern language can be considered as the glue sticking together the evolving system (the base-level using reflection parlance) and the system dealing with evolution (the meta-level) whereas the reification is the representative of one of the aspects of the base-level system. Working on a reification of the nonstoppable system allows the other patterns to "simulate" the adaptation without really affecting the real system.

REIFICATION PATTERN

Intent. *To export a representation of a system aspect both abstract and concrete to another system. Behavior, state, and code are some of the system aspects that can be exported.*

Problem

To monitor and manipulate an inner aspect of another software system means to be able to monitor and manipulate both low-level details (as in the case of system code) and abstract concepts (as in the case of the system behavior) and to access to the inner representation of another software system. This is not a simple job to carry out by using a nonreflective approach because they do not provide a mechanism which allows a system to access the inner representation of another software system forcing the programmer to tightly coupling the code of these two systems (supervisor and supervised). Moreover, it is also missing a high-level representation for some abstract aspects such as the behavior.

Forces

Manipulation of many aspects of a software system from another system is often forbidden due both to a different representation and to protection mechanisms. Moreover, it is a difficult job for a system to manipulate abstract concepts as behavior and collaboration via the API of a traditional programming language. We need to work on representatives without affecting the original system.

Solution

Rather than giving the supervisor access to the inner representation of the supervised, the supervised itself has to provides its own representation to the supervisor. This approach moves the responsibility of representing and therefore interpreting the inner aspects of a system from the supervisor to the supervised system, that is, from a system uncorrelated to such a data to the system owning them, with an obvious simplification. Hence for the system aspect we would reify, the system itself provides a data structure representing such an aspect and some operations

to interpret and manipulate it. Moreover, the representative (that is, the copy of the aspect that is local to the supervisor system) has to be kept constantly consistent with the aspect it represents, that is, the data structure has to be updated when a change in the aspect occurs. For example, in the UTCS, potential reification categories are roads and traffic lights status and their reifications have to be updated every time a road or a traffic light change its state.

Implementation

The implementation of the reification pattern is simple as well as the reflective API of the chosen programming language is articulated, this means that adopting Java instead of C++ the implementation could be simpler.

The code example below illustrates the basic steps carried out by the reification pattern for exporting an aspect of a system to another system. This algorithm is realized by using the C++ language.

```
namespace computational_reflection {

      // reification_category represents the base-level. It is a generic class that can be in-
      // stanced on behavior, structure, collaboration and so on.

  template<typename aspect> class reification_category {
    public:
      reification_category() : _changed(false) {}
      bool is_changed() {return _changed;}
      void changed() {_changed = true; ...}
      void reset() {_changed = false;}
      void local_update() { /* updates the content of the local copy */ }
      void merge_local() { /* merges the local copy to the reified aspect */ }
    private:
      aspect _reified;
      bool _changed;
  };

  template<typename aspect> class reification {
    public:
      reification(reification_category<aspect> a) : rc(a) {}
      void reify() {    // it reifies the corresponding base level aspect.
         rc.local_update();
      }
    private:
      reification_category<aspect> rc;
  };
}
```

Many reification categories can be necessary to represent every aspect of the system and each aspect can need a very different data structure to be represented. However these reification categories provide a common interface for manipulating themselves (such an interface is shown in the code above). The reification categories provide the mechanism used by the evolutionary pattern to notify a change in the representative to the consistency checker pattern (methods `changed()`, `reset()`, and `is_changed()`). Besides, they also provide the methods for keeping synchronized the copy with the original object (methods `local_update()` and `merge_local()`). These two methods directly deal with the base-level and their implementation can be easier if the adopted programming languages has reflective features.

In a system there are as many reification instances as reification categories, and each instance takes care of reifying the corresponding aspect. Therefore, the reification class is parametric on the aspect it is reifying. The reification class provides only the method `reify()`. It is used for reifying the aspect, its implementation does not vary when changes the reified aspect because it delegates the work to the `local_update()` of the corresponding reification category, the `local_update()` is directly related to the reified aspect and its behavior changes when the reified aspect is of a different kind.

```
bool on_event(event &e) {
    // when the base-level changes it returns true then its argument refers to the kind of
    // change occurred.
}

int main() {
  event e;
  reification_category<structure> str;
  reification_category<behavior> beh;
  reflection<structure> obj(str);
  reflection<behavior> state(beh);

  while (true)
    if (on_event(e)) {
       obj.reify(e);
       state.reify(e);
    }
 }
```

The representative of the aspect is updated every time the corresponding aspect changes due to the normal computation of the base-level. The updating takes place reifying the changed aspect on the reification category again.

Applicability

The reification pattern is a basic component for realizing the causal connection between two systems (see section 2 for a brief explanation about the causal connection relation). It can also be used every time it is necessary a local representative of a nonlocal entity, e.g., for implementing a remote method invocation in a client/server system, in this case the client asks the server for services through a representative. It is the server itself which renders available (exports) to the client such a representative (e.g., in Java through the RMI registry and the bind mechanism, see [11]) as in the reification pattern. Moreover, the reification pattern can be used to provide a uniform access to remote and distributed data, e.g., to implement a clustering file system *à la* MOSIX [1]. MOSIX provides to each computer in the cluster a virtual view of the clustering file system. Such a view is a composition of representatives of the singular file systems in the cluster and each change (e.g., to remove files) to the clustering file system affects only the corresponding representative and not the original file system. These representatives are provided by the kernel of the MOSIX system as it is done by the reification pattern.

Collaborations

The reification pattern is tightly coupled with the reflection pattern. Combining reification and reflection patterns grants the causal connection between these two systems. Moreover in the overall of our pattern language for evolution the reification pattern directly collaborates with:

– the evolutionary pattern providing the representatives it will use for evolution, and

– the consistency checker pattern providing a copy of the aspects that the consistency checker will use for validating the adaptation proposed by the evolutionary pattern.

Consequences

To export a representative of (part of) a system to another system has several benefits:

– it permits to simulate the adaptation of a system;
– it easily permits to verify/testing the system before rendering effective the change;
– it allows to modify the reified entity on a representative and testing the modification before rendering effective the change; if the changes are not satisfactory they can be easily discarded, discarding the representative without really affecting the reified entity.

Obviously there are also some drawbacks: the two most relevant are the extra overhead due to keep updated the representatives and the complexity of writing a reification category without using a reflective programming language.

Related Patterns

Reification means to turn something that you would normally not think as an object into an object. Several patterns in the literature approach to reification [10]: *Strategy*, *Mediator*, *Memento*. For example, memento, by widening its application domain to many other system aspects than the system state, can be used to realize our *reification categories*. In [13] a pattern language for implementing communication protocols has been presented, the exchanged data have been reified by the *data-path reification* pattern. The *Delegation pattern* [7], allows objects to share behavior without using inheritance and without duplicating code.

REFLECTION PATTERN

Intent. *To import a representation of a system aspect from another system. Behavior, state, and code are some of the system aspects that can be imported.*

Problem

How to take changes performed on a representative and implement them on the represented entity is not a simple job. We have to face two main problems:

– both represented entity and its representative belong to two separate systems, and
– there is not a simple and well-known mapping between an aspect of a system and its representative in another system.

The representation of an aspect of a system is handled by a component of another system. Each of these systems has its own access rights and usually it does not have the right to access the other system data.

Moreover, represented aspects can be both abstract or concrete concepts, where they are concrete when in the reified system there is a clear part of the code implementing those aspects, such as an object, or a method. Both kind of aspects are represented by a data structure. Therefore, there is no straightforward mapping between a change performed on the representative and its implementation on the represented aspect when it is an abstract concept without a direct counterpart in the system code (e.g., the system behavior).

These facts hinder the reintegration of the changes performed on the representative with the represented aspect.

Forces

The forces involved by the reflection pattern are:

- access rights and protection mechanisms hinder to import the changes made on a representative in the represented entity;
- the knowledge of many details of both systems is necessary to give a mapping between an aspect of a system and its representative in the other system;
- the mapping among these systems tightly couple them together;
- to implement a change carried out on a representation of an abstract concept is not easy;
- to update (or to replace) an aspect of another system is a very intrusive operation;
- we need a wide knowledge of the system to update, of its computational flow and when it is safe interacting with it.

Solution

Rather than enabling the system owning the representative to directly updating the represented aspect of the other system, the represented system itself will import the representative and will merge its content with the represented aspect. This approach overcomes the protection mechanisms because the system surely has all the necessary rights for modifying itself. Moreover, the system has a direct knowledge about its execution (e.g., if it is running or idle) rendering less intrusive the system updating, that is, the system itself will decide when it is time to update itself and if the update is not obsolete with respect to its current state (e.g., it avoids to update an object which does not exist anymore). Then the system itself will ask the supervising system for the content of its representative and will be simpler for it to map the changes on its inner representation than for another system.

Implementation

The consideration we have done for the implementation of the reification pattern still hold for the implementation of the reflection pattern.

The code example below illustrates the basic steps carried out by the reflection pattern for importing the changes done by a system on a representative into the original system. This algorithm is realized by using the C++ language.

```
namespace computational_reflection {
  template<typename aspect> class reflection {
    public:
      reflection(reification_category<aspect> a) : rc(a) {}
      void reflect() {rc.merge_local();}
    private:
      reification_category<aspect> rc;
  };
}
```

Both reflection and reification patterns are part of the same namespace and an instance of the reification class *shares* the reification category with an instance of the reflection class.

The reflection class, as also the reification class, is parametric on the aspect represented by the reification category. Moreover, it provides the method `reflect()` which reflects the changes carried out on the local representative on the corresponding aspect. Obviously the implementation of `reflect()` depends on the aspect we are managing, therefore, on the type of the aspect

and on its operations (`merge_local()` is a method belonging to the class of the aspect).

Applicability

The reflection pattern, as well as the reification pattern, is a basic component for realizing the causal connection between two systems. Moreover, it can be applied to synchronize the content of data structures shared among several processes and for serializing the concurrent writing access to a centralized datum.

Collaborations

The reflection pattern is tightly coupled with the reification pattern and together they grant the causal connection between two systems. Moreover in the overall of our pattern language for evolution the reflection pattern directly collaborates with the consistency checker pattern. The reflection pattern updates the reified system with the changes done to the corresponding local representative after the validation of the local representative performed by the consistency checker.

Consequences

To import a representative of (part of) a system from another system has several benefits:

- it permits to postpone the adaptation performed by another system to the most suitable moment;
- it frees the rest of the application from matters about data representation, protection mechanisms and so on.

The two most relevant problems are the extra overhead necessary for updating the original system and the difficult of modifying the original system in accordance with the content of the local representative above all without using a reflective programming language.

Related Pattern

The *Static Reflection* pattern [12], addresses the particular problem of building wrappers which contain functions which take C function pointers as parameters. The *Reflection Pattern* in [3] on page 193, provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. The *Microkernel Pattern* in [3] on page 171, applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts.

4 Conclusion and Future Works

In this paper we have addressed the problem of evolving a nonstoppable system at run-time through a reflective architecture. We propose a pattern language that describes the meta-level of the reflective architecture and how the components of the meta-level cooperate for evolving the base-level system. The meta-level system and its connection to the base-level system are modeled by *evolutionary*, *reification/reflection*, and *consistency checker* patterns. They cooperate in evolving the base-level system without affecting its consistency and functionalities. The evolution of a system can be settled by techniques based on different information, whereas the overall structure of the evolving mechanism is quite general. Therefore, in this work we have

just modeled the evolving architecture avoiding to fix a technique and detailing such a mechanism. In general we refer to reification categories, consistency rules, and so on, but their detailed description depends on the adopted techniques for evolution and it is beyond the scope of this work. An example of evolution based on design time information has been given by the authors in [5]. The pattern language for system evolution has been mined from the Escort system [17] which exploits a reflective architecture for supervising the traffic flow in the city of Milan.

Acknowledgments

Authors wish to thank Kristian Elof Søresen who has perfectly shepherded us to shape the paper at its best. They would also thank Mikio Aoyama, Richard Gabriel, Lars Grunske, Kevlin Henney, Juha Pärssinen and Michael J. Pont for the kind words they had for the early version of this pattern language and for their advice to render the paper what it is now.

References

1. Ammon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd, 1996.
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, 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.
6. Ward Cunningham. The CHECKS Pattern Language of Information Integrity. In James O. Coplien and Douglas C. Schmidt, editors, *Proceedings of the 1st Annual Conference on Pattern Languages of Programs (PLoP '94)*, Monticello, Illinois, USA, August 1994.
7. Dwight Deugo. Foundation Patterns. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the 5th Annual Conference on Pattern Languages of Programs (PLoP'98)*, Monticello, Illinois, USA, August 1998.
8. 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.
9. Brian Foote and Joseph W. Yoder. Evolution, Architecture, and Metamorphosis. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Proceedings of the 2nd Annual Conference on Pattern Languages of Programs (PLoP '95)*, Monticello, Illinois, USA, September 1996. Addison-Wesley Software Patterns Series.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.
11. jGuru. Fundamentals of RMI (Short Course). Available on `http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html`.
12. Bob Jolliffe. The Static Reflection Pattern. In Dwight Deugo and Federico Balaguer, editors, *Proceedings of the 8th Annual Conference on Pattern Languages of Programs (PLoP'01)*, Monticello, Illinois, USA, September 2001. Addison-Wesley Software Patterns Series.
13. Matthias Jung and Ernst W. Biersack. Order-Worker-Entry: A System of Patterns to Structure Communication Protocol Software. In Martine Devos and Andreas Rüping, editors, *Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLoP 2000)*, Irsee, Germany, July 2000.
14. Luciane Lamour Ferreira and Cecília M. F. Rubira. The Reflective State Pattern. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the Pattern Languages of Program Design*, TR #WUCS-98-25, Monticello, Illinois - USA, August 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. Dirk Riehle, Michel Tilman, and Ralph Johnson. Dynamic Object Model. In Eugene Wallingford and Alejandra Garrido, editors, *Proceedings of the 7th Annual Conference on Pattern Languages of Programs (PLoP 2000)*, Monticello, Illinois, USA, August 2000.
17. 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.
18. Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio. Architectural Reflection: Realising Software Architectures via Reflective Activities. In Wolfang Emmerich and Stephan Tai, editors, *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000)*, Lecture Notes in Computer Science 1999, pages 102–115. Springer-Verlag, University of California, Davis, USA, on 2nd-3rd of November 2000.
19. Emiliano Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 59–78. Springer-Verlag, Heidelberg, Germany, June 2000.
20. Sherif M. Yacoub and Hany H. Ammer. A Pattern Language of Statecharts. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the 5th Annual Conference on Pattern Languages of Programs (PLoP'98)*, Monticello, Illinois, USA, August 1998.