Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level

Walter Cazzola, Andrea Savigni, Andrea Sosio, Francesco Tisato
DISCO – Università di Milano Bicocca. Milan, Italy
emails: {cazzola|savigni|sosio|tisato@disco.unimib.it}

Abstract

As software systems become larger and more complex, a relevant part of code shifts from the application domain to the management of the system's run-time architecture (e.g., substituting components and connectors for run-time automated tuning). We propose a novel design approach for component-based systems supporting architectural management in a systematic and conceptually clean way and allowing for the transparent addition of architectural management functionality to existing systems. The approach builds on the concept of reflection, extending it to the programming-in-the-large level, thus yielding architectural reflection (AR). This paper focuses on one aspect of AR, namely the monitoring and dynamic modification of the system's overall control structure (strategic reflection), which allows the behaviour of a system to be monitored and adjusted without modifying the system itself.

1 Introduction

Any software system of some complexity devolves a relevant part of its code to *self-management activities* i.e., activities whose domain is the system itself. Examples include bootstrap, shutdown, and dynamic reconfiguration. In most cases, the system performs these activities on its own overall architecture, rather than its individual components. It is widely known that implementing this kind of functionality tends to be overly complex. We believe that the major source of such complexity is the *implicit architecture problem*. This term, that we introduced in a previous paper [1], means that most architectural issues are addressed in the components' code itself. For example, once the system architect has designed a protocol for components' cooperation, this protocol will be split into a specification of individual components' behaviour and then implemented by components' code. This code (implementing an architectural choice) will be intermixed with architecture-independent, functional code, thus hindering components' reuse.

In our previous papers [1, 2, 3], we have proposed a novel approach to component-based software development whereby architectures are made *explicit*. This approach, termed *architectural programming-in-the-large* (APIL), enforces a clean separation of concerns between *programming-in-the-small* issues and *programming-in-the-large* issues. In APIL connectors are designed to react to a set of *cooperation events* generated by their environment and, as a reaction, they trigger events on the components they are connected to. The overall behaviour of the system is defined by a strategy i.e., a plan stating which cooperation events should be triggered, and in which order. As a side effect, APIL also supports addressing self-management activities in a clean and systematic way, which can be regarded as an extension of the concepts and techniques of *reflection* to the architectural realm.

We coined the name *architectural reflection* (AR) to describe this approach to dynamic self-management regarded as the activity of a system performing computation on its own software architecture. AR builds on APIL in that adding architectural reflective capabilities to a system is made feasible by the explicitation of the architecture.

This paper focuses on one aspect of AR i.e., Strategic Reflection [2], which is the computation a system performs on its own behaviour at the architectural level, proposing a rule-based approach to the definition of a system's behaviour (its strategy). A companion paper discussing the other aspect of Architectural Reflection, namely Topological Reflection, will be written in the near future.

2 Architectural Reflection

Architectural Reflection is the computation performed by a system about its own software architecture [2]. As opposed to classic reflection, where actions are performed on a single entity or interaction, architectural reflection operates *in the large* i.e., on the whole architecture and on how components interact.

An architectural reflective system is structured into two layers, called respectively *base-layer* and *meta-layer*. The meta-layer is *architecturally causally connected* to the base-layer i.e., in the former entities work, called *architectural meta-entities*, which maintain data structures reifying the software architecture of the underlying layer; every change made to these data structures reflects on the underlying system architecture, and vice versa. Therefore, according to the concept of domain as used by Maes in [5], the application domain of the architectural meta-entities is the software architecture of the computational system.

We remark that both the base-layer and meta-layer entities are to be meant as *roles*, which need not be played by physically separate entities. This means that our model can accommodate even "ordinary", single-layered architectures, in which case the two layers are simply coincident, the architectural choices being dispersed among the components and connectors themselves (thus falling back into the IAP).

The property of transparency holds as in classical reflection i.e., the base layer is unaware of the presence and behaviour of the meta-layer. Based on our definition of topology and strategy as orthogonal aspects of software architecture, we can further refine the definition of architectural reflection by defining *topological reflection* (TR) and *strategic reflection* (SR).

TR is the computation performed by a system about its own topology. Examples of topologically reflective actions include adding or removing components or connectors. SR is the computation performed by the system about its own computation in the large i.e., observation of the abstract state of components and connectors and observation|manipulation of the base-layer strategy. An example of strategically reflective action is changing priorities associated to transitions in a priority-based strategy. Clearly, every change on the system architecture must take into account the restrictions forced by the original architecture, in order to keep the system consistent. As anticipated above, in this paper we confine ourselves to SR.

2.1 Architectural Base-Layer

Since the architectural base-layer is the "ordinary" part of our approach, it can be described in the usual way (see [6]) i.e., via *components* (the locus of computation) and *connectors* (the locus of cooperation between components). Therefore, in the sequel we will only cover those parts of the architectural base-layer that are original with respect to the literature, most notably *strategy*.

2.1.1 Strategy

The system's strategy is described by a set of *rules* governing the occurrence of cooperation events. Such description is actuated at run-time by a *strategy actuator* (SA in the following) that works as an inference engine interpreting rules and thereby triggering cooperation events. A rule-based approach was chosen due to its flexibility, since our intent was that of accommodating a wide range of global control policies.

A rule is an expression of the form:

```
rule <ruleName> {
  <rulePreconditions> → <ruleActions> [→ <rulePostconditions>]
}
```

The *preconditions* section is a boolean expression made up of two separate subsections, which refer to the state of connectors and the state of time respectively. Thus, we have:

```
<rulePreconditions> ::= <stateOfConnectors>
<rulePreconditions> ::= <stateOfTime>
<rulePreconditions> ::= <stateOfConnectors>, <stateOfTime>
```

The state of a connector can be defined as the set of cooperation events it is ready to accept. In this way, the state of a connector does not coincide with its internal state, but is rather an abstraction of it. Thus, the state of a connector is simply a list of allowed events; more formally:

```
<stateOfConnectors> ::=
  <connectorEnabledEvents>{, <connectorEnabledEvents>}
<connectorEnabledEvents> ::=
  <connectorName><enabledCooperationEvent>
  {, <enabledCooperationEvent>}
```

As far as time is concerned, the "`state of time`" serves to express both time-related constraints and time events. The former is a set of clauses such as "`time < 2:00pm`", while the latter can be expressed in the same fashion with expressions such as "`date=today&&time = 3.00pm`", where `date` and `time` are predefined variables that refer to the current date and time, and `today` is a predefined constant. In this way, a uniform notation can be used to express both events and constraints, which allows the designer to build extremely diverse systems. Formally:

```
<stateOfTime> ::= time <relop> <timeExpression>
```

where <relop> is the set of the usual relational operators and <time expression> can be expressed in one of the standard ways.

Clearly, each of the two constituents of the precondition section can be omitted; in this way very diverse systems can be designed, ranging from hard real-time ART systems, in which only the state of time section exists, to rule-based systems proper, where the order of rule activation is dictated entirely by the built-in inference engine of the SA.

The *actions* section of rules is simply a list of cooperation events triggered on the appropriate connectors. More formally:

```
<actions> ::= <connectorAction> {, <connectorAction>}
<connectorAction> ::=
  <connectorName> <cooperationEvent> {, <cooperationEvent>}
```

Note that a potential problem arises from the fact that multiple actions can coexist in the action section of a rule. In fact, one `connectorAction` might bring the system in a state that renders the preconditions of the following `connectorAction` false. In order to address this problem, essentially two approaches can be followed:

1. the SA, after executing each `connectorAction`, turns back to examining the state of the system (i.e., the state of the connectors) and decides whether to execute the next action;

2. the SA simply ignores the problem and leaves every such issue with the strategist (see section 2.2).

Following solution 1 above, it is the task of the SA to ensure system consistency, while adopting approach 2, inconsistencies are dealt with at the layer above.

The *postconditions* section is still a boolean expression that describes that state of the system after the rule has been executed. This state includes, as for the preconditions, the state of connectors (defined in the usual way), and the state of time; in this way, it is possible to specify time constraints on actions. To this aim, a predefined variable (called `elapsedTime`) is provided, which, used in conjunction with the usual relational operators, allows the designer to easily specify time requirements in the form of time elapsed from the moment the action starts to the moment the action ends. In symbols:

```
<rulePostconditions> ::= <stateOfConnectors>, <stateOfTime>
<rulePostconditions> ::= <stateOfConnectors>, <elapsedTime>
<elapsedTime> ::= timeElapsed <relop> <timeExpression>
```

Note that organising the rules and setting the appropriate priorities in order to ensure that time constraints are respected is entirely up to the SA; under this respect, the rules constitute a high-level specification of the system behaviour, which can be implemented in a number of different ways.

The overall definition of the strategy is as follows:

```
<strategySpecification> ::=
  strategy <strategyName> {
    [<ruleDefinition> [{, <ruleDefinition>}]]
  }
```

2.1.2 The APIL Virtual Machine.

The virtual machine of the APIL language is structured into a framework providing both a set of architectural primitives to be used for actuating the topology and strategy (e.g., to instantiate components and connectors, and to trigger cooperation events), and the two actuators which execute the topology and strategy description by using such primitives. This structure has several purposes, one of which will be illustrated in the next section.

2.2 Architectural Meta-Layer

The architectural meta-layer is the portion of an architectural reflective system devoted to observe and manipulate the software architecture of the underlying layer (base-layer).

The meta-layer works as a shell which wraps the base system. The domain on which it operates is the architecture of the base-layer (just as the base-layer operates in the application domain).

As discussed so far, in this work we only consider two aspects of software architecture: topology and strategy. Whatever the implementation, the meta-layer plays two roles: it observes/manipulates the base-layer's topology, and it observes/manipulates the base-layer strategy. We use the terms topologist and strategist to refer to these two roles, respectively. The topologist is whatever entity reflects on the base-layer's topology by using the topological primitives, and the strategist is whatever entity reflects upon the base-layer's strategy using the strategic primitives (described in section 3). The meta-layer can have any implementation (it may be a monolithic entity or a complex system, perhaps comprising a topologist entity and a distinct strategist entity; it can be centralised or distributed, and so forth). What actually defines AR is the fact that the strategist and topologist roles are played by entities which are distinct from those residing in the base-layer. We term these distinct entities (architectural) meta-entities. The interested reader can refer to [2] for a description of one possible architectural organisation of the meta-layer.

2.2.1 Topologist and Strategist.

The topologist reifies information about topology (components, connectors, and their attachments), while the strategist relies on topological information held by the topologist and reifies both the current state of components and connectors and the specific strategy at hand. Due to the fact that architecture is explicit, in order to access and to manipulate architectural information, the architectural meta-entities need only interact with the actuators of the underlying layer.

System bootstrap and shutdown can also be handled by architectural reflection, since they involve topological and strategic actions (creation and destruction of components, activation of initialisation activities, and so on). In this case, the entities playing the topologist/strategist roles must exist before and|or after the creation and|or destruction of the system.

Figure 2.2.1 represents an architectural reflective system. In the base-layer are components (grey spheres), connectors (little black spheres connected by arrows, the little spheres represent component ports), and actuators (big

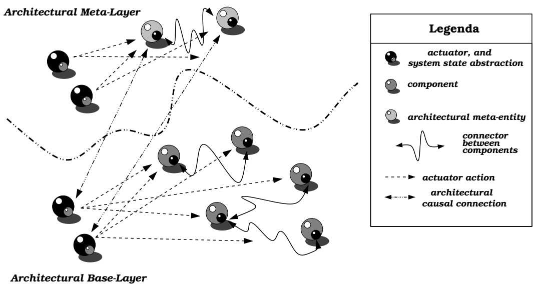

Figure 1. An architectural reflective system

black spheres) that shape and activate the system. Architectural meta-entities are represented by light grey spheres. They directly interact with the underlying actuators, which abstract (or reify to the meta-layer) topology, strategy, and system state. Architectural meta-entities operate on the underlying system's architecture by directing the underlying actuators.

3 Architectural Causal Connection

As in classic reflection, an architectural reflective system must keep, at the $(n+1)$-th layer, an appropriate representation of the n-th layer, and must be able to *reify* any changes in the layer below into its representation and to *reflect* any change in that representation into the n-th layer. In such a system, topology and strategy are reified at the meta-layer and any change in them is reflected in the base-layer; the entities in charge of maintaining such description of topology and strategy are the topologist and the strategist, respectively. Both are implemented based on a set of primitives of which, in accordance with the goals of the paper, we will only cover the strategy-related ones.

As explained above, the system behaviour is governed by the strategy actuator (SA), which accomplishes its task by executing a set of rules. In this context, the goal of the strategist is to *observe* the system behaviour and to *modify* it as needed.

Reifying the system behaviour means essentially two distinct things, namely knowing the rules, and observing the state of the SA. This means that the SA must export both a `getRule` and a `getState` primitive to the strategist.

As the system behaviour is dictated by rules, modifying behaviour implies modifying rules. Thus, the SA exports a set of primitives that allow the strategist to modify the rule set; following is a minimal set of those rules:

- `addRules`: adds a specified set of rules;
- `removeRules`: removes the specified rules;
- `inhibitRules`: specifies a set of rules that, even having their preconditions satisfied, should not be fired;
- `trigRules`: specifies a set of *privileged* rules i.e., a set of rules that should fire before the others (assumed, of course, that their preconditions are met).

4 Related Work

Several works address the problem of *specifying* software architectures (first of all [6]), but with no notion of an executable architectural description. Others (such as [4]) are more close to our approach, but none of them addresses dynamic modification of the architecture as a reflection problem, nor do they include the idea of maintaining at runtime a logically centralised description of the system's architecture.

5 Conclusions and State of the Art

This paper presents Strategic Reflection, an aspect of Architectural Reflection, which is an extension of classic reflection to the software architecture level. The basic application of this extension is to allow for a systematic and conceptually clean approach to designing systems with self-management functionality (such as dynamic reconfiguration) which also supports such functionality to be added to an existing system without modifying the system itself. We are now working both on a complementary paper that describes TR and on a prototype AR environment.

References

[1] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. A Fresh Look at Programming-in-the-Large. In *proceedings of 22nd Annual International Computer Software and Application Conference (COMPSAC'98)*, pages 502–506, Wien, Austria, on 19th-21st August 1998. IEEE.

[2] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *proceedings of 6th Reengineering Forum (REF'98)*, pages 12–1–12–6, Firenze, Italy, on 8th-11th March 1998. IEEE.

[3] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, Università degli Studi di Milano, May 1999. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[4] S. Ducasse and T. Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC'97*, LNCS 1301, pages 483–500. Springer-Verlag, 1997.

[5] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of oopsla'87*, volume 22 of *Sigplan Notices*, pages 147–156. ACM, Oct. 1987.

[6] M. Shaw and D. Garlan. *Software Achitecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.