Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, Università degli Studi di Milano, May 1999.

Architectural Reflection: Concepts, Design, and Evaluation

Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato

Department of Informatics, Systems, and Communication,
University of Milano Bicocca, Milano, Italy
{cazzola|savigni|sosio|tisato}@dsi.unimi.it

Abstract. This paper proposes a novel reflective approach, orthogonal to the classic computational approach, whereby a system performs computation on its *software architecture* instead of individual components. The approach supports system's self-management activities such as dynamic reconfiguration to be realized in a systematic and conceptually clean way and added to existing systems without modifying the system itself. The parallelism between such *architectural reflection* and classic reflection is discussed, as well as the transposition of classic reflective concepts in the architectural domain.
Keywords: Programming-in-the-large, Reflection, Software Architecture

1 Introduction

Within the software engineering field, growing interest is raised by the discipline of software architecture. This discipline focuses on the gross organization of software systems and seeks ways to gather and formalize the structural ideas underlying successful designs produced in current software engineering practice. The overall objective is that of providing a clean conceptual framework for the formalization of architectural knowledge so that architectural designs can be analyzed, compared, described in an unambiguous way, taught to practicing engineers, and so on. To that aim, software architecture has already gone a long way, and several prototype of Architecture Description Languages (ADLs) can be found in the literature.

We argue that the architectural level of abstraction is not only relevant in the design of systems; it is also a convenient point of view from which to describe and *implement* part of the system's functionality itself. Any software system of some complexity devolves some part (often a relevant part) of its code to *self-management activities* i.e., activities whose domain is the system itself. Examples are bootstrap, shutdown, online monitoring of the components' activity (especially, but not only, for distributed systems), and dynamic reconfiguration (with several goals, ranging from support for end-user-tailoring to fault tolerance). In most cases, such functionality deals with the architecture of the system rather than its components considered in isolation. This functionality can be regarded as a set of activities a system is capable of performing on its own architecture. Implementing this kind of functionality is usually overly complex; most systems provide limited capabilities of this kind, usually realized by ad hoc technical solutions. We believe that the major source of such complexity is the *implicit*

architecture problem affecting current software engineering practice i.e., the fact that architectural choices are dispersed in the components' code in implemented systems and intermixed with non-architectural (functional) code.

In a recent paper [4], we have proposed a novel approach to component-based software development whereby architectures are made *explicit*. In this approach, termed *architectural programming-in-the-large* (APIL), components are architecture-independent entities, and architectural choices are expressed in a *program-in-the-large* which controls the components' instantiation and behaviour according to the specified architecture. The original motivation for this work was that of enhancing components' and architectures' reusability by a clean *separation of concerns* between *programming-in-the-small* issues (i.e., those issues related to the components' inner semantics) and *programming-in-the-large* issues (those related to the system's overall architecture). As the work progressed, we observed that this approach 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 (i.e., to the programming-in-the-large level). We coined the name *architectural reflection* (AR) to describe this approach to dynamic self-management regarded as the activity of a system performing computations 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 architectural plan in a dedicated higher-level program.

AR has been described, in its very general lines, in [5]. In this paper, we dwelve deeper into the matter and describe AR by means of a thorough comparison with classic (computational) reflection, we describe the behaviour of architectural meta-entities, and we introduce the notion of architectural causal connection, which is the basic concept for AR. To that aim, a rather complete example is developed, illustrating how AR provides a clean and simple framework for managing dynamic reconfiguration activities. With respect to other approaches to system self-management, AR appears to be more systematic, easier, and to preserve the concept of *transparency* of classic reflection i.e., self-management activities can be added to any system without modifying any of its components.

The paper outline is as follows. Section 2 introduces preliminary notions for AR, including both a brief summary of concepts from classic (computational) reflection, a description of the implicit architecture problem, and the general lines of architectural programming-in-the-large. Section 3 introduces the concept of AR in its general lines, and compares AR with computational reflection. Section 4 introduces the concept of architectural causal connection within the context set by the previous sections. Section 5 provides an example illustrating the advantages brought by AR, in particular to dynamic reconfiguration. Section 6 compares our approach to other research efforts. Finally, section 7 draws some conclusions and describes future work.

2 Preliminary Concepts

2.1 Computational Reflection

Computational reflection, or reflection, is defined as the activity performed by an agent when doing computations about itself [15]. Behavioural and structural reflection are re-

flection sub-branches which involve, respectively, agent computation and structure [7]. behavioural reflection can be defined as the ability of the language to provide a complete reification of its own semantics as well as a complete reification of the data it uses to execute the current program. Structural reflection can be defined as the ability of a language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types.

A reflective system is logically structured into two or more levels, constituting a *reflective tower*. Entities working in the base level, called base-entities or reflective entities, define the basic system behaviour. Entities working in the other levels (meta-levels), called meta-entities, perform the reflective actions and define further characteristics beyond the application-dependent system behaviour.

Each level is causally connected to adjacent levels i.e., entities belonging in a level maintain data structures representing (or, in reflection parlance, reifying) the states and the structures of the entities in the level below. Any change in the state or structure of an entity is reflected in the data structures reifying it, and any modification to such data structures affects the entity's state, structure and behaviour.

Computational reflection allows properties and functionality to be added to the application system in a manner that is transparent to the system itself (separation of concerns) [20]. For a classification and comparison of classic approaches to reflection see [3].

2.2 The Implicit Architecture Problem

Component-Based Software Development (CBSD) aims at supporting the development of complex software systems through the assembly of large, independently developed, reusable components. CBSD brings forth on a distinction between *programming-in-the-small* (PIS, the development of individual components) and *programming-in-the-large* (PIL, the construction of systems out of components) [8]. While object-oriented technology provides a sound basis for the development of both fine- and coarse-grained reusable components, we still lack adequate notations and tools for programming-in-the-large. Such notations should let system designers specify the "plan of how components fit together and cooperate" [14] i.e., the *software architecture* of the system [19]. The architecture of a software system is defined by stating:

❶ how the overall functionality is partitioned into *components*;
❷ how the components are *topologically* arranged i.e., which interacts with which;
❸ the protocols used by components to communicate and cooperate i.e., which *connectors* exist between them;
❹ the *global control structure* of the system.

Existing notations for composing software modules are usually limited to expressing some small subset of these issues (e.g., topology alone). This implies that most architectural issues be *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. In current practice, all architectural choices follow this fate and get dispersed in the components' code (at PIS level) in implemented systems. This is what we termed the *implicit architecture problem* in a previous paper [5].

Building system with implicit architectures has several drawbacks: most notably, it hinders components' reuse due to the architectural assumptions components come to embed [11]; it makes it infeasible to reuse architectural organizations independent of the components themselves; it makes it overly complex to modify software systems' architecture; and it is also cause of the despiteful, yet empirically observed fact that architectural choices produced by skilled software architects are always distorted and twisted by implementers [16]. A more detailed discussion of the implicit architecture problem and its consequences can be found in [4].

2.3 Architectural Programming-in-the-Large

In a recent paper [5] we have proposed a novel approach to PIL, termed Architectural Programming-in-the-Large (APIL), explicitly aimed at solving the implicit architecture problem. In this approach, a system comprises a set of architecture-independent components (i.e., components embedding no architectural assumption) and a program-in-the-large prescribing how components should be assembled and interact i.e., the architecture of the system. The program-in-the-large is executed at run-time by a dedicated virtual machine to rule over the instantiation and behaviour of components.

More precisely, the overall system organization which results from our approach comprises three layers. At the first layer reside *components*. Each component interacts with the environment through a set of *ports* i.e., distinct interaction points. This interaction is modeled as a set of *events* occurring on ports; such events are generated by the environment and trigger a reaction within the component. The behaviour of a component (in terms of the events it can react to) is modeled as a state machine. At the second layer reside *connectors*. Connectors are entities ruling over the interaction among components. Each connects two or more ports belonging to two or more components. A connector reacts to a set of *cooperation events* generated by its environment and, as a reaction, it triggers events on the components' ports. The behaviour of a connector, in terms of the cooperation events it can react to and the events it generates on ports as a reaction to cooperation events, is also modeled as a state machine.

At the third layer resides the program-in-the-large virtual machine. The behaviour of the program-in-the-large virtual machine consists in actuating a description of the system's architecture comprising two aspects: *topology* and *strategy*. The topology describes the *structure* of the system i.e., which components and connectors comprise the system. Actuating the topology means instantiating components and connectors accordingly. The strategy defines the overall *behaviour* of the system. It is a plan stating in which order(s) cooperation events should be triggered in connectors. As a result of what stated above, this plan indirectly rules over the components' behaviour, since triggering a cooperation event indirectly triggers events on the components' ports.

In the APIL approach, architectures are *explicit* in the sense that the program-in-the-large is a description of the software architecture which, rather than being *implemented* inside components, *rules* over the components' creation, their interactions (by defining

connectors between components), and the system's overall flow of control (the strategy).

3 Architectural Reflection

Architectural Reflection is the computation performed by a system about its own software architecture [2].

An architectural reflective system is structured into several (potentially infinite) layers[1], called *architectural layers*, constituting an *architectural reflective tower*. Every layer is *architecturally causally connected* to the layer below i.e., in every *architectural meta-layer* entities work, called *architectural meta-entities*, which maintain data structures reifying the software architecture of the underlying system; 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 [15], the application domain of the architectural meta-entities is the software architecture of the computational system.

Each architectural layer has the necessary *hidden* hooks that allow it to be linked to a potential new meta-layer above, but each layer is created only when needed, in order to avoid an infinite regression.

Each architectural meta-layer operates on the architecture of the level below and adds new functionality to the original system. The property of transparency holds as in classical reflection i.e., each layer is unaware of the presence and behavior of the layers above.

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 and strategic reflection*.

Topological reflection is the computation performed by a system about its own topology. Examples of topologically reflective actions include adding or removing components or connectors.

Strategic reflection 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 strategy. An example of strategically reflective action is changing priorities associated to transitions in a priority-based strategy.

In the next sections we examine in more detail each layer of the architectural reflective tower. In particular we illustrate the agents working in each generic architectural base or meta-layer, their duties and the layers interconnection (the architectural causal connection).

3.1 Architectural Base-Layer

Based on the model described in section 2.3 for APIL, we have devised a notation for describing components, connectors, the system's topology and the system's strategy, as

[1] In order to avoid confusion between classic and architectural reflection, we will use hereafter the terms "level" and "layer" to distinguish, respectively, the levels of the classic reflective tower from those of the architectural one.

well as an interpretation model where dedicated system entities encapsulate architectural issues and rule over the execution of architecture-independent components. What APIL (notation and interpretation model) describes will be referred to, in this paper, as the *architectural base-layer*. This section dwelves into the details of how the architectural base-layer is structured and how it works, and introduces the APIL notation.

Components are the locus of computation. A component is made up of two parts: an architecture-independent *core* providing the required functionality and no architectural issues; and an *architectural component* wrapping the core to let its functionality be accessible according to the behaviour described in the APIL notation. A component's behaviour is described according to the following syntax[2]:

```
<componentSpecification> ::=
   component <componentName> {
     {<portSpecification>}
     <portDependencies>
   }

<portSpecification> ::= port <portName> {
  initial_state: <stateName>
  transitions: (<portTransition>({, <portTransition>}))
}

<portTransition> ::=
    <componentEvent>: <portPreconditions> → <portActions> → <portPostconditions>

<portActions> ::= (core.<command>({, core.<command>}))

<portDependencies> ::= (dependencies: <dependence>({<dependence>}))

<dependence> ::= <portName>.<stateName> → <portName>.<stateName>
```

Each component's behaviour specification is segmented into a set of *ports* i.e., points of interaction with the environment, each described by a state machine. A transition is labeled by a *component event* name, termed the *trigger event*, meaning that its firing is triggered by the occurrence of such event. *Preconditions* are constraints on the port's state; for a transition to fire, the relevant event must occur *and* its preconditions must be satisfied. A port is said to be `ready for event e` if it is in a state which satisfies the preconditions of at least one transition with `e` as the trigger event. *Postconditions* describe the state of the port *after* the firing of the transition. The *action* clause specifies actions that are taken by the component as a consequence of the firing of the transition. In this context, actions are invocations of commands on the internal component's core. Such *core actions* model activities performed in the small. The execution of a core action is the only link between PIL and PIS level i.e., from the architectural component (whose behaviour is that modeled by the component specification) and the component's core. The *dependencies* section describes any interdependencies existing

[2] Nonterminals are written in plain text, terminals are in `typewriter`.

between the state of different ports, with the intent of *gluing* the port specifications together. Each dependence `p.s → q.t` states that whenever port `p` is in state `s`, port `q` is forcibly moved onto state `t`.

Connectors are the locus of cooperation between components, each playing a certain role in the cooperation. Connectors' transitions define *cooperation events* which may correspond to a sequence of component events involving the components playing the role supported by the connector. The syntax for describing connectors is very close to that used for components:

```
<connectorSpecification> ::=
  connector <connectorName> {
    roles: <roleName> {,<roleName>}
    initial_state: {<stateName>}
    transitions: <connectorTransition> {,<connectorTransition>}
  }

<connectorTransition> ::=
   <cooperationEvent>: <connectorPreconditions> → <connectorActions>
   → <connectorPostconditions>

<connectorActions> ::= (<roleName>.<componentEvent>
  ({, <roleName>.<componentEvent>}))
```

The connector's behaviour is described as a state machine. *Roles* can be seen as formal parameters; the connector can manage the cooperation among any set of entities which can play the listed roles. In actual systems, roles will be played by components' ports. Transitions describe the connector's behaviour. Each transition is labeled by a *cooperation event* name, termed the trigger event. As for port transitions, a connector transition fires if the relevant cooperation event occurs *and* the preconditions are satisfied, and pre- and postconditions model the machine's state before and after the transition. As for ports, a connector is `ready for cooperation event e` iff its state satisfies at least one transition whose trigger event is `e`. Connector actions represent the fact that the firing of a connector's transition recursively fires transitions in the entities playing the roles by triggering some events. For any action `r.e` in the action clause, an implicit precondition must be added to the explicit preconditions, namely that the entity playing role `r` is ready for event `e`.

Topology. The *program-in-the-large* proper comprises a description of the overall system's topology and strategy. The system's topology defines which components and connectors comprise the system and how they are attached to each other (i.e., which component ports play which connector roles). Components' and connectors' specifications can be regarded as subspecifications of the topology, describing in more detail each of the entities referred to in the topology itself. A topology is described in a rather straightforward way as follows:

```
<topologySpecification> ::=
```

```
topology <topologyName> {
  components: {<componentName>}
  connectors: {<connectorName>}
  attachments:
    {<componentName>.<portName> plays <connectorName>.<roleName>}
}
```

In the APIL interpretation model, the system's topology is *actuated* by a *topology actuator* (TA in the following). Given that one of the overall aims of our approach is to keep the architecture of the system explicit, it would clearly be impossible to hard code links between components and connectors inside them. Hence, the role of the TA is twofold. On the one hand, it realizes the topology by instantiating the appropriate components and connectors. On the other hand, it keeps track of the interconnections between ports and roles dispatching events between component and connectors (i.e., it dispatches component events generated by connectors to the appropriate component(s)).

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) which 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, such as event-driven hard real-time, hard real-time ART (i.e., time-driven), concurrent, priority-based, and so on. The rest of this section is devoted to illustrating the syntax and semantics of these rules, along with the necessary definitions. 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. Note that, due to the layered approach we adopted, the SA has no knowledge whatsoever of components, as it can only see connectors' states. Thus, we have:

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

The state of a connector can be formally 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 "`time = 3.00pm`", where `time` can be regarded as a predefined variable that refers to the current date and time. 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. For instance, the SA may need to be multi-threaded, in order to execute rules that overlap in time.

Since the SA only knows about connectors and, possibly, time, the only entity on which it can perform an action is a connector. Thus, the *actions* section of rules is but 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:

❶ 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;
❷ the SA simply ignores the problem and leaves every such issue with the strategist (see section 3.2).

Following solution ❶ above, it is the task of the SA to ensure system consistency, while adopting approach ❷, 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 organizing 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 according to the will of the SA. The only constraint that the set of rules must respect is that the rules it contains must be non-conflicting; apart from that, every decision is up to the SA.

The overall definition of the strategy is as follows:

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

A *system* is defined by a topology and a strategy:

```
<systemSpecification> ::=
  system <systemName> {
    topology: <topologyName>
    strategy: <strategyName>
  }
```

The APIL Virtual Machine. The virtual machine of the APIL programming-in-the-large 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. In the general context of APIL (i.e., if we abstract from its relevance for AR), it is mainly intended to support reuse of architectural organizations (i.e., programs-in-the-large) across different platforms (where the basic architectural primitives may have completely different implementations e.g., CORBA, Java, JavaBeans, and so on).

3.2 Architectural Meta-Layer

Each architectural meta-layer is a portion of an architectural reflective system devoted to observe and manipulate the software architecture of the underlying layers.

We can consider each meta-layer (see Fig. 1) as a shell which wraps the underlying system. The domain of the meta-entities working at a given meta-layer is defined inductively as follows:

Base step: the domain of the architectural meta-entities working in the first meta-layer is the software architecture of the base-layer;

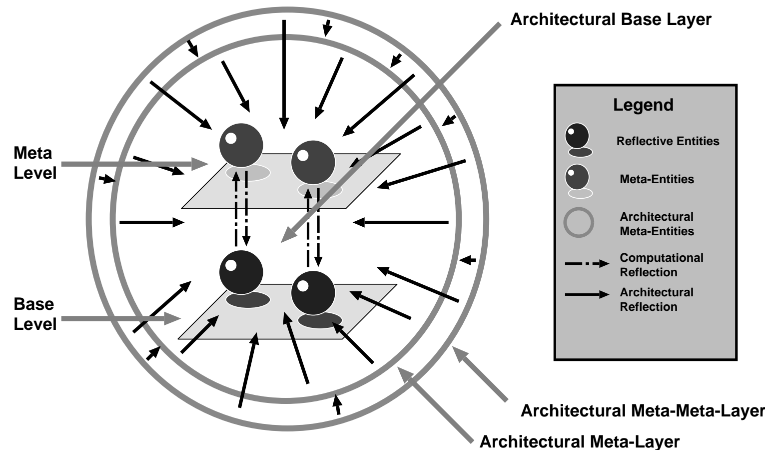

Fig. 1. Base- and meta-layers structure

Inductive step: the domain of the architectural meta-entities working in the n-th meta-layer is the software architecture of the base-layer, and of the first meta-layer, . . . , and of the (n-1)-th meta-layer i.e., the software architecture of the system composed by these layers.

As discussed so far, in this work we only consider two of the numerous aspects of software architecture: topology and strategy. In order to simplify the architectural reflective model, strategical and topological reflection are charged to two distinct architectural meta-entities, termed respectively *strategist*, and *topologist*.

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 structure of the considered systems, in order to access and to manipulate architectural information, the architectural meta-entities need only interact with the actuators of the underlying layer.

Topologist and strategist could be implemented as a single entity for the sake of efficiency, but a separated implementation enhances chances for design reuse.

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 initialization activities, and so on). In this case, topologist and strategist must exist before and|or after the creation and|or destruction of the system.

Figure 2 represents an architectural reflective system. In the base-layer are components (gray spheres), connectors (little black spheres connected by arrows, the little spheres represent component ports), and actuators (big black spheres) that shape and activate the system. Also observe that each meta-layer has its own actuators ruling over

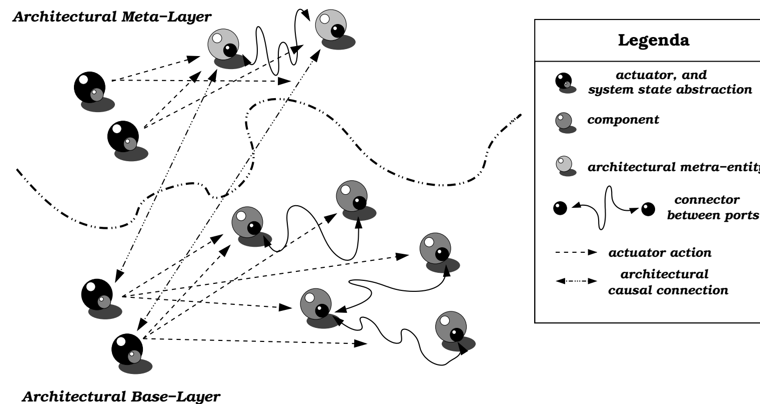

Fig. 2. Architectural reflective system structure

the meta-layer's architecture. The architectural meta-entities are represented by light gray 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.

Meta-Layer Architectural Modeling Let us now consider how the first meta-layer is described at the architectural level (of course the same considerations apply to any meta-layer). The idea is that the meta-layer is again a system of interacting entities and hence can be modeled with the same syntax used for the base-layer. It comprises two components (topologist and strategist), respectively in charge of manipulating the topology and the strategy of the base-layer. Of course, just as for the APIL description of base-layer components we omit details on their semantics (i.e., their operation on their domain), likewise when describing topologist and strategist in APIL language we will not detail their actual operation on the base-layer. Architectural causal connection from meta- to base-layer (downwards architectural connection, or reflection) is thus hidden at the architectural level; reflective actions performed by the topologist|strategist on the base-layer architecture will be modeled by core actions. Such core actions will involve invoking architectural modification primitives (discussed in section 4). A component (such as topologist and strategist) whose core action involves modification of the underlying layer's architecture is labeled with the keyword **meta-component** instead of **component**. In the meta-layer architectural model we adopted, the strategist is the only meta-component accessing the strategy actuator and the topologist is the only meta-component accessing the topology actuator.

For what concerns the upwards architectural connection, or reification (making the base-layer architecture observable at the first meta-layer), saying that the base-layer architecture is observable at the first meta-layer actually means that its state can influence the strategist's and topologist's operation. In other words, transitions and rules at the meta-layer can have pre- or postconditions including predicates on the current state of the base-layer's architecture. To model this fact in the APIL language we use the key-

word **reify** in both the topology and the strategy. If a topology reifies a system, this means that predicates on this system's state can occur in the components' and connectors' behaviour specification. Likewise, if a strategy reifies a system, this means that predicates on this system's state can occur in rules.

As another extension to the basic APIL model required to describe the first meta-layer, we also introduce constructs for modeling the fact that the state of the actuators of the base-layer is observable by the meta-layer. This is analogous to reifying the state of the virtual machine of a language in classic reflection. In particular, to model information about the strategy actuator's state, we introduce *meta-rules* at the first meta-layer. Meta-rules are labeled with the keyword **meta-rule** instead of **rule**, and they include predicates about the firing (and time-related issues such as time of firing, duration, and so on) of the base-layer rules. Notice that meta-rules are used in junction with the `reify` keyword (the subsystem must be observed in order for meta-rules to be allowed on the subsystem's rules). Expressions on rules that can be included in meta-rules can have the following forms:

```
⟨ruleName⟩.enabled    //boolean expression: this rule is enabled
⟨ruleName⟩.disabled   //boolean expression: this rule is disabled
⟨ruleName⟩.fired      //boolean expression: this rule has fired in the past
⟨ruleName⟩.lastFired  //boolean expression: this rule was the last one to be fired
⟨ruleName⟩.time       //time expression: this rule fired at time x
⟨ruleName⟩.duration   //time expression: this rule took x time to complete the
                      //last time it fired
⟨ruleName⟩.failure    //boolean expression: the last firing of this rule
                      //did not work to meet the rule's postconditions
```

3.3 Architectural and Computational Reflection: the Dualism

Under several respects, architectural reflection is a transposition in the large of computational reflection. Starting from the fact that topology and strategy represent, respectively, the system's structure and behaviour in the large, two statements follow:

❶ topological reflection can be compared to structural reflection, as both act on the structure of the entities they manipulate, the former operating in the large i.e., on the topology of the system, the latter in the small i.e., on the code of a single entity;
❷ strategic reflection can be compared to computational reflection, as both act on the behaviour of the entities they manipulate, the former in the large, by observing and modifying the strategy of the system, the latter in the small, by observing and altering the computational flow of a single entity;

In both approaches the two reflective aspects (structural|topological, and behavioural|strategical reflection) are charged to different entities: the topologist, and the strategist in the architectural approach; the object's class, and specific meta-entities (meta-classes [6], meta-objects [13], channels [2] or messages [10]) in the classic approach.

Both kinds of reflection can be viewed in a compositional way. In classic reflection, the meta-entities working in a generic meta-level of the reflective tower observe and manipulate the meta-entities of the underlying level, which in turn observe and manipulate the entities (base- or meta-) of the underlying level and so on down to the

base-level. Analogously, as stated previously, the architectural meta-entities working in a certain layer of the architectural reflective tower observe and manipulate the software architecture of the whole underlying system.

Of course the two approaches are orthogonal, because the software architecture of a system includes the components and the connectors related to meta-entities (if any) employed in the system. Therefore, as shown in Fig. 1, the reflective tower is a part of the domain of the first meta-layer of the architectural reflective tower.

4 Architectural Causal Connection

In classic reflection, a reflective system must keep, at the $(n+1)$-th level, an appropriate representation of the nth level and must be able to *reify* any changes in the level below into its representation and to *reflect* any change in that representation into the base-level. This process, called causal connection, is at the heart of a reflective system, as it allows to manipulate the *description* of a system, rather than the system itself.

Architectural reflection is no exception, in that topology and strategy can be 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. As discussed in the previous section, the link between the base-layer and the meta-layer is represented by the `reify` keyword for reflection and by core actions in topologist and strategist for reification; both these features are implemented based on a set of primitives, which will be set forth in the rest of this section.

4.1 Topological Primitives

As mentioned earlier, the tasks of the topologist are:

- to keep a representation of the system topology;
- to update the representation in order to reify any changes that should occur in the topology;
- to ensure that any changes in its representation are reflected in the base-layer (i.e., in the topology itself).

Thus, the *domain* of the topologist is the topology, and the base-layer entity it cooperates with is the topology actuator. Therefore, we will not be concerned here with how the topologist is made or how it works inside, nor with how it can be programmed to accomplish its tasks, but only with how it interacts with the topology actuator in order to observe and|or manipulate the topology. Therefore, the TA acts as a bridge between the topologist and the APIL virtual machine; as such, it exports to the topologist the appropriate directives for this to be able to accomplish its topology tasks.

In the context given above, the topologist, in order to reify the current topology of the system, relies on the TA, rather than knowing the single components and connectors directly. Every change in the system topology is known by the topologist through the TA's representation.

As far as reflection is concerned, the commands that the topologist can issue to the TA can be broadly partitioned into three categories:

❶ creation of new types of entities (`define` primitive);
❷ creation of new entities of an existing type (`instantiate` primitive);
❸ destruction of entities (`destroy` primitive).

Each of the above commands applies to all kinds of entities, if with some limitations, as we will see below. Thus, the directives exported by the TA to the topologist are the following:

- `define{Component|Connector}`: defines a new type of entity;
- `instantiate{Component|Connector|Attachment}`: creates a new entity of an existing type[3];
- `destroy{Component|Connector|Attachment}`: destroys an existing entity;

4.2 Strategical Primitives

As explained above, the system evolution is governed by the strategy actuator (SA), which accomplishes its task by executing a set of rules; in other words, the system behaviour is governed by those 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:

❶ knowing the rules;
❷ observing the state of the SA.

As for point ❶, it is clear that the strategist knows the rules, as it makes them. Point ❷ simply means that a `getState` primitive (or some similar mechanism) must be exported by the SA to the strategist. For example if, after a rule is fired, the postconditions are not respected, the SA enters an error state; this state is observed by the strategist that can then take the appropriate measures (see section 5 for a complete example).

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, as follows:

- `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. Obviously, the strongest requirement that a rule can only be fired when its preconditions are met, still holds; this means that this directive has only effect if and when the preconditions are satisfied.

Note that the above set of primitives is to be intended as a minimal set, which can be enriched at will according to particular requirements.

[3] Note that since attachments are all of the same type, there is no `defineAttachment` directive.

5 Advantages

The most important advantage introduced by our approach is represented by the possibility to dynamically reconfigure a running system. Given a running architectural reflective system, the architectural meta-entities have the power to modify the current system's topology|strategy according to their programming. When a running system has the hooks (actuators, and the other entities needed for APIL [4]) required to bind the meta- to the base-layer, it is extendible without stopping it, thanks to the properties of transparency and separation of concerns of architectural reflection.

5.1 An Example

We will now give an example illustrating the idea of AR and the advantages it brings. Since we are interested in the architectural structure of systems and reflection on such architecture, in the example we will not get into the details of the components' programs-in-the-small (actual computation).

The Scenario and the Base-Layer. Let us consider a non-stopping distributed system, periodically receiving a large input set, processing it, and generating a new output set. The system considered is a subsystem of a more complex system, and inputs and outputs come from, or go to external subsystems (which we don't detail). The system performs a task based on the data-parallelism paradigm, and is composed by several identical components; each component works on an equal slice of the original input set. Those components are described as follows:

```
component unit {
  initial_state: waiting_for_input
  port input {
    initial_state: ready_for_input
    transitions:
      inValues(real *data):
        ready_for_input → core.evaluate(real *data) → idle
  }
  port output {
    initial_state: idle
    transitions:
      outValues(real *data):
        ready_for_output → core.retrieve(real *data) → idle
  }
  dependencies:
    input.idle → output.ready_for_output
    output.idle → input.ready_for_input
}
```

Units have two ports, the former for receiving the inputs to elaborate, the latter for dispatching the elaborated outputs. The input and output entities are modeled as components:

```
component input {                          component output {
  port input {                               port output {
    initial_state: idle                        initial_state: idle
    transitions:                               transitions:
      getValues(real *data):                     putValues(real *data):
        idle → core.get(real *data) → idle           idle → core.put(real *data) → idle
  }                                            }
}                                            }
```

In this system there are two kind of connectors, the former handles the receiving of data, while the latter handles the dispatching of the produced data.

```
connector inputConnector {                 connector outputConnector {
  roles: in, unit                            roles: out, unit
  initial_state: empty                       initial_state: empty
  transitions:                               transitions:
    transfer(real *d):                         transfer(real *d):
      empty → in.getValues(real *d),             empty → unit.outValues(real *d),
              unit.inValues(real *d) → empty             out.putValues(real *d) → empty
}                                          }
```

They connect, respectively, a generic component with the input, or the output entities.

Topology and Strategy of this system are parametric (the value of n in the specification) on the number of the components unit belonging to the system and consequently of the number of connectors connecting them[4].

```
topology topology_base-layer {
  components:
    IN: input; OUT: output;
    unit{1...n}: unit;
  connectors:
    INUnit{1...n}: inputConnector;
    Unit{1...n}OUT: outputConnector;
  attachments:
    IN.input plays INUnit{1...n}.in;
    OUT.output plays Unit{1...n}OUT.out;
    unit{1...n}.input plays INUnit{1...n}.unit;
    unit{1...n}.output plays Unit{1...n}OUT.unit;
}
```

The strategy is very simple, and consists of only one rule, which makes advance the system when an input is ready to be elaborated

[4] when we write: component_name.port_name **plays** connector_name{1...n}.rule_name; and component_name{1...n}.port_name **plays** connector_name{1...n}.rule_name; we, respectively, mean: $\forall i \in \{1,...,n\}$.(component_name.port_name **plays** connector_name$_i$.rule_name); and $\forall i \in \{1,...,n\}$.(component_name$_i$.port_name **plays** connector_name$_i$.rule_name);

```
strategy strategy_base-layer {
  rule advance{1...n} {
    → INUnit{1...n}.transfer(), Unit{1...n}OUT.transfer() →
  }
}
```

Finally, the system is represented by the composition of topology and strategy.

```
system base-layer {
  topology: topology_base-layer;
  strategy: strategy_base-layer;
}
```

The Meta-Layer. Let us now assume that a dynamic behaviour must be added to the system, so that whenever a step takes too long, the number of units is doubled to speed the computation up. In order to realize such a behaviour, the designer must add a meta-layer hooking the subsystem up and in which topologist and strategist cooperate for doubling the number of the computational units of the base-layer. As explained above, topologist and strategist are special components introduced by the keyword **meta-component**, and in this case are described as follows:

```
meta-component topologist {
  port double {
    initial_state: idle
    transitions:
      doubleComponents():
        idle → core.istantiateComponents(
                   components:
                     unit{n+1...2n}: unit;),
                 core.istantiateConnectors(
                   connectors:
                     INUnit{n+1...2n}: inputConnector;
                     Unit{n+1...2n}OUT: outputConnector;),
                 core.istantiateAttachments(
                   attachments:
                     IN.input plays INUnit{n+1...2n}.in;
                     OUT.output plays Unit{n+1...2n}OUT.out;
                     unit{n+1...2n}.input plays INUnit{n+1...2n}.unit;
                     unit{n+1...2n}.output plays Unit{n+1...2n}OUT.unit;)
               → idle
  }
}

meta-component strategist
  port double {
    initial_state: idle
    transitions:
```

```
      addRules():
        idle → core.addRules(
                    rule advance{n+1...2n} {
                      → INUnit{n+1...2n}.transfer(),
                        OUT{n+1...2n}unit.transfer()
                      → )
                → idle
    }
  }
```

Strategist and topologist reconfigure the system by adding units and connectors to the base-layer, by attaching ports and connectors properly, and by adding the necessary rules to coordinate the new elements' work. As explained in section 3, they do so by invoking architectural modification primitives on the actuators via core actions.

```
connector doublingCoordinator {
  roles: topologyDoubler, strategyDoubler
  initial_state: idle
  transitions:
    coordinateDoubling():
      idle → topologyDoubler.doubleComponents(),
             strategyDoubler.addRule() → idle,
}
```

Topologist and strategist are of course coordinated by a connector, which guarantees that topology and strategy are modified accordingly.

```
topology topology_meta-layer {
  reify: base-layer;
  meta-components:
    t: topologist;
    s: strategist;
  connectors:
    dc: doublingCoordinator
  attachments:
    t.double plays dc.topologyDoubler;
    s.double plays dc.strategyDoubler;
}

strategy strategy_meta-layer {
  reify: base-layer;
  meta-rule doubleArchitecture {
    advance().duration > "10ms" → dc.coordinateDoubling(), →
}
```

The strategy of this layer is very simple and consists of triggering the connector when the computation of the base-layer takes too long. The fact that too much time has elapsed is expressed by the precondition of the `doubleArchitecture` meta rule.

6 Related Work

The idea of making architectures explicit as exposed in this paper has been influenced by several contributions in the software engineering literature, which obviously include the very idea of *programming-in-the-large* introduced by DeRemer and Kron. Mary Shaw, Robert Allen, David Garlan and other colleagues at CMU wrote several papers, now collected in a book [19], proposing notations and models for explicitating architectural designs. Their work is primarily aimed at *specifying* software architectures, usually with analytical purposes, but with no notion of an *executable* architectural description. As such, their work is similar in purpose to Helm's formal contracts [12], except that it addresses general rather than object-oriented architectures.

David Luckham from Stanford [14] proposes an executable architecture description language termed Rapide. As a main difference to our approach, execution of a Rapide architectural description is to be regarded as a *simulation* of the behaviour the system *will* have once implemented (with conventional means).

Separating architectural information in general from application-dependent information is one of the possible application of Object-Oriented frameworks [18]. While object-oriented frameworks address this problem in terms of a static separation achieved through inheritance (whereby an architectural design is represented by a set of abstract classes from which concrete component classes are derived, which ultimately embed, through inheritance, both architectural and application functionality), our work deals with maintaining such separation at run-time.

Several proposals have been made in the software engineering literature aimed at localizing information about the cooperation of components in dedicated run-time entities (like our connector) and insulating components from such information. Pintado's *gluons* [17] are intended to be the locus of cooperation patterns definition, but in his model components *are* aware of the gluon they interact with and they are actually designed explicitly for being connected through a specific gluon (protocol-centered design). A similar consideration holds for Aksit et al.'s composition-filter approach [1]. Sullivan's *mediator* concept [21] supports both localization of cooperation patterns and cooperation-transparent components. As a major difference to our connectors, mediators do not *control* components but they only enforce state interdependencies among them.

The closest approach to APIL is that of Stéphane Ducasse and Tamar Richner, who propose introducing connectors as run-time entities with much the same purpose considered here (making architectural designs explicit in implemented systems) in the context of an extended object model termed FLO [9]. FLO's connector model is very rich and interesting, and has several similarities to APIL's connector. Nevertheless, FLO is based on a simpler component model which does not include a notion of a port or a behavioural component specification.

While the mentioned efforts are close to APIL in several respects, none of them addresses dynamical modification of the architecture as a reflection problem; dynamical modifications are at best regarded as destruction and creation of connectors (gluons, mediators, etc.), but no clean conceptual model is provided for ruling over such events. Moreover, none of them includes the idea of maintaining at run-time a (logically) centralized description of the system's architecture (our program-in-the-large).

With respect to the three levels of system description outlined in section 2, all these efforts reach at most level two (description of cooperation patterns), and none addresses level three (description of the whole system as a composition of cooperation patterns). For this reason, while AR could be based on several of these models (particularly FLO), APIL seems most convenient to that aim.

7 Conclusions

This paper presents 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. Since dynamic reconfiguration and other self-management activities are conceived as operations performed by a system on its own architecture, the approach is strongly related to the idea of localizing and explicitating architectural designs in implemented systems, as realized by Architectural Programming-in-the-Large. The ideas presented in this paper formalize those presented in a previous paper on the same topic [5], explain the relationship between AR and APIL, and between AR and classic reflection.

At present, a prototype APIL environment is under development. We believe that making AR feasible is one of the major advantages of the APIL approach itself, so that integrating architectural reflective capabilities in this environment will be a prominent goal in the near future. On a more conceptual level, the presented work can be extended by including other aspects of software architecture (distribution, performance, hierarchical organization in subarchitectures) in the APIL model and extending AR to include reflection on these new aspects.

References

[1] Mehmet Akşit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of Object-Based Distributed Programming (ECOOP'94 Workshop)*, Lecture Notes in Computer Science 791, pages 152–184. Springer-Verlag, July 1994.

[2] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In Roy Jenevein and Mohammad S. Obaidat, editors, *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.

[3] 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.

[4] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco 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.

[5] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco 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, Italia, on 8th-11th March 1998. IEEE.

[6] Pierre Cointe. MetaClasses are first class objects: the ObjVLisp model. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22(10) of *Sigplan Notices*, Orlando, Florida, USA, October 1987. ACM.

[7] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, Montréal, Canada, August 1995.

[8] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2:80–86, June 1976.

[9] Stéphane Ducasse and Tamar Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of ESEC'97*, LNCS 1301, pages 483–500. Springer-Verlag, 1997.

[10] Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[11] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of XVII ICSE*. IEEE, April 1995.

[12] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Compositions in Object-Oriented Systems. In *Proceedings of OOPSLA/ECOOP'90*, pages 169–180. ACM, 1990.

[13] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[14] David C. Luckham, Larry M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, SE-21:336–355, April 1995. Special Issue on Software Architecture.

[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] Gail C. Murphy. Architecture for Evolution. In Alexander L. Wolf, Anthony Finkelstein, George Spanoudakis, and Laura Vidal, editors, *Proceedings of 2nd International Software Architecture Workshop (ISAW'96)*, pages 83–86, San Francisco, CA, USA, October 1996. ACM.

[17] Xavier Pintado. Gluons: A Support for Software Component Cooperation. In *Proceedings of ISOTAS'93*, LNCS 742, pages 43–60. Springer-Verlag, 1993.

[18] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[19] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ 07458, 1996.

[20] Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.

[21] Kevin J. Sullivan, Ira J. Kalet, and David Notkin. Evaluating the Mediator Method: Prism as a Case Study. *IEEE Transactions on Software Engineering*, 22(8):563–579, August 1996.