Architectural Reflection:
Bridging the Gap Between a Running System and its Architectural Specification

W. Cazzola A. Savigni A. Sosio F. Tisato
DSI-University of Milano, Via Comelico 39-41, 20135 Milano, Italy
Tel: (+39) 2 5500 62 {98|98|98|31} Fax: (+39) 2 5500 6373
E-mails: {cazzola|savigni|sosio|tisato}@dsi.unimi.it

Abstract—As the size and complexity of software systems increase, a relevant part of the system overall functionality shifts from the applicative domain to run-time system management activities, i.e., management activities which cannot be performed off-line. These range from monitoring to dynamic reconfiguration and, for non-stopping systems, also include evolution, i.e., addition or replacement of components or entire subsystems. In current practice, run-time system management is impeded by the fact that the knowledge of the overall structure and functioning of the system (i.e., its software architecture) is confined in design specification documents, while it is only implicit in running systems. In this paper we introduce, provide rationale for, and briefly demonstrate an approach to system management where the system maintains, and operates on, an architectural description of itself. This description is *causally connected* to the system's concrete structure and state, i.e., any change of the system architecture affects the description, and vice versa. This model can be said to extend the principles of computational reflection from the realm of programming-in-the-small to that of programming-in-the-large.

Keywords: Reflection, Layered Systems, Software Architecture, Software Components, System Management.

I. Introduction

As the size and complexity of software systems increase, a relevant part of the system overall functionality shifts from the application domain to run-time system management activities such as dynamic reconfiguration, on-line monitoring, fault detection and recovery, security, and so on. In the commercial arena, products are widely available based on SNMP, CMIP and other protocols [1] which address system management for both hardware and software resources.

In our view, current practice in system management would benefit from considering the software architecture [2] of systems, that is, the rules governing the assembly and interaction of their constituting parts, e.g., system topology or cooperation protocols. While architectural concepts are widely appreciated by software engineers as a means to describe software systems at a higher level than that supported by conventional languages and development tools, and are ubiquitous in design specifications, exploiting such concepts in run-time system management is impeded by the fact that in current practice architectural information is not explicit in running systems, but implicit and scattered in the code of components.

As an example, consider a distributed system whose components interact according to a certain protocol. While architectural specifications may include a description of the protocol in its entirety, at run-time such a description is lacking, and each component only knows the part of the protocol it is concerned with. Now consider the following system management operations:

1. inserting a new component into the system, which entails verifying its compatibility with the protocol and, if insertion proves possible, determine *when* it may take place;
2. dynamically modifying the protocol.

Verifying the compatibility of the new component with the existing protocol requires knowledge of the overall protocol. Modifying the protocol entails modifying the various component which participate in it. Unfortunately, since at run-time the protocol is implicit, it is hardly possible to reason about it (e.g., to check components' compatibility) or modify it. In general, in order to solve this kind of problems, the following must hold:

1. a run-time description of the overall architecture of the system must exist;
2. a run-time description must also exist of the *state* of the system, conveying all information relevant to operate on such architecture (e.g., to insert a new component);
3. these descriptions must be maintained coherent with the concrete system itself, i.e., it must be possible to observe and manipulate the system in terms of such descriptions.

In our example, we need an entity that both knows the protocol and tracks the state of the interaction, being able to manipulate such information in such a way that any change made to it be propagated to the structure and behaviour of the system.

Using a concept from the discipline of *reflection*, the three points above can be restated by saying that a description of the software architecture and overall state of the system must exist which is *causally connected* to the system's architecture and state. In this paper, we introduce *architectural reflection*, an approach to designing software systems that applies reflective concepts to *software architecture* in order to fulfill this goal.

The outline of the paper is as follows. Section II. introduces some preliminary concepts in both reflection and software architecture. Section III. defines architectural reflection, including a comparison with computational reflection and a discussion of its possible applications within a software system's lifecycle. Section IV. presents an example, and section V. draws some conclusion and presents future work.

II. Preliminary Concepts

A. Computational Reflection

Computational reflection or reflection is defined as the activity performed by an agent when doing computations about itself [3]. Behavioural and structural reflection are reflection sub-branches which involve, respectively, agent computation and structure (for more details see [4]).

A reflective system is logically structured in two or more levels, constituting a *reflective tower*. Entities working in the base level, called base-entities or reflective entities, define the system basic 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), for details see [5].

B. Software Architecture

The *architecture* of a software system is the system's overall structure as an organized collection of interacting components. It is described by stating:

1. how the overall functionality of the system is partitioned into its constituting modules, or *components*;
2. how such modules interact and cooperate (i.e., what *connectors* exist between components [2]).

The description of components and connectors, in turn, omits internal details such as algorithms and data structures, while it conveys information related to system integration and coordinated behaviour, e.g., interaction protocols, obligations posed by a module on its environment, and so on. In other words, architectural descriptions express concepts belonging to the realm of *programming in the large* rather than that of *programming in the small* [6].

To describe architectures we adopt a model close to that of the Wright language [2]. Components are described by state machines whose state and behaviour is an abstraction of those of the *concrete components* (modules) which constitute the actual system. Examples of such *abstract states* could be "ready_to_send_a_message", "ready_to_receive_a_message", "faulty", and so on. The transitions are labeled by events which modify such a state (e.g., the sending of a message).

Connectors are also described by state machines that model *interactions* rather than the behaviour of modules in isolation. The connector in figure 1 describes a simple client|server interaction.

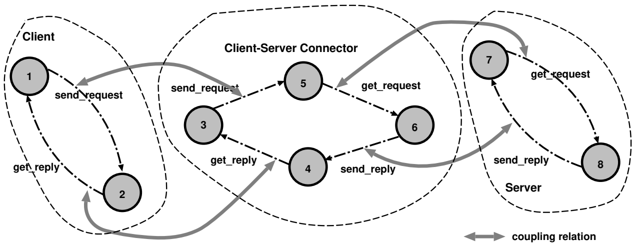

Fig. 1. **State machine and transition coupling example**

The transitions of a connector between two or more components can be *coupled* to transitions in such components; for the sake of simplicity, we represent transition coupling by labeling coupled transitions with the same event name, as in figure 1. The dynamic behaviour of a system of components and connectors is modeled in the usual way by the firing of *ready* transitions and the rule that two or more coupled transitions are deemed ready only if and when they are *both* ready (as considered in isolation); see [7].

Based on this model, we can isolate two orthogonal concepts in architectural description: *topology* and *strategy*.

The topology (or configuration) of a system is defined by the collection of its components and the connectors between them (all described by state machines). The system's strategy is defined by the set of rules governing the order and time of firing of ready transitions. Concurrent execution, as assumed by most state machine models, is only one such strategy. Other examples include sequential and temporized (real-time) plans.

We also define the system's *computation in the large* as the firing of ready transitions and its *computation in the small* as the execution of code inside concrete components. The former is an abstraction of the latter. For example, the firing of transition "send_request" may involve the execution of several operations in the small, e.g., parameter marshaling, writing in output buffers, and so on.

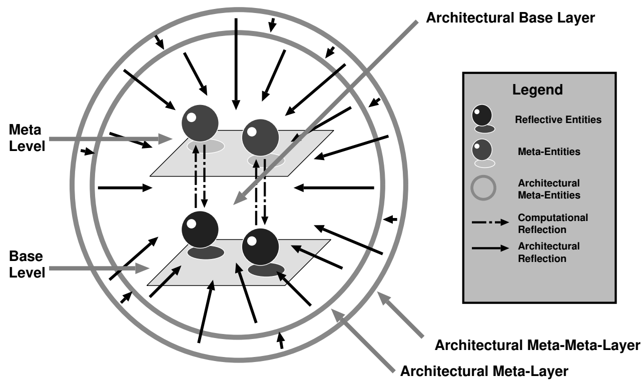

Fig. 2. **Classic and Architectural Reflective Tower**

III. Architectural Reflection

def. Architectural Reflection is the computation performed by a system about its own software architecture.

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 several (potentially infinite) layers, called *architectural layers*, costituting 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. Such layers are causally connected, i.e., every change made to these data structures reflects on the system architecture, and vice versa.
Every architectural layer reifies a description of the software architecture of the underlying system that includes the architectural meta-entities operating in the layers below, as these architectural meta-entities, too, are part of the software architecture of the system that the entities in the layer above manipulate. Note that we use the terms "level" and "layer" to distinguish, respectively, the levels of the reflective tower from those of the architectural one.

Also note that the architectural reflective tower is distinct from the computational reflective tower as classic meta-entities are also described by the software architecture of the system, and are therefore reified by architectural reflection. In other words, recalling what Maes stated in [3] about computational domains, the meta-entities and the architectural meta-entities operate on different domains and particularly the domain on which the architectural meta-entities operate includes the meta-entities (see figure 2). Note that we use the terms "level" and "layer" to distinguish, respectively, the levels of the reflective tower from those of the architectural one.

Every architectural layer operates on the architecture of the layer below being unaware of the presence and behaviour of the layers above (i.e., the property of transparency typical of the computational reflection still holds for architectural one). In general, every layer of the tower accomplishes system control tasks and adds, still on a system basis, new functionalities to the original system (separation of concerns).

Based on our definition of topology and strategy as orthogonal aspects of software architecture, we can further define the definition of architectural reflection by defining *topological and strategic reflection*.

def. 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.

def. 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 actions is changing priorities associated to transitions in a priority-based strategy.

As depicted in figure 3, 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. Likewise, strategic reflection

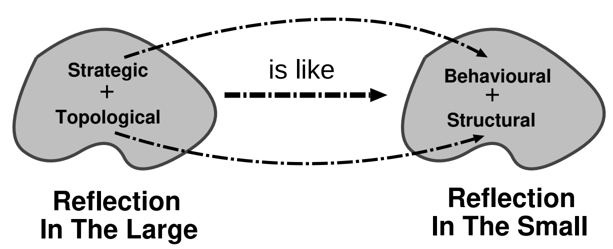

Fig. 3. **Reflection in the Large vs Reflection in the Small**

can be compared to behavioural 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.

Obviously, just as computational reflection requires ad-hoc languages and compilers, so architectural reflection would require suitable development tools which support building components and connectors that do not embed any assumption about the topology or the strategy of the system where they are employed, or, at least, provide hooks for architectural meta-entities to override such assumptions.

A. *Realizing Architectural Reflection: Who and How*

In the computational reflective approach, as supported by reflective object-oriented programming languages, structural and behavioural reflection are usually charged to different entities. In most models structural information is maintained by the reflective object class, as by definition this describes objects' structure. Computational reflection is performed by meta-entities which rely on structural information maintained in classes and reify only the information that may be different in different class instances (e.g., state). Meta-entities (be they meta-objects [8], channels [9] or messages [10]) perform meta-computation by trapping the normal computation of reflective objects; in other words, each base-entity action is trapped by a meta-entity, which performs a meta-computation either substituting or encapsulating the base-entity's action. This can be done either at run-time [11] or at compile-time [12].

The parallelism between computational and architectural reflection depicted in figure 3 can be extended to the realization model. Just as in the computational approach, structural and behavioural reflection are charged to different entities, so for architectural reflection there will be two architectural meta-entities per system, termed *topologist* and *strategist*, in charge of topological and strategic reflection respectively. The topologist maintains information about topology (components, connectors, and their state machines), 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.

As stated above, in order for the strategy to be subject to manipulation, the base-level system's strategy must lend itself to be substituted by the strategist, just as behavioural meta-entities encapsulate|substitute single actions. This implies that the strategist must also play the role of a transition scheduler *enforcing* the flow of control in the large, i.e., it must have operations for firing ready transitions.

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

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.

B. *Architectural Reflection Applied to System Lifecycle*

Architectural reflection holds the promise of providing benefits for all software lifecycle activities which relate to the overall system rather than its components considered in isolation. Such activities may belong to the system's *development, run-time management,* or *evolution*.

- <u>Development</u>. Architectural reflection can aid system development in several ways, thanks to the possibility of designing and developing architectural meta-entities independent of, and before, the architectural base-layer system. For example, architectural meta-entities could be used to simulate the system's computation in the large before developing the system itself (compare [13]); they could support testing and debugging at a higher level than that currently supported by programming environments (e.g., checking the system against deadlock or other undesiderable properties of the components' *integrated behaviour*). Moreover, reusing architectural meta-entities (and hence the architecture they embody) is a way to extend software reuse from code to design (see [14]).
- <u>Run-time system management</u>. As already mentioned in this paper, a whole range of run-time activities involve manipulating the architecture of a system and would benefit from a causally connected representation of such architecture as proposed by this paper. Examples include start-up of large component-based systems, dynamic reconfiguration, on-line monitoring, and fault detection and recovery.
- <u>Evolution</u>. While dynamic reconfiguration is an architectural modification which moves the system within a space of "configurations" foreseen by its designer (e.g., adding a new set of modules to cope with a new user in a mail system), the term *evolution* is used to indicate dramatic changes such as the introduction of radically new features or the integration with other systems. In a sense, evolution can be regarded as a "redesign" of the system architecture (including new kinds of components or recasting the old architecture as a sub-architecture of a larger system). Since, in our approach, the architectural structure of a system is encapsulated by its architectural meta-entities rather than being scattered over its components' code, evolution can effectively performed by substituting the architectural meta-entities themselves, as discussed in the next section.

IV. AN EXAMPLE: A DISTRIBUTED TRACK CONTROL SYSTEM

We now consider an example involving a distributed track control system (loosely inspired by [7]). The system controls a set of vehicles moving along a ring-shaped path formed by a set of tracks. It is a distributed system: there is a software component for each track. Vehicles are not modeled by software components: rather, each track component maintains

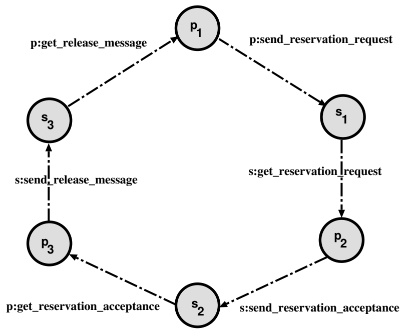

Fig. 4. State machines for connectors of the track control system

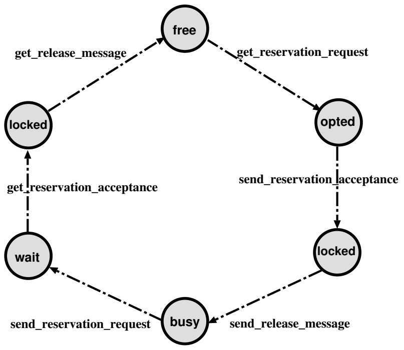

Fig. 5. State machines for components of the track control system

a flag indicating whether it is currently being run through by a vehicle. Each track is only connected with its neighbours in the ring.

The overall system protocol requires that tracks be reserved before a vehicle can enter them. Tracks are only released when the vehicle that entered them has successfully reached the next track.

To adhere to such protocol, each track component loops through the following steps:

1. waits for a reservation message from the previous track, and sends an acceptance message;
2. when the vehicle arrives, sends a release message to the previous track;
3. when the vehicle approaches the track's end, sends a reservation message to the next track;
4. waits for a reservation acceptance message, then allows the vehicle to go on to the next track;
5. waits for a release message from the next track.

The state machines for components and connectors are depicted in figure 4 and figure 5. For each connector transition t, p and s are used to distinguish in which of the two connected tracks the transition with that event name is coupled to t.
The overall strategy (not represented) is concurrent: ready transitions fire in an unspecified order. Nevertheless, as discussed in the following, the strategist must also be able to freeze undesiderable transitions in certain situations. Its internal representation of the strategy could be, say, the *set of enabled* transitions.

The operations provided by the topologist to manipulate the system's topology define the *class of topologies* the system can exhibit. In this specific example, we assume that the system can be reconfigured, say, to add tracks, as long as the following hold:

1. components and connectors behave as described by the state machines above;
2. tracks form a ring.

The topologist's operations assume and preserve such constraints. This means, for example, that insertion of a new track will be done by an operation that actually interposes it between two existing ones, reconfiguring connectors to preserve the ring structure.

In other words, the topologist's operations define a space of topologies and specify how the current topology can evolve within such space. In this example, the space of topologies is the space of all track rings.

Topological manipulations actually involve the strategist as well. The topologist handles the effective creation of components and connectors (via architectural causal connection), while the strategist guarantees that the reconfiguration performed by the topologist take place in a stable system.

In the case of the insertion of a new track (say, between track $\mathcal{A}$ and track $\mathcal{B}$), the strategist has to guarantee that the system state is a consistent state to perform that operation (for example, that $\mathcal{B}$ is not about to release $\mathcal{A}$, which would cause deadlock if the insertion was done before this actually happens). In order to do so, the strategist must check the components' and connectors' abstract states, and regulate the firing of transitions (for example, any transition that can lead the system in a non-consistent state while reconfiguration takes place must be freezed).

Modifications of the topology which would imply moving out of the space of topologies defined by the topologist (for example, switching to a star organization for track components) require substitution of the topologist itself, as it involves a change in the set of operations performed by the topologist. Obviously, this operation cannot be performed at the first architectural meta-layer, because architectural meta-entities are in the domain of architectural *meta-meta-entities*. In turn, such substitution must be an operation supported by the topologist of the meta-meta-layer, which hence defines a *class of classes of topologies*.

Just as described for the substitution of components for the base-layer system, substituting the topologist (considered as a component of the meta-meta-layer domain), may require cooperation between the meta-meta-layer's topologist and strategist. For example, the topologist should not be substituted while it is reconfiguring the application-level system; the meta-meta-layer strategist is in charge of guaranteeing this.

In our approach, as stated above, the strategist plays a double role: it *enforces* the strategy besides observing|manipulating it; in other words, it must explicitly activate transitions, or the flow of control in the large could not be subject to manipulation (see §A.).

As for the topologist, the strategist provides a set of operations for strategy manipulation|observation which depend on the specific strategy at hand. In a real-time system, for example, the strategist should provide operations to associate transitions to specific clock values or timelines.[1] The strategist's operations define a class of strategies and govern the evolution of the current strategies within such space. This space can be abandoned by the system's strategy only if the strategist itself is substituted by the architectural meta-meta-entities.

V. Conclusion and Future Work

This paper introduces the basic concepts and rationale for a novel approach to designing software systems which draws from recent results in the disciplines of software architecture and reflection. In this approach, the architectural description of a system, traditionally confined in design specification documents, is maintained at run-time to be inspected and manipulated by special entities (termed architectural meta-entities), and it is causally connected to the system's concrete structure. This provides a means to operate on complex, component-based systems at a higher level of abstraction than that supported by current practice techniques.

The approach holds the promise of providing benefits in all software engineering activities that entail observing and manipulating a software system as a whole rather than its components considered in isolation. All stages of a system's lifecycle, from design and development to evolution, comprise activities of this kind. Sample applications are: reusing architectural designs, on-line monitoring, and dynamic reconfiguration.

Our research on architectural reflection is currently addressing two main topics: the definition of a complete architectural model including issues outside those considered in this paper (such as distribution and performance); and the definition of an implementation model that does not impede system efficiency and that effectively supports reusing architectural meta-entities (hence architectures) in the design of new systems, most likely within an object-oriented perspective. This latter topic turns out to be closely related to the incremental definition of architectural styles through specialization [2] and the reuse of design patterns in object-oriented frameworks [14].

We are further analyzing the reflective aspect of architectural reflection, in particular the comparison with the classic approach. We are also working on semantics and implementation of the architectural causal connection and on a framework to introduce architectural reflection in a programming language.

[1] The set of operations should also allow the strategist to cooperate with the topologist during reconfiguration, as discussed above; for example, operations should be provided to disable dangerous transitions.

VI. References

[1] W. Stallings, *SNMP, SNMPV2 and RMON-Pratical Network Management*, Addison Wesley, 2nd edition, 1996.

[2] Mary Shaw and David Garlan, *Software Achitecture: perspectives on an emerging discipline*, Prentice Hall, Upper Saddle River, NJ 07458, 1996.

[3] Pattie Maes, "Concepts and experiments in computational reflection", in *proceedings of OOPSLA'87*. ACM, October 1987, number 22 in Sigplan Notices, pp. 147–156.

[4] François-Nicola Demers and Jacques Malenfant, "Reflection in logic, functional and object-oriented programming: a short comparative study", in *proceedings of the workshop section*, Montréal, Canada, August 1995, in IJCAI'95 (International Join Conference on AI).

[5] Robert J. Stroud, "Transparency and reflection in distributed systems", *ACM Operating System Review*, vol. 22, pp. 99–103, April 1992.

[6] Frank DeRemer and Hans H. Kron, "Programming-in-the-large versus programming-in-the-small", *Transactions on Software Engineering*, vol. SE-2, pp. 80–86, June 1976.

[7] S. Fischer, A. Scholz, and D. Taubner, "Verification in process algebra of distributed control of track vehicles - a case study.", in *4th International Workshop on Computer Aided Verification*, Montreal, Canada, July 1992, pp. 192–205, published in LNCS 663.

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

[9] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi, "Channel reification: A reflective model for distributed computation", in *proceedings of IEEE IPCCC'98*, Phoenix, Arizona, USA, on 16th-18th February 1998, IEEE.

[10] Jacques Ferber, "Computational reflection in class based object oriented languages", in *proceedings of OOPSLA'89*. ACM, October 1989, number 24 in Sigplan Notices, pp. 317–326.

[11] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa, "Object-oriented concurrent reflective languages can be implemented efficiently", in *proceedings of OOPSLA'92*, Vancouver, Canada, October 1992, ACM, number 22 in Sigplan Notices.

[12] Shigeru Chiba, "A meta-object protocol for C++", in *proceedings of OOPSLA'95*, Austin, Texas, October 1995, ACM, number 30 in Sigplan Notices, pp. 285–299.

[13] David C. Luckham, Larry M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann, "Specification and analysis of system architecture using Rapide", *TSE*, vol. SE-21, pp. 336–355, April 1995, Special Issue on Software Architecture.

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