Archetype: Addressing configuration issues in Software Architectures

Sergio Bandinelli Antonio Carzaniga Giovanni Vigna

CEFRIEL
via Emanueli, 15 20126 Milano, Italia
e-mail: {bandinelli, carzan, vigna}@mailer.cefreil.it

1 Motivation

Software Architecture is emerging as a major research focus in the software engineering field [9, 5]. Issues such as software architecture foundations, formal languages for describing architectures, identification of architectural styles, tools supporting analysis of architectures, etc., are currently receiving a great deal of attention from both industry and academia [3, 6].

The concept of software architecture is not new. Software designers have been working on architectures for many years. A bunch of consolidated well-engineered architectural solutions have already been produced and are now being used in everyday practice. Rather than inventing new solutions, research on software architectures is thus concentrating on providing a systematic organization of existing knowledge.

The increasing interest in software architectures is also due to the fact that software developers are nowadays more frequently faced with the problem of assembling existing "off-the-shelf" components, instead of programming new ones. As a consequence, overall feasibility and performance problems of a software system depend primary on architectural design decisions, rather than in finding and applying intelligent, efficient, or "tricky" algorithms.

At the architectural level, software designers abstract away from such systems aspects as functionality, user-interface, etc., and concentrate on the overall organization of the system, its components and their relationships. At this level of description, we find that configuration aspects are of major importance in an architectural description. In particular, an architectural description should not only be concerned with static connectivity issues, but should also include run-time behavior modeling in terms of dynamic component instantiations and configuration statements.

2 Current work on architectural description languages

Recently, several new approaches to software architectures have been proposed.

The approach proposed in [7] uses the CHAM (Chemical Abstract Machine) model, for the specification of software architectures. The CHAM model was originally introduced for representing concurrent computations. Intuitively, the state of a system represented by a CHAM is given as a chemical solution (represented by terms of an algebra) where the transformation process is operationally defined by the application of reactions (i.e., rules).

The CHAM approach to architecture description takes advantage of the neutrality, naturality, and expressiveness of the CHAM model, and of the possibility to formally prove relevant properties of the modeled architecture. The CHAM model is very flexible, but suffers from the fact that the number of solution molecules may grow considerably reducing the model understandability.

WRIGHT [1] was specifically designed for describing software architectures. WRIGHT distinguishes *component types* and *connector types* from their instances. Each component and connector instance corresponds to an actual entity in the architecture. Instances may be combined by attaching component instance ports to connector instance roles.

Component and connector behaviors are defined as interacting protocols using a simplified version of CSP. WRIGHT introduces the concept of compatibility of a port with a role and provides automatic support for compatibility checking. This allows the software architect to perform static checking on the architecture description.

UniCon (language for Universal Connector Support) [11, 10] provides a pragmatic approach. One of its main goals is to gain experience with the practical details of software architectures description. In UniCon, components and connectors are defined by providing an interface and an implementation.

The interface of a component includes a component type and a list of features to characterize the component, although they do not completely specify its behavior. The implementation of a component may provide directly the component's code or the description of the component in term of its parts. The parts of a component are instantiations of other components and connectors, appropriately linked to implement the component functionality.

Conic[8] is a programming environment and a run-time support for building, executing and managing distributed systems. Many of its features are also suitable for describing software architectures. In particular, a Conic system is given as collection of interconnected logical nodes. Each logical node (usually associated with a machine) is composed of many several modules organized in a hierarchical structure. A module is a closed independent set of concurrent sequential tasks. Conic provides two distinct languages: the *programming language* (somehow similar to Ada) is used to implement one single module. The *configuration language* provides primitives for creating modules and group instances, connecting their ports to form bigger groups and assigning groups to machines. The configuration language is used also interactively to manage run-time configuration, e.g., new logical nodes can be created, they can be connected to other nodes, links can be created and closed, etc.

Connectors in Conic are predefined, almost passive, entities. Module implementation and configuration are separated. Any dynamic reconfiguration of the system is explicitly carried out by an external operator. Therefore it is difficult to include the architectural behavior (e.g., associate configuration statements with events) of a module within the module description.

In general, these approaches are mostly concerned with providing a set of concepts and constructs, integrated in an architectural description language. The approach we describe in this paper, instead, emphasizes the configuration aspects of architectures and follows a complementary strategy, by describing and analyzing existing systems, without a particular concern on language issues (at least in the short term).

3 Archetype approach

In Archetype, we describe architecture behavior as a reactive system. The occurrence of an event causes a reaction in the system architecture and the system evolves from one state to another. From the architectural point of view, the state of a system is characterized by the configuration of component instances and the existing connections among them. The behavior of the system is thus described by the reconfiguration procedures associated with the occurrence of events.

The system architecture evolution is represented as a series of "snapshots". Each snapshot represents one configuration of the system. The configuration changes as a consequence of an event occurrence. In Archetype, we focus on the description of how the system evolves from one configuration state to another, abstracting from the functional semantics of architectural entities (which may be modeled using WRIGHT). We are interested in describing the dynamics of system startup and shutdown procedures, what are the events that may generate new component instances, how hardware resources are allocated for the execution of software components, etc. For example, in a distributed system, the "connection request" event may imply that one process is forked and a new connection channel is established with the requester.

The ultimate goal in Archetype is to gain experience and understanding of the useful primitive concepts to describe the dynamic configuration aspects of software architectures.

The description of a software architecture requires the use of a formal language. We require this language to be executable and to provide some data abstraction facilities. Since at the moment we do not want to focus on language issues, we believe that any object-oriented language (such as C++, Eiffel, or O_2C) would be acceptable for our purposes[1].

As a result of modeling software architectures, we obtain an executable description, which may be used for different purposes:

- **Documentation.** The architecture model constitutes a formal documentation of the behavioral aspects of the system. This documentation centralizes precious information about the overall organization of a system that otherwise often remains hidden in the system code and fragmented in different components.

- **Simulation.** A clear advantage of using a programming language is that the obtained architecture module is directly executable. Thus, it is possible to perform simple simulations of the

[1] Actually, since these are sequential programming languages, concurrent events are necessarily sequentialized by interleaving them explicitly.

architecture behavior, gaining the complete picture of how components are connected in a run-time dynamic scenario. The architecture may be validated by performing *what-if* analysis and by getting some understanding of non functional architectural properties, such as performance, and evolvability of the architecture. In other words, the modeler is able to *play* with an architectural definition. For example, one may introduce a delay in a method execution, representing the actual time a process takes in coming up and reason on that basis.

- **Prototyping.** The architectural model can be coded directly in the target system programming language to obtain a prototype of the system. This prototype may contain only the architectural skeleton emptied from the functional parts. This is similar to what is done for user interfaces when a system mockup is developed, based only on the user interface without bothering about other aspects of the system. The architecture skeleton gives real feedback on feasibility and performance issues in the early phases of development.
- **Guide for implementation.** The architecture model can also serve as a guide for the implementing of the whole system. Such guide could be useful in avoiding loosing track of architectural aspects in the code.

4 Preliminary experience and conclusions

As a first experience we started with the modeling of a software-engineering environment, which involves the interaction of several user, using independent tools, allocated in different machines. The modeled system is SPADE[2], since we know it very well (we have participated in its development). To code the model, we used O_2C (an object-oriented extension of C, supported by the O_2 [4] data base system) because we are used to it and because it provides some ready-to-use display facilities.

The formalization of the architectural description includes a data model of the architecture. For each identified architecture entity we have defined a class. Objects of these classes are dynamically instantiated and linked. In general, each public method of a class represents one possible event involving an architecture component of that class. When the event occurs the action is triggered and this may involve changes in the component state, the creation of new components or links, the deallocation of old ones, reconfigurations, etc.

The obtained O_2C program can be executed to simulate the system. Events are read from a file or generated interactively to drive the model evolution. The system state can be inspected at any time during simulation. In addition, a user-configurable graphical representation is provided, using the O_2 built-in graphical user interface.

To give a flavor of the approach, we provide in Figure 1 an extremely simplified version of SPADE architecture. In `state 1`, only tool A is connected to the SCI (Spade Communication Interface). The event `RunTool B` causes the system to evolve into `state 2`. The architectural behavior, associated with the event, is specified by the following code:

```
B = new Tool; /* new element creation */
P = new SCIPort;
C = new SCIConnection;

B.SCIlink = C;/* link configuration */
C.tool = B;   /* among new elements */
P.Out = C;
C.SCI = P;
/* link to an existing element (SCI) */
SCI.toolports += list(P);
```

We assume that `Tool`, `SCIPort`, and `SCIConnection` are classes that model tools, SCI ports, and connections, respectively. Moreover, `SCI` is an instance of the class `SCIModule` representing the SCI. The elements instances that are created with the `new` statements are connected through the assignments of their attributes, e.g., tool `B` is linked to the new connection (`C`) through its `SCIlink` port by the assignment:

```
B.SCIlink = C;
```

The link is then completed by stating that `B` is connected as a `tool` to `C` by the instruction:

```
C.tool = B;
```

The other three statements connect the channel `C` to the new port `P` and add `P` to the SCI port list.

Acknowledgements

We are grateful to Francesco Tisato for clarifying discussions and to Luigi Lavazza for his comments on an earlier version of this paper.

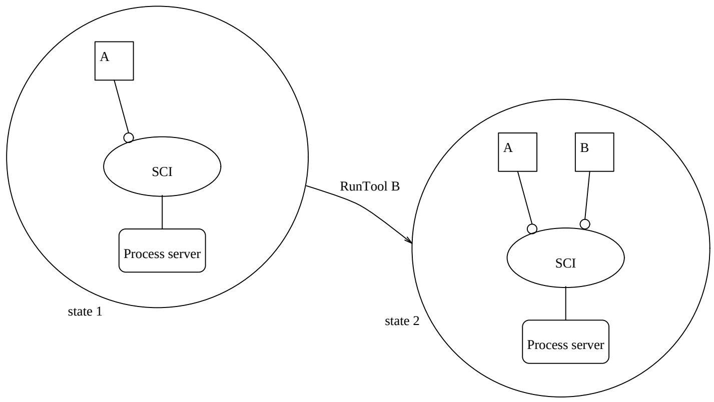

Figure 1: Example of architecture dynamic behavior.

References

[1] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento (Italy), May 1994.

[2] Sergio C. Bandinelli, Marco Braga, Alfonso Fuggetta, and Luigi Lavazza. The Architecture of the SPADE-1 Process-Centered SEE. In *Third European Workshop on Software Process Technology*, Grenoble, France, February 1994.

[3] Barry Boehm. Software Architectures: Critical Success Factors and Cost Drivers. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento (Italy), May 1994.

[4] O. Deux. The O_2 System. *Communications of the ACM*, 34(10), October 1991.

[5] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[6] David Garlan, Mary Shaw, Chris Okasaki, Curtis Scott, and Roy Swonger. Experince with a Course on Architectures for Software Systems. In *SEI Conference on Software Engineering Education*, October 1992.

[7] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. To appear on TSE, 1995.

[8] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing Distributed System in Conic. *IEEE Transactions on Software Engineering*, 6(15), June 1989.

[9] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[10] Mary Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[11] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions fro Software Architecture and Tools to Support Them. Unpublished manuscript, Computer Science Department, Carnegie Mellon University, February 1994.