A Fresh Look at Programming-in-the-Large

Walter Cazzola, Andrea Savigni, Andrea Sosio, Francesco Tisato
{cazzola|savigni|sosio|tisato}@dsi.unimi.it
Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione

Abstract

Realizing a shift of software engineering towards a component-based approach to software development requires the development of higher level programming systems supporting the development of systems from components. This paper presents a novel approach to the design of large software systems where a program-in-the-large describing the system's architecture is executed at run-time to rule over the assembly and dynamic cooperation of components. This approach has several advantages following from a clean separation of concerns between programming-in-the-small and programming-in-the-large issues in instantiated systems.

1: Introduction

For thirty years software engineering has pursued a paradigm shift from line-based to component-based software development [8]. Component software requires high levels of reusability and interoperability, and brings forth a higher level of programming, namely the assembly of independently developed and reusable components into systems, termed *programming-in-the-large* (PIL) [3] as opposed to *programming-in-the-small* (PIS, i.e., the development of individual components). While object-oriented technology provides a sound basis for the development of both fine- and coarse-grained reusable components [5], 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" [6], i.e., the *software architecture* of the system [13].

In current approaches to component software, software architectures are *implicit*, i.e., architectural decisions such as those related to the overall system's topology and global control structures are implemented in the components' code, which has several drawbacks. For example, components are less reusable and harder to integrate due to the architectural assumptions they embed [4].

Based on these considerations, we propose a new approach to programming-in-the-large which is centred around two main ideas: the *program-in-the-large* of a system should describe the system's architecture; and it should be executed at run-time as an independent (higher-level) program with respect to the components' programs. We term this approach Dynamic Architectural Programming-in-the-Large (DAPIL). This paper presents this vision. Section 2 provides the rationale for DAPIL by stating the *implicit architecture* problem underlying current approaches to component-based development. In section 3, we present a sample syntax and semantics for a programming language for DAPIL and in section 4 we sketch the component and system model underlying them. Section 5 discusses how our approach accommodates addressing dynamic architectures in a reflective way. Section 6 points out at some benefits of the proposed approach and section 7 draws some conclusion and presents future work.

2: The implicit architecture problem

Designing software architectures is a complex activity where very high-level decisions are taken affecting all the functioning of a system. For example, a phrase such as a "layered system" has a very rich meaning concerning the expected behavior of the involved components (layers), the communication topology, the expected interactions between layers, and so forth. Our goal is that of providing tools to describe such ideas and integrating components according to such description. Related research efforts can be broadly partitioned into two classes: Module Interconnection Languages (MILs) and Architectural Description Languages (ADLs). MILs [12] support the separate description of implementation relationships between modules. They are based on a very limited notion of software composition (e.g., routine definition/use binding), and hardly capture high-level, architectural choices [13]. ADLs [13] represent a more recent approach, explicitly addressing the description of architectures. Nevertheless, they are intended to be *specification languages*, whereby specified architectures must subsequently be implemented by conventional means.

In both approaches, architectural choices are dispersed in the components' code, either because they cannot be expressed by the higher-level notation (in the case of MILs) or because the notation describes concepts which anyway require conventional PIS implementation (in the case of ADLs). We term this problem the *implic-*

it architecture problem. Some consequences of implicit architectures are the following.

Implicit architectures affect components' reusability. Saying that architectural choices are dispersed in the components' code is just the same as saying that components *embed architectural assumptions*. This implies that components be reusable, at best, within a class of systems sharing common architectural characteristics; moreover, it may hinder the combined reuse of components with conflicting architectural assumptions [4].

Implicit architectures affect systems' adherence to specifications. Architectural design documents provide a high-level view of a system which is potentially a valuable support for development, maintenance, and evolution. Unfortunately, architectural descriptions are seldom coherent with implemented systems, i.e., the implicit (implemented) architecture usually differs from the explicit (specified) one, either because architectural designs are twisted in their very first implementation (due to the unfeasibility of mapping architectural concepts to those supported by traditional programming languages) or as a consequence of unfaithful maintenance [10].

Implicit architectures affect run-time system management complexity. If architectures are implicit, run-time management activities such as dynamic reconfiguration or support for plug-and-play components, which are related to the system's architecture and conceptually independent of the components' inner workings, must nevertheless be defined and implemented at the component level. This yields the well-known technical complexity of providing any form of dynamic reconfigurability.

Implicit architectures affect design reusability. If architectures are implicit, architectural designs can be reused only by reusing the code which implements them within components (this is actually the basic idea behind object-oriented frameworks [11]). So, just like components can only be reused within a class of systems with the same architecture, architectures can only be reused within a class of systems with similar implementation (e.g., same programming language).

3: The linguistic aspect of DAPIL

Our approach separates two different programming levels related to PIL. First, the basic building blocks of a "program-in-the-large" (*components* and *connectors*) are described in an *interface description language*. Second, their assembly is described by the proper program-in-the-large.

3.1: Components

We let a component's interface be structured (or *segmented* [13]), i.e., a component interacts with the environment through a set of distinct interaction points termed *ports*. Each port is specified by a state machine whose states represent abstract views of the component's internal state, as relevant for a specific pattern of cooperation with the environment, and whose transitions are labelled by *component events* occurring on that port. Component events represent a high-level view of the components' behaviour as relevant for composition with other components, and abstracting from details about the inner control flow structure; the occurrence of an event can hence involve any sort of (finite) computation *inside* the component.

In order to support the description of their assembly and interaction in a PIL language, the components' interfaces are described in an interface description language. The following fragment describes a component calculating the average of a set of numerical values. On port *data*, the component can either receive a value to store or a request to remove one of the stored items. On port *average*, it can produce the average of the currently stored numbers.

```
component average
{
    port data {
        initial_state: holds(0)
        transitions
            from holds(i) when item_inserted(?real item)
                to holds(i+1);
            from holds(i) where i>0 when item_removed()
                to holds(i-1);
        }
    port average {
        initial_state: ready
        transitions
            from ready when average_produced(!real average)
            to ready;
        }
}
```

Transitions are described as follows. Clauses **from** and **when** specify the state of the port's machine before the transition and the event that triggers the transition respectively, while clause **to** specifies the state after the transition. For example, the first transition of port *data* states that when the port is in state *holds(0)* and an event *item_inserted* occurs, the resulting state is *holds(1)*. The additional **where** clause can be used to modify either the **from** or **when** clause with preconditions. We use a CSP-style for denoting whether the value of parameters in component events is established by the component (denoted with "!") or by its environment ("?"), i.e., whether parameters are output or input.

3.2: Connectors

A connector describes a pattern of cooperation between two or more components, including protocols for communication and synchronization; it implements a protocol specified as a state machine. The protocol is based on component events each labelled by a *role*, and prescribes a pattern of events in which a set of cooperating entities can engage, each playing some role. Roles are instantiated by components' ports in actual systems (ports are said to be *bound* to roles).

The connector specified in the following fragment describes an interaction between an entity producing values ("sensor"), an entity providing a buffer to store sets of such values, an entity calculating the average (of the buffered values), and an entity displaying values to the user. The syntax is very close to that used to specify components, the main difference being that transitions are named.

```
connector SensorAverageVisualize
{
    roles: sensor, display, buffer, average_calc;
    initial_state: ready
    transitions:
       Store (real item):
          from ready when sensor.produced (?real item),
             buffer.item_insterted (!real item) to ready;
       Remove:
          from ready when buffer.item_removed()
             to ready;
       Disp_Avg:
          from ready when
             average.average_produced(?real average),
             display.value_displayed(!real average)
             to ready;
       Disp_Val (real item):
          from ready  when sensor.produced (?real
             item), display.value_displayed (!real item)
             to ready;
}
```

The described protocol is actually stateless and nondeterministic. It specifies that (at any time) values produced by a sensor can be buffered (transition *Store*), that items can be removed from the buffer (transition *Remove*), and that both averages from the average calculator and values directly coming from the sensor can be displayed (transitions *Disp_Avg* and *Disp_Val*).

As the example illustrates, connector transitions can involve a sequence of multiple component events at multiple roles; they can be regarded as higher level events termed *cooperation events*. The values of parameters in actual events is still dictated by some participant component (that which actually output the value according to its port's definition). For example, the value 1.0 of *item* in cooperation event Store(1.0) is a value which has been produced by the sensor.

3.3: The Program-in-the-Large

In our work on DAPIL, we have devised two complementary aspects in the description of software architectures: *topology* and *strategy* [2]. The system's topology defines which components comprise the system and how they are connected to one another. The system's strategy defines the global *control structure* of the system. For example, the strategy states whether the system must run as a pure concurrent system, a real-time system (with a specified plan), concurrent with priorities, and so on.

According to these definitions, the program-in-the-large is composed of two specifications. The topology specification lists the components, the connectors between them, and the bindings between ports and roles, as demonstrated in the following fragment. It describes the topology of a system comprising a component producing numerical data as collected from an external sensor, a component calculating the average, and a display component presenting values to the user. The three components are connected by a connector of the type presented in section 2.2.

```
topology SensorAverageVisualizeSystem {
    components
        ac: average;
        s: sensor;
        d: display;
    connectors
        c: SensorAverageVisualize;
    bindings
        c.sensor boundTo s.data_out;
        c.buffer boundTo ac.data;
        c.average_calc boundTo ac.average;
        c.display boundTo d.data_in;
}
```

Observe that component *ac* declared in the topology is of type *average* (described in section 3.1). Since such component can both store values and calculate averages (on distinct ports), here it is employed in both roles by binding port *average* to role *average_calc* and port *data* to role *buffer*.

The strategy specification describes the system's global control structure in terms of *a plan governing the occurrence of cooperation events*. Of course, events must be allowed by components' and connectors' protocols at the time of their occurrence, so that the strategy can be regarded as a restriction of the nondeterminism of the protocol obtained as a product of the components' and connectors' protocols. The intent is that such a restriction be made on the basis of *system-wide concerns* such as: interdependencies between computations of components which are not directly connected; timing constraints; performance optimization; availability of system resources (e.g., CPUs), and more. The following fragment illustrates a sample syntax for the strategy, very close to that adopted for components and connectors.

```
strategy SensorAverageVisualizeSystem {
    const
        STEP = 1000
        THRESH = 110.0
    initial_state
        stored(0)
    transitions
        from stored(i) where i<STEP  when c.Store(item)
           where item<THRESH to stored(i+1);
        from stored(i) where i=STEP  when c.Disp_Avg()
           to to_remove(STEP);
        from to_remove(i) where i>1 when c.Remove()
           to to_remove(i-1);
        from to_remove(1) when c.Remove() to stored(0);
        from stored(i) when c.Disp_Val(item)
           where item>THRESH to stored(i);
        from to_remove(i) when c.Disp_Val(item)
        where item>THRESH to to_remove(i);
}
```

As the reader may see, cooperation events are referred to connectors, e.g., *c.Store(item)* represents event Store(item) occurring in the connector *c* declared in the topology. The strategy restricts the nondeterminism of the connector's protocol as follows. First, a step is specified in the strategy which indicates how many values must be read before an average is displayed (transitions

1-2). When it has been displayed, the buffer is emptied (transitions 3-4), and a new cycle begins. Finally, a value read from the sensor is directly displayed without being buffered and averaged if and only if it exceeds a specified threshold (for example, this could be an alarm).

4: Component and system model

As stated above, our approach differs from that of ADLs in that we address the *execution* of architectural descriptions (programs in the large). We term *computation-in-the-large* the execution of the program-in-the-large and *computation-in-the-small* the execution of the individual components' programs [2]. The computation-in-the-large is a sequence of cooperation events as dictated by the strategy. As stated in section 3, each cooperation event consists of several components events which, in turn, can correspond to arbitrarily complex activities in the small. Hence, the computation-in-the-large can be regarded as an abstraction of the computation-in-the-small. Nevertheless, this is not achieved by having the program-in-the-large's semantics implemented in the components' programs-in-the-small; rather, a *dedicated virtual machine* executes the program-in-the-large, driving activities within components by *triggering* component events. This way, the definition of the system architecture is explicit and separated from the components' inner workings. In the component and system model, this idea is articulated in several points, listed below.

Components are architecture-transparent. This requirement on components amounts to two constraints on their implementation. The first is that the code of a component cannot directly reference entities which are outside the component's interface; the environment is visible to the component's code only through (parametric) events occurring on ports. The second constraint is that components must be reactive entities. More precisely, they have no control on the occurrence of the events they are willing to participate in.

Components are driven by connectors. From the point of view of components, control resides in connectors; this reflects the fact that component events are grouped into cooperation events. Whenever a cooperation event is triggered, the involved connector triggers the corresponding component events.

The PIL virtual machine "executes" the topology and strategy. The PIL virtual machine is conceptually composed of two entities, termed *topology actuator* and *strategy actuator*. The topology actuator "executes" the topology by working as a *mediator* [14] between components' ports and connector roles (and vice versa), conveying stimuli between them based on the topology's bindings. The strategy actuator executes the strategy by triggering cooperation events in connectors based according to the strategy's rules. Observe that since connectors are driven by the strategy actuator and, in turn, drive components, they actually constitute the interface between the computation in the large (described by the strategy) and the computation in the small (described by the components' PIS code corresponding to events).

Although at this stage we do not pose constraints on the actual implementation of the PIL virtual machine, supporting *architectural reflection* (to be discussed below) suggests that the strategy and topology actuators be implemented as *interpreters* of the topology and strategy specifications.

5: Dynamic architectures and architectural reflection

Although dynamic architectures (i.e., architectures which change at run-time, for example by the addition or removal or components) have been neglected in current research in Software Architecture [9], it is a fact that all architectures are dynamic: for example, every system at least goes through a bootstrap and a shutdown. In our approach, dynamic modifications of a system's architecture are addressed in a clean conceptual way by borrowing some concepts from *computational reflection* [7] and applying them at the PIL level, yielding *architectural reflection* (AR) [2]. We define an *architecturally reflective* as one that maintains a collection of data structure which represent (*reify*) its software architecture and are *causally connected* to the system's architecture, i.e., any modification to such structures results in the concrete system's architecture being modified, and vice versa. Dynamic modifications of the architecture are charged to *architectural meta-entities* which manipulate the architecture through the manipulation of the data structures reifying it. DAPIL cleanly accommodates AR due to the separation between architectural (PIL) concerns, charged to the topology and strategy actuators, and PIS concerns. More specifically, if the actuators are implemented as interpreters of the PIL program (or a representation thereof), AR can be realized by having architectural meta-entities operate on such program, which is implicitly a causally connected representation of the system's architecture. In other implementation schemes, meta-entities may control and modify the behavior of the actuators just as traditional meta-entities modify the behavior of objects in computational reflection. In any case, DAPIL has the advantage that reflecting on the software architecture of a system amounts to reflecting on the behavior of a localized subsystem (that comprising the two actuators), rather than on the components' internals (as would be the case if the architecture was implicit).

6: General advantages of DAPIL

Two basic advantages come from making architectures explicit through DAPIL. First, it enhances component reuse since components do not embed architectural assumptions. Second, a (trustworthy) description of architecture of a system into a program-in-the-large provides benefits for all software lifecycle activities which relate to, or otherwise can benefit from a knowledge of, the architecture itself. Many such activities exist, belonging to the system's *development*, *run-time management*, and *evolution*.

Development. DAPIL can aid system development in several ways, thanks to the possibility of designing and developing the program-in-the-large independent of, and before, the components themselves so that it can be used, for example, to simulate the system's overall computation before developing or integrating components. Moreover, reusing programs-in-the-large is a way to extend software reuse from code to (architectural) design.

Run-time system management. In the DAPIL approach, run-time system management facilities are addressed at PIL level. In particular, dynamic reconfiguration is addressed by dynamic modifications of the program-in-the-large, as discussed in section 5.

Evolution. Evolution of a software system often includes architecture redesign, e.g., including new kinds of components, modifying the overall control structure, or integrating the system into a larger one, i.e., recasting its architecture as a sub-architecture. In the DAPIL approach, architecture redesign involves a (localized, high-level) modification of the program-in-the-large rather than complex and error-prone modification of the architectural information dispersed in the components' code as PIS constructs.

7: Conclusions and future work

This paper presents a new approach to the development of systems from coarse-grained components. In the proposed approach, the software architecture of a component-based system is described by a program-in-the-large which is executed at run-time. In particular, the notion of *strategy* captures a high-level view of the system's execution flow as relevant at the architectural level, the *flow of execution in the large*. We argue that the proposed approach provides several benefits following from the explicitation of architectural design in a separated higher-level program.

We are currently working on the development of a proof-of-concept Java-based PIL environment. On a more conceptual level, we are investigating several issues which can be seen as extensions to the PIL level of well understood concepts from PIS. Most notably, we are addressing the problem of defining a sound type system for PIL-level entities. Some examples of typing issues for DAPIL are the following:

Port/role compatibility. Safe PIL requires that connector roles be bound only to compatible component ports. Several related notions of compatibility have been proposed in the literature; most simply amount to deadlock-freeness of the composition (e.g., [1]). We would like to devise a deeper notion of compatibility capturing the concept of semantics-preserving interoperability.

Port/role (sub)type. A notion of subtyping should be defined between port and role types so that one can derive port/role compatibility for subtypes, e.g., if a port type is compatible with a role type it is also compatible with all of its subtypes, and so are all of its supertypes.

Component/connector (sub)type. Components and connectors' types should be based on port/role types and express the fact that whenever an instance of a component (connector) type can be placed in a certain position of an architecture, so can instances of any subtype.

Closure. Closure rules should be devised specifying under which circumstances a component/connector can be substituted with a (sub)architecture or, in other words, under which circumstances an architecture can be regarded as an instance of a component/connector type.

References

[1] Allen, R. and Garlan, D., "Formalizing Architectural Connection," in *Proc. 16th Int. Conf. on Softw. Eng.*, Sorrento (Italy), 1994.

[2] Cazzola, W., Savigni, A., Sosio, A. and Tisato, F., "Architectural Reflection: Bridging the Gap between a Running System and Its Architectural Specification," in *Proc. Reengineering Forum '98*, Florence (Italy), 1998.

[3] DeRemer, F. and Kron, H., "Programming-in-the-Large Vs. Programming-in-the-Small." *IEEE Trans. Softw. Eng.*, vol. 2, no. 2, 1976.

[4] Garlan, D., Allen, R. and Ockerbloom, J., "Architectural Mismatch, or Why It's Hard to Build Systems Out of Existing Parts," in *Proc. 17th Int. Conf. Softw. Eng*, Seattle, 1995.

[5] Jazayeri, M. "Component Programming: A Fresh Look at Software Components." Tech. Rep. TUV-1841-95-01, Technical University of Vienna, 1995.

[6] Luckham, D.C., Kenney, J.F., Augustin, L.M., Vera, J., Bryan, D., Mann, W., "Specification and Analysis of System Architecture Using Rapide." , *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, 336-355, Apr. 1995.

[7] Maes., P. "Concepts and Experiments in Computational Reflection," in *Proc. OOPSLA'87*, Sigplan Notices, ACM, Oct. 1987.

[8] McIlroy, D., "Mass-Produced Software Components," in P. Naur et al. (eds.), *Software Engineering Concepts and Techniques*, Petrocelli Charter 1976, pp. 88-98.

[9] Medvidovic, N. "ADLs and Dynamic Architecture Changes," in *Proc. 2nd International Workshop on Software Architecture*, ACM 1996, pp. 24-27.

[10] Murphy, G.C., "Architecture for Evolution," in *Proc. 2nd International Software Architecture Workshop*, ACM, 1996.

[11] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley 1994.

[12] Prieto-Diaz, R. and Neighbors, J.M., "Module Interconnection Languages." *Journal of Systems and Software*, vol. 6, no. 4, 307-334, 1986.

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

[14] Sullivan, K.J., Kalet, I.J. and Notkin, D., "Evaluating the Mediator Method: Prism as a Case Study." *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, Aug. 1996.