Specifying and Analyzing Dynamic Software Architectures*

Robert Allen, Rémi Douence, and David Garlan

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

rallen@cs.cmu.edu douence@irisa.fr garlan@cs.cmu.edu

Abstract. A critical issue for complex component-based systems design is the modeling and analysis of architecture. One of the complicating factors in developing architectural models is accounting for systems whose architecture changes dynamically (during run time). This is because dynamic changes to architectural structure may interact in subtle ways with on-going computations of the system.

In this paper we argue that it is possible and valuable to provide a modeling approach that accounts for the interactions between architectural reconfiguration and non-reconfiguration system functionality, while maintaining a separation of concerns between these two aspects of a system. The key to the approach is to use a uniform notation and semantic base for both reconfiguration and steady-state behavior, while at the same time providing syntactic separation between the two. As we will show, this permits us to view the architecture in terms of a set of possible architectural snapshots, each with its own steady-state behavior. Transitions between these snapshots are accounted for by special reconfiguration-triggering events.

1 Introduction

Recently, there has been considerable progress on the development of architecture description languages (ADLs [12]) to support software architecture design and analysis. These languages capture the key design properties of a system by exposing the architectural structure as a composition of components interacting via connectors. Examples include Wright [1], UniCon [14], Rapide [10], Darwin [11] and ACME [5].

There are many aspects of a software system that can be addressed in an architectural description, including functional behavior, allocation of resources, performance, fault-tolerance, flexibility in the face of altered requirements, and so on. Each ADL tends to focus on one or more of these aspects.

* Research sponsored by the INRIA, the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031, and by the National Science Foundation under Grant No. CCR-9357792. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of INRIA, the Defense Advanced Research Projects Agency Rome Laboratory, or the U.S. Government.

In this paper we address the problem of capturing *dynamic* architectures. By "dynamic" we mean systems for which composition of interacting components changes during the course of a single computation. We distinguish this aspect of dynamic behavior from the *steady-state behavior*, by which we mean the computation performed by a system without reconfiguration.

We argue that it is both possible and valuable to separate the dynamic re-configuration behavior of an architecture from its non-reconfiguration functionality. While there exist ADLs, such as Darwin, that capture reconfiguration behavior, and facilities such as object-oriented languages that permit the combined description of both dynamic aspects and steady-state behavior, we believe that, at the architectural level, it is important to provide a notation that supports *both* aspects of design *while maintaining a separation of concerns*. In this paper we illustrate a new technique by which these two aspects can be described in a single formalism, while keeping them as separate "views". This facilitates the understanding of each aspect in isolation while still supporting analysis of the combined interaction between the two.

By providing a notation that provides a precise interpretation of each of these aspects, we permit the description to be analyzed for consistency and completeness, as well as for whether the system has application-specific properties desired by the architect. For example, we would like to guarantee that reconfigurations occur only at points in the computation permitted by the participating components and connectors. Also, whenever a new connection is established, we would like to show that the participating components exist at the moment of attachment. By considering interactions between the two forms of description, we can consider whether changing the participants at run time will result in inconsistencies among participants' states.

In the remainder of this paper we present our technique using the Wright ADL as our notational basis. We first review related works (Section 2). We introduce Wright (Section 3) and illustrate the problem of specifying dynamic architectures (Section 4). Then we show how a language originally designed for steady-state architectures, such as Wright, can be extended to handle dynamic aspects of architecture (Section 5). Next, we present the semantic model on which the approach is based (Section 6). Then we illustrate the kinds of analysis that such a formalism supports (Section 7). Finally, we discuss possible extensions of our work (Section 8).

2 Related Works

Our work is most closely related to two general classes of research. The first is architecture description languages. While there are a large number of such languages, only a few are capable of modeling dynamic architectures. The most prominent among these are Rapide and Darwin. In the case of Rapide, the notation takes an object-oriented view: new architectural components can be created much as one would create new objects in a object-oriented programming language [10]. A consequence of this design is that it is in general undecidable what topologies will be created during a Rapide execution. For this and other reasons, Rapide focuses on simulation and analysis of sets of execution traces. In contrast, Wright focuses on static checking.

In the case of Darwin, the language is solely concerned with the *structural* aspects of an architecture [11]. Thus, the issue of how reconfigurations interact with on-going computations does not arise. However, their use of the Pi-Calculus to give semantics to recon-

figuration is elegant and suggestive of the power of a more flexible "dynamic" process algebras.

The second area of related work is general formalisms for reasoning about architectural designs. Among these two are most closely related. The first is term rewriting systems. For example, Inverardi and Wolf have shown how to model architectures using the CHAM [7]. As a general term rewriting system, CHAM is can describe arbitrary reconfigurations of architectures. While this approach has considerable power, the cost is that the description of systems and reconfigurations must be encoded in system rewriting rules, which may be far removed from the intuitive descriptions used by system designers. In contrast, Wright has tried to provide a notation that makes explicit the intentions of a designer for handling reconfigurability.

Another general-purpose formalism applied to dynamic architectures is the use of graph grammars to describe the allowable topologies of architectures [9]. Graph grammars provide a nice notation for capturing patterns of transformation. However, thus far they have not been used to relate the reconfiguration aspects of an architecture with its behavior. Thus, as before, it is not possible to reason about when it is legal to carry out architectural reconfiguration.

3 Motivating Example

Consider the simple client-server system shown in Figure 1. It consists of one client and one server interacting via a link connector. Such a system is easy to describe in an ADL such as Wright [1,2]. There are two essential aspects of a Wright description of a system architecture: architectural structure, and architectural behavior. We illustrate both aspects using the example, which is described in Wright in Figure 2.

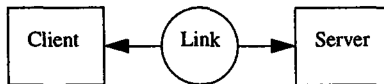

Figure 1 Static Topology : Simple Client-Server System

3.1 Architectural Structure

Wright represents architectural structure as graph of components and connectors. Components represent architecturally-relevant units of computation and data storage, while connectors represent the interactions between components. In Wright components and connectors are typed. Thus to define a system, one first declares a set of component and connector types, termed as a *Style*. Then one declares a set of instances of these types and the way in which they are assembled.

Figure 2 opens with a definition of the *Client-Server* style by declaring a *Client* and *Server* component types, as well as a *Link* connector type. Components have interfaces, which in Wright are called *ports*. Any component may have multiple ports, each port defining a logically separable point of interaction with its environment. Here, both our client and server component types have only one port *p*. Connectors also have interfaces, which

are termed *roles*. The roles of a connector identify the logical participants in the interaction represented by the connector, and (as we will see shortly) specify the expected behavior of each participant in the interaction. In Figure 2, the *Link* connector has a role *c* for the client and one *s* for the server. Finally, the constraints define a set of predicates that every configurations conforming to that style must satisfy.

Style Client-Server
  Component Client
    Port p = $\overline{\text{request}} \rightarrow \text{reply} \rightarrow \text{p} \sqcap \S$
    Computation = internalCompute $\rightarrow \overline{\text{p.request}} \rightarrow$ p.reply $\rightarrow$ **Computation** $\sqcap \S$

  Component Server
    Port p = request $\rightarrow \overline{\text{reply}} \rightarrow$ p $\square \S$
    Computation = p.request $\rightarrow$ internalCompute $\rightarrow \overline{\text{p.reply}} \rightarrow$ **Computation** $\square \S$

  Connector Link
    Role c = $\overline{\text{request}} \rightarrow$ reply $\rightarrow$ c $\sqcap \S$
    Role s = request $\rightarrow \overline{\text{reply}} \rightarrow$ s $\square \S$
    Glue = c.request $\rightarrow \overline{\text{s.request}} \rightarrow$ **Glue**
        $\square$ s.reply $\rightarrow \overline{\text{c.reply}} \rightarrow$ **Glue**
        $\square \S$

  Constraints
    $\exists!\, s \in \text{Component}, \forall\, c \in \text{Component} : \text{Server}(s) \wedge \text{Client}(c) \Rightarrow \text{connected}(c,s)$
EndStyle

Configuration Simple
  Style Client-Server
  Instances C : Client ; L : Link ; S : Server
  Attachments C.p **as** L.c ; S.p **as** L.s
EndConfiguration

Figure 2 Static Wright Specification : Simple Client-Server System

---

A small configuration is also shown in Figure 2. It uses the Client-Server style (making the components and connector types available) to declare a single client-server connector. These part are then assembled. Specifically, the port *p* of the client *C* fills the role *c* of the connector *L*, while the port *p* of the server *S* fills the role *s* of the connector.

3.2 Architectural Behavior

Wright focuses on architectural behavior characterized in terms of the significant events that take place in the computations of a components, and the interactions between components as described by the connectors. It allows the user to formally specify behavior such as: "the client makes a request, which is received by the server, and the server provides a response, that is communicated to the client; this sequence of actions can be repeated many times". The notation for specifying event-based behavior is adapted from CSP [6]. Each CSP process defines an alphabet of events and the permitted patterns of events that the process may exhibit. These processes synchronize on common events (i.e., interact)

when composed in parallel. Wright uses such process descriptions in computation, port, role and glue specifications.

A *computation* defines a component's behavior: the way in which it accepts certain events on certain *ports* and produces new events on those or other ports. As illustrated in Figure 2, a *Client* iteratively makes a request ($\overline{p.request}$) and waits a reply (*p.reply*) on port *p*, or terminates successfully (§*). The use of internal choice ($\sqcap$) in the specification indicates that it is the client that decides whether it makes a request or terminates. In contrast, the use of external choice ($\square$) in the *Server* specification indicates that the server is expected to respond to any number of requests, and may not terminate prematurely. Moreover, because we are interested in how different components control interactions, Wright distinguishes initiated events from observed events by an overbar. For example, the client initiates requests (e.g., $\overline{p.request}$) while the server observes them (e.g., *p.request*).

A *port* process defines the local protocol with which the component interacts with its environment through that port. This protocol is effectively the projection of the component's computation onto the particular interface point. For example, the client single port *p* faithfully reproduces the client computation pattern, but hides the internal computation modeled by the event *internalCompute*.

A *role* specifies the protocol that must be satisfied by any port that is attached to that role. In general, a port need not have the same behavior as the role that it fills, but may choose to use only a subset of the connector capabilities. In our simple case, the link role *c* and the client port *p* for example are identical.

Finally, a glue specification describes how the roles of a connector interact with each other. In the example, a client request (*c.request*) must be transmitted to the server ($\overline{s.request}$), and the server reply (*s.reply*) must be transmitted back to the client ($\overline{c.reply}$). We have described in [1,2] how these formal descriptions can be used to check the consistency and completeness of architectural descriptions. We come back to this topic in section 7.

In this description, the server does not terminate until the client is ready. But, what of a more realistic situation, where the server is running on an unreliable processor over a network, and may crash unexpectedly? In this case, the architect must consider two aspects of a robust architectural design. First is the simple view of the normal system behavior, in which the client makes a request of the server, and receives a response. Second is the architect's solution to the problem of server crashes (i.e., the way in which a server is restarted or replaced so that there is always a service available for the client). In the next section we look at one approach to unifying these two aspects of design.

4 Simulating Dynamism

Let us now consider one possible solution (see Figure 3) in which there are two servers: a "primary" server, which is more desirable to use, but which may go down unexpectedly, and a "secondary" server, which, while reliable, provides a lesser form of the service. One way to use these is to alter the architecture such that both servers are present, and when the

---

* § is similar to the CSP process TICK, except that § represents a *willingness* to terminate rather than a *decision* to terminate. So, § can occur at choice points (e.g., P $\square$ § and P $\sqcap$ §), which is illegal in standard CSP. An alternative definition of the sequencing operator ";" makes this kind of expression consistent. See [1] for details.

primary server goes down, the client uses the secondary server until such time as the primary server returns to service. (This kind of fault-tolerant architecture is actually used by the Simplex System [13].) The topology of the system is shown in Figure 3.

A Wright description of a possible client is shown in Figure 4. In this solution the primary server communicates its status to the client with *down* and *up* events when it is about to go down or come up (resp). (In practise, such events might be supplied by a time-out service.) These events appear in the *Client* definition. The port *Primary* expresses the client assumptions that the primary server can go down anytime, except in the middle of a request (i.e., not between $\overline{request}$ and *reply*), and once it goes down it provides no service until it comes up. The secondary server is a safe one that is always ready to serve. It is not concerned with *down* and *up* events (which do not occur on the port *Secondary*). The client computation now has two states (*UsePrimary* and *UseSecondary*) encoding which is the active server. The down and up events switch from one state to the other.

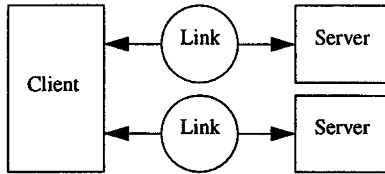

Figure 3 Static Topology : Fault-Tolerant Client-Server System

Component Client
Port Primary = § ⊓ (($\overline{\text{request}}$ → reply → Primary) [] down → (§ ⊓ up → Primary))
Port Secondary = § ⊓ $\overline{\text{request}}$ → reply → Secondary
Computation = UsePrimary
where UsePrimary = $\overline{\text{internalCompute}}$ → (TryPrimary ⊓ §)
TryPrimary = $\overline{\text{primary.request}}$ → primary.reply → UsePrimary
[] primary.down → TrySecondary
UseSecondary = $\overline{\text{internalCompute}}$ → (TrySecondary ⊓ §)
TrySecondary = $\overline{\text{secondary.request}}$ → secondary.reply → UseSecondary
[] primary.up → TryPrimary

Figure 4 Static Wright Specification : Client Component (Fault-Tolerant Client-Server)

While this accomplishes what we set out to do, the description of the new architecture has several disadvantages:

- The simple client-server functional pattern and topology have been lost. In particular, its specification is now muddied by the need to consider the effects of reconfiguration at almost each step, and the client-server style constraint (see Figure 2: exactly one server is connected to every clients) is no longer true.

- It has been necessary to significantly alter the original simple client in order to accommodate a change that arguably should occur on the server's side. We have had to duplicate Client's Port, so this component now must keep state to know which port to use (i.e., which is the active server). Ideally, the client should be able to continue to operate as before, but have the *system* handle rerouting of requests to the new server.
- Distribution of the configuration state and re-configuration actions in the components makes the modifications of this system difficult. For example, adding a third backup, or permitting only the primary to go down once before abandoning it, requires extensive changes to all parts of the system, thus reducing the reusability of the constituent elements.

Instead of rigid encoding of the dynamism in the steady state behavior of the components, what we would like is to provide constructs to describe the dynamics of the system explicitly. In this case, rather than using a *fixed* topology of two servers and hiding the changes inside a client component' "choice" about which server to use, we could describe the server's failures as triggers that change the topology during computation. In effect, instead of the single configuration shown in Figure 3, we would have *two* configurations, shown in Figure 5. These configurations, each simple in itself, alternate as the primary server goes down and comes back up.

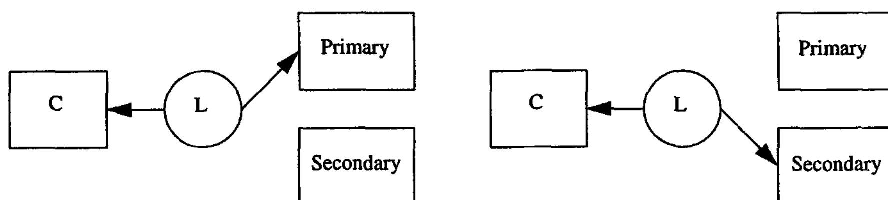

Figure 5 Dynamic Topology : Alternating Configurations of a Fault-Tolerant Client-Server

In order to achieve this effect, we must introduce notations for characterizing changes in the architecture during a computation. Such a characterization includes: (a) what events in the computation trigger a re-configuration, and (b) how the system should be reconfigured in response to a trigger.

5 Our Approach

Our solution consists of two parts. First, special "control" events are introduced into a component's alphabet, and allowed to occur in port descriptions. In this way, the interface of a component is extended to describe when reconfigurations are permitted in each protocol in which it participates. Second, these control events are used in a separate view of the architecture, the configuration program, which describes how these events trigger reconfigurations.

To illustrate, consider the fault-tolerant client-server style described in Figure 6. The architectural types (*Client*, *FlakyServer*, *SlowServer*) are declared in the usual way, but they also include events like *down* and *up*, now explicitly marked as "control" rather than "communication" events. The *FlakyServer* indicates the states in which it may go down

(with *control.down*) and come up (with *control.up*). Specifically, it may go down at anytime, except in the middle of a transaction (i.e., not between *request* and $\overline{reply}$) and it may come up anytime after a *down*. These events indicate the states in which the server accepts a reconfiguration. In the same way, a *SlowServer* component can be turned *on* then *off* anytime, except in the middle of a transaction. And a *FTLink* can be reconfigured (*changeOk*) between request and reply transmissions.

Style Fault-Tolerant-Client-Server
  Component FlakyServer
    Port p = § $\square$ (request $\rightarrow$ $\overline{\text{reply}}$ $\rightarrow$ p $\sqcap$ control.down $\rightarrow$ (§ $\square$ control.up $\rightarrow$ p))
    Computation = § $\square$ (p.request $\rightarrow$ internalCompute $\rightarrow$ $\overline{\text{p.reply}}$ $\rightarrow$ **Computation**
        $\sqcap$ control.down $\rightarrow$ (§ $\square$ control.up $\rightarrow$ **Computation**))

  Component SlowServer
    Port p = § $\square$ control.on $\rightarrow$ μLoop.(request $\rightarrow$ $\overline{\text{reply}}$ $\rightarrow$ Loop $\square$ control.off $\rightarrow$ p $\square$ §)
    Computation = § $\square$ control.on $\rightarrow$ μLoop. (control.off $\rightarrow$ **Computation** $\square$ §
      $\square$ p.request $\rightarrow$ internalCompute $\rightarrow$ $\overline{\text{p.reply}}$ $\rightarrow$ Loop

    Connector FTLink
      Role c = $\overline{\text{request}}$ $\rightarrow$ reply $\rightarrow$ c $\sqcap$ §
      Role s = (request $\rightarrow$ $\overline{\text{reply}}$ $\rightarrow$ s $\sqcap$ control.changeOk $\rightarrow$ s) $\square$ §
      Glue = c.request $\rightarrow$ $\overline{\text{s.request}}$ $\rightarrow$ **Glue**
        $\square$ s.reply $\rightarrow$ $\overline{\text{c.reply}}$ $\rightarrow$ **Glue**
        $\square$ §
        $\square$ control.changeOk $\rightarrow$ **Glue**

  Constraints
    $\exists!\ s \in$ Component, $\forall\ c \in$ Component : Server(s) $\wedge$ Client(c) $\Rightarrow$ connected(c,s)
End Style

Figure 6 Dynamic Wright Specification : Fault-Tolerant Client-Server Style

---

This new style can be used to build different systems. For example, Figure 7 pictures our version of the fault-tolerant client-server system. The *Configuror* is responsible for achieving the changes to the architectural topology (triggered by *up* and *down*) using instances of architecture types (e.g., *Client*, *FlakyServer*, *SlowServer*), and *new*, *del*, *attach*, and *detach* actions, as illustrated in Figure 8.

In this reconfiguration program, unlike the previous solution, the *Client* component type is identical to the original one in Figure 2. The initial sequence of actions (*new* and *attach*) builds the original system. Then *WaitForDown* describes two situations: the system can run and successfully terminate (§) or a fault can occur. If the primary server goes down, the secondary server is in state *on* and the link connector is reconfigurable, the primary server is detached from the link and is replaced by the secondary server. The new configuration then resumes its execution until it terminates or the primary server comes up. In this latter case, *WaitForUp* specifies that when the secondary server is *off* and the link connector is reconfigurable (*changeOk*), the secondary server is detached from the link and is replaced by the primary server.

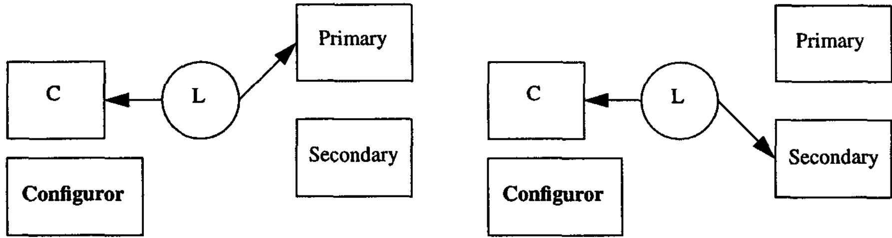

Figure 7 Dynamic Topology : Alternating Configurations of Fault-Tolerant Client-Server

```
Configuror DynamicClient-Server
   Style Fault-Tolerant-Client-Server
         new.C:Client
         → new.Primary : FlakyServer
         → new.Secondary : SlowServer
         → new.L : FTLink
         → attach.C.p.to.L.c
         → attach.Primary.p.to.L.s → WaitForDown
where
WaitForDown = (Primary.control.down → Secondary.control.on →
                    L.control.changeOk → Style Fault-Tolerant-Client-Server
                                            detach.Primary.p.from.L.s
                                            → attach.Secondary.p.to.L.s
                                            → WaitForUp)
              ▯ §
WaitForUp   = (Primary.control.up → Secondary.control.off →
                    L.control.changeOk → Style Fault-Tolerant-Client-Server
                                            detach.Secondary.p.from.L.s
                                            → attach.Primary.p.to.L.s
                                            → WaitForDown)
              ▯ §
```

Figure 8 Dynamic Wright Specification : Configuror (Fault-Tolerant Client-Server System)

Thus, this configuror describes three configurations (an initial one, and two alternating configurations). In the configuror, a style annotation specifies the set of component and connector types a configuration can use and the constraints it must satisfy. Here we use the same style. In general, each configuration can use a different style. In this example, it is easy to see that the *Fault-Tolerant-Client-Server* style constraint is verified in each of the three possible configurations: in each configuration, the client is connected to one server.

6 Semantics

Thus far we have relied on the reader's intuition and good faith that the notation outlined above makes sense. In this section we present the formal basis for this.

The basic idea is to translate the notation into pure CSP [6]. The behavior of the system is constructed from the process that defines the constituent components and connectors. Specifically, the behavior of a system is the parallel composition of all the computation, glue and configuror processes. In attempting to provide such a semantics, the key difficulties are to account for the "dynamic" creation and deletion of processes, and to arrange things so that the alphabets of the evolving topology leads to the intended interactions. The problem, of course, is that CSP can only describe a static configuration of processes. (That is, one can't create new processes and communication channels "on the fly"). How then can one give meaning to such actions as *new*, *delete*, etc.?

Our approach is based on two key ideas. First we restrict systems to those for which there are a finite (albeit potentially large) set of possible configurations (see section 8 for further discussion of this issue). Second, in the translation to CSP we "tag" events with the configuration in which that event occurs. (Because of our restriction to finite set of configurations, there is a finite number of possible tags.) The effect of a reconfiguration action is to select the properly tagged version of CSP expressions so that the interactions occur as defined by the new configuration. Thus an (untagged) *e*, in one configuration might end up (after tagging) synchronizing with one process in one configuration, but with another in a different configuration.

More formally, each event *p.e* of the component *Cp** is relabelled as *Cp.p.e.Cn.r* when the port *p* of *Cp* is attached to the role *r* of the connector *Cn*. For example, in our fault-tolerant client server system when the *FTLink* connector interacts with the *Primary* server, the "reply" case of *L* (s.reply $\rightarrow$ $\overline{\text{c.reply}}$ $\rightarrow$ **Glue**) is translated to:

L.s.reply.Primary.p $\rightarrow$ $\overline{\text{L.c.reply.C.p}}$ $\rightarrow$ **Glue**

This indicates that a reply received by *L* on role *s* (from the *Primary* server port *p*) must be transmitted (by *L* on the role *c*) to the client *C* on port *p*.

Our transformation also introduces the "plumbing" that allows selection of the proper version: each version of a transformed component begins with a $Cp.go.p_1.Cn_1.r_1 \ldots r_n.Cn_n.r_n$ event selecting the version of *Cp*, where its port p_1 is attached to the role r_1 of $Cn_1, \ldots$ and its port p_n attached to the role r_n of Cn_n. For example, the connector L is translated to:

L = L.go.c.C.p.s.Primary.p $\rightarrow$ $\mathbf{Glue_1}$
[] L.go.c.C.p.s.Secondary.p $\rightarrow$ $\mathbf{Glue_2}$
$\mathbf{Glue_1}$ = L.c.request.C.p $\rightarrow$ ($\overline{\text{L.s.request.Primary.p}}$ $\rightarrow$ $\mathbf{Glue_1}$ [] ...
$\mathbf{Glue_2}$ = L.c.request.C.p $\rightarrow$ ($\overline{\text{L.s.request.Secondary.p}}$ $\rightarrow$ $\mathbf{Glue_2}$ [] ...

Finally, the configuror definition is transformed into a CSP process too, where the *new*, *del*, *attach* and *detach* actions are transformed into the previous *Cp.go...* events that select the proper configuration of the components at each reconfiguration step. For example, the following configuror portion:

new.L $\rightarrow$ Attach.C.p.to.L.c $\rightarrow$ Attach.Primary.p.to.L.s $\rightarrow$ ...

is transformed to L.go.c.C.p.s.Primary.p $\rightarrow$ ...

* In the rest of this paper, *Cp* designates a Wright component, *p* a port, *Cn* a connector, *r* a role, and *C* a component or a connector.

The component transformation $\mathcal{TComp}$ is formally defined in Figure 9. The transformation $\mathcal{TComp}_1$ introduces a main label (to restart the component in its initial state) and enumerates the different versions of the code with the help of the CSP notation $\forall\, i : \{1, 2, 3\}\ \Box\ P_i = P_1\ \Box\ P_2\ \Box\ P_3$. In each version the events are renamed according to the current attachments (an unattached port is treated as attached to the role *void* of the dummy connector *Void*). The next to last substitution renames the internal events, and the last one restarts the process in its initial state after each control event, so that a different version can be selected. If a control event occurrence is followed by an arbitrary expression (rather than the label **Computation**), an auxiliary definition should be introduced and compiled similarly.

$\forall$ Cp: Component and Ports(Cp) = $\{p_1, \ldots\ p_n\}$
    $\mathcal{TComp}$ [[Cp]] = $\mathcal{TComp}_1$[[Computation(Cp)]]

$\mathcal{TComp}_1$[[P]] = μCp.main.
  § $\Box$
  ($\forall$ Cn_1: Connectors $\cup$ {Void},...,$\forall$ Cn_n: Connectors $\cup$ {Void}
  $\forall$ r_1: Roles(Cn_1),...,$\forall$ r_n: Roles(Cn_n)
  $\Box$
  (Cp.go.p_1.Cn_1.r_1...p_n.Cn_n.r_n $\rightarrow$ P [Cp.p_1.e.Cn_1.r_1 / p_1.e]
    ...
    [Cp.p_n.e.Cn_n.r_n / p_n.e]
    [Cp.e / e]
  [Cp.control.evt $\rightarrow$ Cp.main / control.evt $\rightarrow$ **Computation**]))

Figure 9 Component Cp Semantics

The connector definitions are "compiled" in the same way. The Glue transformation (not shown here) is symmetric: rather than enumerating all possible roles that may attach to the given port, we vary the ports, while holding the role fixed. The renaming pattern is similar to that of Figure 9 (i.e., *Cp.p.e.Cn.r*), so that it matches the "compiled" component events.

The grammar in Figure 10 defines legal configurors. An initial sequence of action events building the original system is followed by re-configuration rules. Basically, a reconfiguration rule is defined by a triggering sequence of control events followed by a sequence of action events. Several rules can compete in parallel ($Rule_1\ \Box \ldots \Box\ Rule_n$); one of them is selected as the components produce the proper control events. Every rule is followed by a piece of configuror (i.e., the next rules to apply). The success process § terminates the reconfigurations and recursion permits infinite reconfiguration sequences.

Main = $Action^+$ $\rightarrow$ Configuror   where $A^+ = A \rightarrow A^+ \mid A$
Configuror = ($Rule_1\ \Box \ldots \Box\ Rule_n$) | μX.Configuror(X)
Rule = § | C.control.evt^+ $\rightarrow$ $Action^+$ $\rightarrow$ Configuror
Action = new.C | del.C | attach.Cp.p.to.Cn.r | detach.Cp.p.from.Cn.r

Figure 10 Configuror BNF Grammar

The Configuror transformation $\mathcal{TConf}$ is defined in Figure 11 with the help of a *cfg* argument recording the current configuration. In order to make the transformation simpler, we assume a first pass has reordered all actions so that each reconfiguration rule follows the sequence: detachments, deletions, attachments and creations. The first rule in Figure

11 reproduces the control events, the second and fourth ones maintain the *cfg* argument. The rule for *new* produces *go* events according to *cfg*. The rule for *del* produces nothing, since according to our transformation in Figure 9 after a control event the component or connector is in its initial state waiting for a go. The seventh case detects the end of a rule and resumes (go events) the components and connectors which triggered the current rule and haven't been deleted, then it calls $\mathcal{T}Conf_2$. The four $\mathcal{T}Conf_2$ rules reproduce the configuror structure.

$$\mathcal{T}Conf[\![\, \text{C.control.e} \rightarrow P \,]\!]\ \text{cfg} = \text{C.control.e} \rightarrow \mathcal{T}Conf[\![\, P \,]\!]\ \text{cfg}$$
$$\mathcal{T}Conf[\![\, \text{detach.Cp.p.from.Cn.r} \rightarrow P \,]\!]\text{cfg} = \mathcal{T}Conf[\![\, P \,]\!]\ (\text{detach Cp.p Cn.r cfg})$$
$$\mathcal{T}Conf[\![\, \text{del.C} \rightarrow P \,]\!]\ \text{cfg} = \mathcal{T}Conf[\![\, P \,]\!]\ (\text{del C cfg})$$
$$\mathcal{T}Conf[\![\, \text{attach.Cp.p.to.Cn.r} \rightarrow P \,]\!]\ \text{cfg} = \mathcal{T}Conf[\![\, P \,]\!]\ (\text{attach Cp.p Cn.r cfg})$$
$$\mathcal{T}Conf[\![\, \text{new.Cp} \rightarrow P \,]\!]\ \text{cfg} = \text{Cp.go.p}_1.\text{Cn}_1.\text{r}_1 \ldots \text{p}_n.\text{Cn}_n.\text{r}_n \rightarrow \mathcal{T}Conf[\![\, P \,]\!]\ (\text{new Cp cfg})$$
$$\mathcal{T}Conf[\![\, \text{new.Cn} \rightarrow P \,]\!]\ \text{cfg} = \text{Cn.go.r}_1.\text{Cp}_1.\text{p}_1 \ldots \text{r}_n.\text{Cp}_n.\text{p}_n \rightarrow \mathcal{T}Conf[\![\, P \,]\!]\ (\text{new Cn cfg})$$
$$\mathcal{T}Conf[\![\, P \,]\!]\ \text{cfg} = \text{Cp}_1.\text{go.p}_{11}.\text{Cn}_{11}.\text{r}_{11} \ldots \text{p}_{1n}.\text{Cn}_{1n}.\text{r}_{1n} \rightarrow \ldots \ \text{Cn}_m.\text{go.r}_{m1}.\text{Cp}_{m1}.\text{p}_{m1} \ldots \text{r}_{mp}.\text{Cp}_{mp}.\text{p}_{mp} \rightarrow (\mathcal{T}Conf_2[\![\, P \,]\!]\ \text{cfg})$$
$$\mathcal{T}Conf_2[\![\, \text{Rule}_1 \,\square \ldots \square\, \text{Rule}_n \,]\!]\ \text{cfg} = (\mathcal{T}Conf[\![\, \text{Rule}_1 \,]\!]\ \text{cfg}) \,\square \ldots \square\, (\mathcal{T}Conf[\![\, \text{Rule}_n \,]\!]\ \text{cfg})$$
$$\mathcal{T}Conf_2[\![\, \mu X.P(X) \,]\!]\ \text{cfg} = \mu X.(\mathcal{T}Conf[\![\, P(X) \,]\!]\ \text{cfg})$$
$$\mathcal{T}Conf_2[\![\, X \,]\!]\ \text{cfg} = X$$
$$\mathcal{T}Conf_2[\![\, \S \,]\!]\ \text{cfg} = \S$$

Figure 11 Configuror Semantics

7 Analysis

Having specified a system and given its semantics, we would now like to analyze it. The formal semantics based on CSP allows us to adapt or extend the consistency and completeness analysis provided by (static) Wright. First, we formally present these checks in our dynamic context. Then we apply them to our fault-tolerant client-server example. Finally we show how to check the equivalence of a dynamic system with a static one.

7.1 Formal Tests Definition

As a configuror describes a sequence of steady state systems, the original Wright checks can be easily adapted to the dynamic extension. In this section, we review and adapt two of the most relevant Wright checks in our example. (A complete presentation of the original tests can be found in [1,2].) We then propose a new test dealing with the configuror and the dynamic aspects of the system.

7.1.1 Connector Consistency

In Wright the connector roles specify the expected behaviors of connected components and the glue specifies the coordination of these behaviors. Thus, inconsistencies between the participants in an interaction and the coordination of the glue are detected by the following check (using the CSP process labelling operator *label:process*):

Check 1 $(C.\text{Glue} \parallel r_1{:}(C.r_1) \parallel \ldots \parallel r_n{:}(C.r_n))$ *is deadlock free.*

Another kind of inconsistency is also detectable as deadlock: if a role specification is internally inconsistent. In a complicated role specification, there may be errors that lead to a situation in which no event is possible for that participant, even if the glue were willing to take any event. This is detected by another test:

Check 2 *Each role $r_1, \ldots r_n$ of the connector C is deadlock free.*

Both checks can be directly reused in our dynamic extension. They are easily performed with the help of the FDR model checker [4]. In case of failure, the tool provides the execution traces leading to the deadlock. This information may help the user to debug his specifications.

7.1.2 Attachment Consistency

In Wright, a port is a specification of a component, as it is seen from the point of view of a single interaction. A role, on the other hand, acts has a placeholder representing the range of potential participants in the interaction described by a connector. An important check is whether the port of a component is "consistent" with respect to a role to which it is attached (see especially [2]). Specifically, the port-role consistency check must ensure that when in a situation described by the role protocol, the port must always continue the protocol in a way that the role could have. This property can be expressed with a CSP refinement check. When the role *r* of the connector *Cn* is attached to the port *p* of the component *Cp*, we must check:

Check 3 $R \sqsubseteq (P \parallel \mathrm{det}(Cn.r))$ *with* $P = Cp.p \parallel \mathrm{Stop}_{\alpha r \setminus \alpha p}$ *and* $R = Cn.r \parallel \mathrm{Stop}_{\alpha p \setminus \alpha r}$

As the role specifies the range of behavior that the component may have, and it circumscribes the behaviors over which the connector's rules are expected to apply, the check uses its deterministic version det(*Cn.r*) to constraint the port behavior. The CSP refinement operator $P \sqsubseteq Q$ requires *P* and *Q* have the same alphabet. Stop processes, in the previous check, are used to extend the port and role alphabets.

This check must be redefined in our dynamic context. First, because of reconfigurations, it may not be a port which is attached to a single role in a dynamic system, but a sequence of ports as described by the configuror. For each role, a corresponding "virtual" port must be constructed using the configuror and port definitions. If a control event *e* has several occurrences in a port (e.g., $p = \ldots\ control.e \rightarrow E_1 \ldots\ control.e \rightarrow E_2 \ldots$), we cannot select either E_1 or E_2 when *e* occurs. We must assume the component can be in any of these states and use the virtual port expression $control.e \rightarrow (E_1 \sqcap E_2)$. Second, the ports and roles use different control events, which are associated by the configuror rules. So, the configuror must appear on both sides of the refinement checks. Third, a re-configuration rule expresses the rendez-vous of several components (the re-configuration actions are performed once *all* parts are ready). To express this synchronization, the control events of the port and role, and the control events sequences of the configuror rules are bracketed by a shared event *Synchro* (see example in section 7.2). With these transformed expressions (noted as *E"*), the port-role compatibility check becomes:

Check 4 $(Cn.r'' \parallel Configuror'') \sqsubseteq (Virtual.p'' \parallel \mathrm{det}(Cn.r'') \parallel Configuror'')$

Wright provides other tests, which are easily adapted to our extension. We do not detail them here. These tests include: component consistency checks (is a port a projection of the computation?), attachment completeness checks (does an unattached port expect to inter-

act with its environment?), style checks (is a style constraint satisfied by a configuration?) and initiator checks (are the initiated-observed annotations consistent?). However, our dynamic context provides also new opportunities for checking.

7.1.3 Configuror Consistency

Consistent configurors must guarantee that when the event *new.C* occurs, *C* does not already belong to the current configuration. As the configuror is defined in CSP, the test of this property can be formally expressed as a CSP refinement check:

Check 5 $(\text{prop} = (\text{new.C} \rightarrow (\text{del.C} \rightarrow \text{prop} \,\square\, \S)) \,\square\, \S) \sqsubseteq \text{configuror}$

Similar properties dealing with attachments can be expressed and checked in the same way. Also, more configuror checks can be defined. These tests include: configuror-connector consistency (do a glue and the configuror agree on next control events?), and configuror-component consistency (do a computation and the configuror agree on next control events?).

7.2 Applications

In this section, we apply the previously-defined checks to our fault-tolerant client-server system specifications (Figures 6 and 8).

7.2.1 Connector Consistency

The *FTLink* connector consistency checks are expressed as:

Claim 1 (FTLink.Glue $\|$ c:(FTLink.c) $\|$ s:(FTLink.s)) *is deadlock free.*

Claim 2 *Roles* FTLink.c *and* FTLink.s *are deadlock free.*

Checking these properties with FDR, we discover that the connector of Figure 6 is, in fact, not deadlock-free (Claim 1 fails). Indeed, if a reconfiguration (*changeOk*) occurs after the client has sent a request (*c.request*), but before it has been transmitted to the server ($\overline{s.request}$), the connector will deadlock. On the other hand, we do not have to worry about reconfiguration when the Link is transmitting a reply back to the client. In this case we can postpone the reconfiguration until the end of the transaction.

This failure has an intuitive explanation. In a concrete implementation, if the active server goes down in the middle of a request transmission (between *c.request* and $\overline{s.request}$), the reconfiguration must be taken into account or the connector will try to send the request to the wrong server. But, if the active server goes down in the middle of a reply transmission (between *s.reply* and $\overline{c.reply}$), the *FTLink* connector can still send the reply to the client, before reconfiguring its attachments with the servers.

This specification and its analysis can help the programmer modify the original Link definition (Figure 2) to achieve a fault-tolerant one. In practice, the code dealing with reconfiguration (e.g., switching the communication channel) should not be simply inserted in the original Link definition as a new fourth case, but it must also be interleaved with the transmission of a request. A correct version of Link's Glue is detailed in Figure 12.

Finally, this example reveals a point about fault-tolerance: this kind of system requires a buffer to handle transient states. In our case, a request is stored in the connector until a stable configuration with a working server is reached (see **RequestToSend**).

Glue = c.request → ($\overline{\text{s.request}}$ → **Glue** [] control.changeOk → **RequestToSend**)
[] s.reply → $\overline{\text{c.reply}}$ → **Glue**
[] §
[] control.changeOk → **Glue**
where RequestToSend = $\overline{\text{s.request}}$ → **Glue** [] control.changeOk → **RequestToSend**

Figure 12 Dynamic Wright Specification : Deadlock Free FTLink Connector

7.2.2 Attachment Consistency

In our dynamic fault-tolerant client-server system, the *FTLink* connector is alternatively attached to a *FlakyServer* and a *SlowServer*. So, the "virtual" server port attached to *L.s* is:

Virtual.p_1 = § [] (request → $\overline{\text{reply}}$ → Virtual.p_1 ⊓ Primary.control.down → Virtual.p_2)
Virtual.p_2 = request → $\overline{\text{reply}}$ → Virtual.p_2 [] Secondary.control.off → Virtual.p_1 [] §

Then, the shared event *Synchro* is introduced in the port, role and configuror definitions. Because of space constraints we have abstracted the sequences of actions in the configuror as a single *actions* event. The resulting process definitions are:

Virtual.p_1" = § [] (request → $\overline{\text{reply}}$ → Virtual.p_1"
⊓ Synchro → Primary.control.down → Synchro → Virtual.p_2")
Virtual.p_2" = request → $\overline{\text{reply}}$ → Virtual.p_2"
[] Synchro → Secondary.control.off → Synchro → Virtual.p_1" [] §
Link.s" = (request → $\overline{\text{reply}}$ → Link.s"
⊓ Synchro → Link.control.ChangeOk → Synchro → Link.s") [] §
Configuror = actions → WaitForDown
where
WaitForDown = § [] (Synchro → Primary.control.down → Secondary.control.on →
L.control.changeOk → Synchro → actions → WaitForUp)
WaitForUp = § [] (Synchro → Primary.control.up → Secondary.control.off →
L.control.changeOk → Synchro → actions → WaitForDown)

Finally, the attachment consistency of the connector *L* role *s* is checked as:

Claim 3 (Link.s" || Configuror") ⊑ (Virtual.p_1" || Configuror" || det(Link.s"))

We do not detail here the configuror consistency checks of our simple system.

7.3 Equivalence

The previous analysis ensures that the system satisfies certain consistency properties. Another application of our semantics is to prove that a dynamic system (or sub-system) is equivalent to some steady-state one. This is a useful result because for certain purposes we can treat the dynamic system as a static one. For instance, as in the previous example, we might like to show that the reconfigurable client-server system is equivalent to a simple client-server system in which the server never goes down.

In order to compare a dynamic system with a static one, the *control* and *go* events must be hidden, the remaining communication events should be stripped of their configuration encoding part (i.e., renaming *Cp.p.e.Cn.r* into *Cp.p.e*) and some components may have to be renamed (e.g., *Primary* and *Secondary* are unified with *S*). The following check ensures that our fault-tolerant system has the functionality of a simple client-server system:

Rename(Strip(DynamicClient-Server \ {_.control._, _.go...})) = Simple

With the tool FDR [4], we discover that these two systems are nearly equivalent: they have the same traces and failures, but *Simple* never diverges, while the left-hand side system may diverge (performing no "useful" communication, but only an infinite sequence of reconfigurations). This discrepancy indicates that without some additional guarantees of fairness, the two systems are not, in fact, identical. However, although it is not possible to add fairness to our CSP description, in practice it would not be difficult to ensure this property for an implementation.

Finally we can examine the reusability of code. Our definitions of the client component are the same in the first Client-Server steady-state system (Section 3) and in the fault-tolerant Client-Server dynamic system (Section 5). We did not have to introduce control events in the client definition. So, in an implementation the same client code could be used in both steady-state and dynamic systems.

8 Future Work and Discussion

We believe that a number of technical extensions of the research are worth exploring. First, fault-tolerance introduces notions (e.g., time-out, preemption) along with idiomatic encoding (e.g., replacing the absence of event by a time-out event, or set an interruptible interpretative cycle). We think style libraries with specialized analyses and transformations might support these idioms.

Wright provides abstract specifications of software architectures. Our central configuror may not be realistic in a concrete implementation. In this case, a partial evaluation of the specifications could distribute the configuror actions in the components and connectors to be closer to real code.

Our semantics is a formal basis to further develop analysis and transformations. Intractable analysis could be replaced by proofs based on the semantic expressions and CSP. For example, a case study in [1] does not rely on model checking to study the deadlock freedom of a buffered connector style. Also, we restricted our study to dynamic systems with a finite number of configurations. The subordination CSP operator (//) can be used to dynamically duplicate processes by recursion. For example, [6] gives a definition of a factorial process where each level of recursion declares a new local process to deal with the recursive call. This technique might allow us to express semantics of dynamic architectures with regular pattern topology involving an unbounded number of configurations.

The present work focuses on protocols and deadlock freedom. Other properties should be studied in Wright. For example, [3] proposes a security analysis based on information flow in CSP expression. We think, this work could be adapted to Wright by introducing extra information specifying information flows hidden by the specifications. Finally, all interesting properties can't be expressed in CSP, but our framework and the tagging technique of our semantics could be reused with another formalism (e.g., CSP which lacks fairness must be replaced by temporal logic [8]).

In this paper, we have described an approach to architectural specification that permits the representation and analysis of dynamic architectures. The approach is based on four ideas:

- *Localization of reconfiguration behavior*, so that it is possible to understand and analyze statically what kinds of dynamic (topological) architectural changes can occur in a running system.
- *Uniform representation of reconfiguration behavior and steady state behavior*, so that interactions between the two can be analyzed.
- *Clearly delimited interactions between the two kinds of behavior through the use of control events*, so that one can explicitly identify the points in the steady state behavior at which reconfigurations are permitted.
- *Semantic foundations based on CSP*, so that we can exploit traditional tools and analytic techniques based on process algebras.

To make this approach work we have limited ourselves to dynamic architectures that have a finite number of possible reconfigurations. Thus we have made a tradeoff between generality and power: by considering a restricted set of dynamic architectures we are able to provide strong support for static reasoning and automated analysis.

Because of space constraints, we illustrated the approach with a very simple example, showing how specification and analysis could help us detect and then fix a bug in the description. However we are confident that our extension scales up to realistic systems, as static Wright does [1]. Indeed, our checks remain local, and identical checks can be shared in systems with a high degree of symmetry (e.g. multiple instances of the same component). Further study would be necessary.

References

[1] R. J. Allen. *A Formal Approach to Software Architecture*. Ph.D. Thesis, Carnegie Mellon University, School of Computer Science, May 1997.

[2] R. J. Allen and D. Garlan. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] J.-P. Banâtre, C. Bryce and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proc. of ESORICS'95*, LNCS 875, 1995.

[4] *Failures Divergence Refinement: User Manual and Tutorial*. Formal Systems (Europe) Ltd., Oxford, England,1.2β edition October 1992.

[5] D. Garlan, R. T. Monroe and D. Wile. ACME: An Architecture Description Interchange Language. *Submitted for publication*, January 1997.

[6] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice Hall, 1985.

[7] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Trans. SW Eng.*, 21(4), 95.

[8] L. Lamport. The Temporal Logic of Actions. In *ACM TOPLAS*, 16(3), 1994.

[9] D. Le Métayer. Software Architecture Styles as Graph Grammars. In *Proc. of FSE'96*, ACM SIGSOFT.

[10] D. Luckham, L. Augustin, J. Kenney, J. Vera, D. Bryan and W. Mann. Specification and Analysis of System Architecture Using Rapide. In *IEEE Trans. on Soft. Eng.*, 21(4), 1995.

[11] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of FSE'96*, 1996.

[12] N. Medvidovic.A Classification and Comparison Framework for Software ADLs. *Univ. of Irvine, Dept. of Information and Computer Science*, 1997.

[13] L. Sha, R. Ragunathan and M. Gagliardi. Evolving Dependable Real-Time Systems. *Carnegie Mellon University SEI Report* CMU-SEI-95-TR-005, 1995.

[14] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. In *IEEE Trans. on Soft. Eng.*, 21(4), 1995.