Specifying Dynamism in Software Architectures *

Robert Allen
Rémi Douence
David Garlan

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pensylvania, 15213-3890
Email: douence@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.

Keywords: Software Architecture, Wright, Dynamic Topology, Analysis, CSP

*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.

1 Introduction

Recently, there has been considerable progress on the development of architecture description languages (ADLs [Med97]) 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 [All97] [AG97], UniCon [SDK+95], Rapide [LAK+95], Darwin [MK96] and ACME [GMW].

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 tend to focus on one or more of these aspects.

In this position 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 the remainder of 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, it must be shown 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 short paper we will sketch the basic idea using the Wright ADL as our notational basis. We first illustrate the problem. Then we show how a language originally designed for steady-state architectures, such as Wright, can be extended to handle dynamic aspects of architecture modeling and analysis. Next, we illustrate the kinds of analysis that such a formalism supports. Then we briefly sketch the semantic model on which the approach is based.

2 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 a steady-state ADL such as Wright. A description would capture the structure of the architecture, describing the topology of

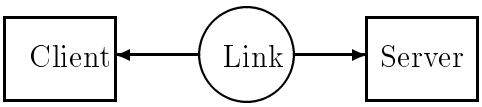

Figure 1: Static Topology: Simple Client-Server

the system and its composition, as well as the behavior of the system as a whole. Each component description provides a high-level specification of its functionality and interfaces, while the connector specification indicates how the "link" pattern of interaction combines the behaviors of a client and a server. In this case, the client makes a request, which is received by the server, and the server provides a response, which is communicated by the client. This sequence of actions can be repeated many times.

The Wright specification shown in Figure 2 shows this interaction pattern, and further provides important details about the rules of the interaction. As detailed elsewhere [All97] [AG97], Wright uses a variant of CSP [Hoa85] to characterize architectural behavior. The essential idea is to treat both components and connectors as processes, which synchronize over suitably renamed alphabets. For example, the use of internal choice ($\sqcap$) in the client specification indicates that it is the client that decides whether it will make a request. The use of external choice ($\Box$) in the server specification indicates that the server is expected to respond to any number of requests, and may not terminate prematurely. Successful termination is indicated by §. We have described in [All97] [AG97] how these formal descriptions can be used to check the consistency and completeness of architectural descriptions.

Note, however, that the server role will never 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 functionality, 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).

3 Simulating Dynamism

Consider one possible solution, in which there are two servers, one 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 primary server goes down, the client uses the secondary server until such time as the primary server returns to service. The topology of the system is shown in Figure 3, while a Wright description of a possible client is shown in Figure 4. In this description the server communicates with "*serverDown*" and "*serverUp*" when it is about to go down or come up (resp). (In practise, such events might be supplied by a time-out service.)

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

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

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

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

Configuration Client-Server
Instances
C : Client ; L : Link ; S : Server
Attachments
C.p as L.c ; S.p as L.s
End Configuration

Figure 2: Static Wright Specification: Simple Client-Server

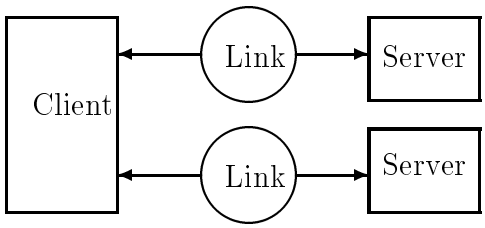

Figure 3: Static Topology: Fault Tolerant Client-Server

Component Client

Port $Primary = UseConnect \sqcap \S$

where

$UseConnect = \overline{request} \rightarrow reply \rightarrow Primary$

$\quad \Box\ serverDown \rightarrow (serverUp \rightarrow Primary \ \Box\ \S)$

Port $Secondary = (\overline{request} \rightarrow reply \rightarrow Secondary) \ \Box\ \S$

Computation $= UsePrimary$

where

$UsePrimary = internalCompute \rightarrow (TryPrimary \ \Box\ \S)$

$TryPrimary = \overline{primary.request} \rightarrow primary.reply \rightarrow UsePrimary$

$\quad \Box\ primary.serverDown \rightarrow TrySecondary$

$UseSecondary = internalCompute \rightarrow (TrySecondary \ \Box\ \S)$

$TrySecondary = \overline{secondary.request} \rightarrow secondary.reply \rightarrow UseSecondary$

$\quad \Box\ primary.serverUp \rightarrow TryPrimary$

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

disadvantages:

1. The simple client-server functional pattern has been lost. In particular, its specification is now muddied by the need to consider the effects reconfiguration at almost each step.

2. It has been necessary to significantly alter the client in order to accommodate a change that should occur on the server's side. Ideally, the client should be able to continue to operate as before, but have the *system* handle rerouting of requests to the new server.

3. Changes to this system (e.g., adding a third backup, or permitting only the primary to go down once before abandoning it) require 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 the client components "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.

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) what re-configuration is triggered by each event.

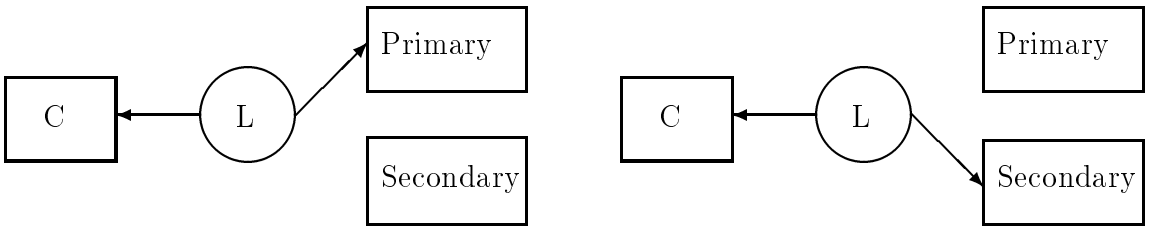

Figure 5: Dynamic Topology: Alternating Configurations of a Fault Tolerant Client-Server System

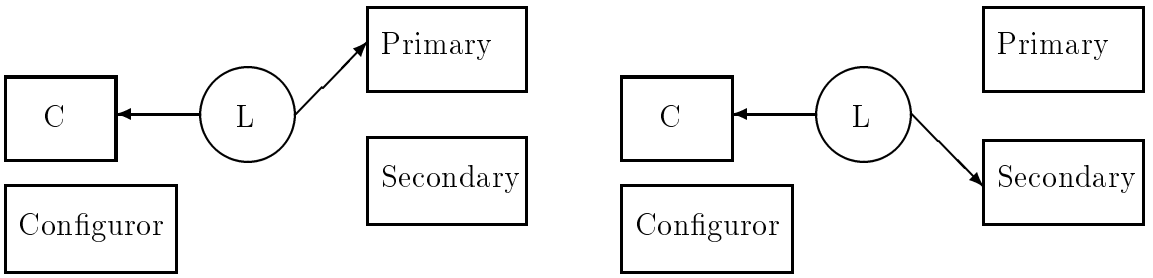

Figure 6: Dynamic Topology: Alternating Configurations of a Fault Tolerant Client-Server System

4 Our Approach

Our solution consists of introducing special "control" events, into a component's alphabet, and permitting them to be used 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. These control events are then used in a separate view of the architecture, the configuration program, which describes how these events trigger reconfigurations.

To illustrate, the configurations pictured in Figure 5 are triggered (as before) by the "*serverDown*" and "*serverUp*" events, now explicitly marked as "control" rather than "communication" events. The changes to the architectural topology are achieved using sequences of "*new*", "*del*", "*attach*", and "*detach*" actions, as defined by the reconfiguration program, "*Configuror*", as illustrated in Figure 6 and Figure 7.

This reconfiguration program is responsible for describing change of the architecture in terms of the architecture types (e.g., "*Client*", "*FlakyServer*", "*SlowServer*") declared in the usual way. In this example, the "*Client*" component type is as in Figure 2. The primary server (a component of type "*FlakyServer*", not shown here) indicates the states in which it may go down, and the link and secondary server indicate the states in which they can be reconfigured. For example, Figure 8 shows how a "*FaultTolerantLink*" might indicate its possible reconfigurations. In this description, the control event "*ChangeOK*" indicates the states at which the link is prepared to accept reconfiguration.

Configuror Simple

$new.C : Client$
$\rightarrow new.Primary : FlakyServer$
$\rightarrow new.Secondary : SlowServer$
$\rightarrow new.L : FaultTolerantLink$
$\rightarrow attach.C.p.to.L.c$
$\rightarrow attach.Primary.p.to.L.s \rightarrow WaitForDown$

where

$WaitForDown = Primary.control.serverDown \rightarrow detach.Primary.p.from.L.s$
$\rightarrow attach.Secondary.p.to.L.s \rightarrow WaitForUp$
$\square$ §
$WaitForUp = Primary.control.serverUp \rightarrow detach.Secondary.p.from.L.s$
$\rightarrow attach.Primary.p.to.L.s \rightarrow WaitForDown$
$\square$ §

Figure 7: Dynamic Wright Specification: Configuror (Fault Tolerant Client-Server System)

Connector FaultTolerantLink

Role $c = \overline{request} \rightarrow reply \rightarrow c \ \square$ §
Role $s = (request \rightarrow \overline{reply} \rightarrow s \ \square \ control.ChangeOK \rightarrow s) \ \square$ §
Glue $= c.request \rightarrow \overline{s.request} \rightarrow$ **Glue**
$\square \ s.reply \rightarrow \overline{c.reply} \rightarrow$ **Glue**
$\square$ §
$\square \ control.ChangeOK \rightarrow$ **Glue**

Figure 8: Dynamic Wright Specification: Link Connector

$$
\begin{aligned}
\textbf{Glue} = {} & c.request \rightarrow (\overline{s.request} \rightarrow \textbf{Glue} \;\Box\; control.ChangeOK \rightarrow GlueRequestToSend) \\
& \Box\; s.reply \rightarrow \overline{c.reply} \rightarrow \textbf{Glue} \\
& \Box\; \S \\
& \Box\; control.ChangeOK \rightarrow \textbf{Glue} \\
& \textbf{where} \\
& GlueRequestToSend = (\overline{s.request} \rightarrow \textbf{Glue}) \;\Box\; control.ChangeOK \rightarrow GlueRequestToSend
\end{aligned}
$$

Figure 9: Dynamic Wright Specification: Deadlock Free Link Connector

5 Analysis

Having specified the system, we would now like to analyze it. Although there are many things that might in principle be checked, one particularly important property is whether the link connector that we defined above is consistent. By "consistent" we mean whether it describes a protocol that is deadlock-free in the presence of reconfigurations carried out by the "*Configuror*".

We can check this property directly using Wright's tools for consistency checking. When we do this we discover that the connector is, in fact, not deadlock-free. Indeed, if a reconfiguration ("*control.ChangeOK*") occurs after the client has sent a request ("*c.request*"), but before it has been transmitted to the server ("*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 its treatment until the end of the transaction.

These results have an intuitive explanation. In a concrete implementation, if the active server goes down in the middle of a request transmission (between "*c.request*" and "*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 "*c.reply*"), the Link 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 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 9.

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

6 Semantics

Thus far we have relied on the reader's good faith that the notation outlined above actually makes sense, and that the analytic techniques are sound. In this section we sketch the formal basis for

this.

The basic idea is to translate the notation into pure CSP [Hoa85]. 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. Second in the translation we "tag" events with the configuration in which that event occurs. The effect of a reconfiguration action, is to cause the tags to change so that the interactions occur as defined by the new configuration. Thus an event, e, in one configuration might end up 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, when the Link connector interacts with the Primary server, the reply case of L is defined as:

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

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,... and its port p_n attached to the role r_n of Cn_n. For example, the connector L is compiled to:

$L = (L.go.c.C.p.s.Primary.p \rightarrow Glue_1) \,\Box\, (L.go.c.C.p.s.Secondary.p \rightarrow Glue_2)$
$Glue_1 = L.c.request.C.p \rightarrow (\overline{L.s.request.Primary.p} \rightarrow Glue_1 \,\Box\, \ldots$
$Glue_2 = L.c.request.C.p \rightarrow (\overline{L.s.request.Secondary.p} \rightarrow Glue_2 \,\Box\, \ldots$

Finally, the configuror definition is transformed into a CSP process too, where the "*new*", "*del*", "*attach*" and "*detach*" actions are compiled into the previous $Cp.go\ldots$ events that select the proper configuration of the components at each reconfiguration step. For example, the following configuror portion: $new.L \rightarrow Attach.C.p.to.L.c \rightarrow Attach.Primary.p.to.L.s \rightarrow \ldots$ is transformed to $L.go.c.C.p.s.Primary.p \rightarrow \ldots$

The formal semantics based on CSP allows us to adapt or extend the consistency and completeness analysis provided by Wright.

Most of the Wright checks can be directly adapted to the dynamic context. For example, the deadlock checks for connectors remain the same, but the terminating success events $\surd$ must be replaced by a control event triggering a connector deletion.

The port-role consistency test of Wright must be redefined in our dynamic context. Indeed, because of reconfiguration, it isn't a port which is attached to a role in a dynamic system, but a sequence of ports. For example, our previous FaultTolerantLink connector is alternatively attached to a FlakyServer and a SlowServer. A "virtual" port defined by the sequence of ports attached to a connector role can easily be constructed with the help of the configuror.

Dynamic systems provide new opportunities for checking. For example, consistent configurors should guarantee that when the event $new.C$ occurs, C must not already belong to the current configuration. As the configuror is defined in CSP, the test of this property can be formally expressed as CSP refinement checks:

$(prop = new.C \rightarrow (del.C \rightarrow prop \,\Box\, \S)) \sqsubseteq configuror$

Similar properties dealing with attachments can be expressed and checked in the same way.

The previous analysis ensures that the complete system is consistent. Another application of our semantics is to prove that a dynamic system (or sub-system) is equivalent to a steady-state one. For example, we could show that the two alternating servers can be merged in a bigger component with only one port, or that an extra port could be added to the client component to make the attachments static as in section 3.

Finally, we can examine the code reusability. For example, our definitions of the client component are the same in the first Client-Server steady-state system (Section 2.) and in the FaultTolerant Client-Server dynamic system (Section 4.). We did not have to introduce control events in the client definition. So, in a real implementation the same client code could be used in both steady-state and dynamic systems.

7 Summary and Discussion

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

1. *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.
2. *Uniform representation of reconfiguration behavior and steady state behavior*, so that interactions between the two can be analyzed.
3. *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.
4. *Semantic foundations based on CSP*, so that we can exploit traditional tools and analytic techniques based on process algebras.

We illustrated the approach with a simple example, to show how specification and analysis could help us detect and then fix a bug in the description. We argued briefly that such a bug is one that might well occur in a naive implementation of such a system.

The approach we have taken differs in several fundamental ways from those taken by other researchers. The most obvious, and perhaps most contentious, is the use of CSP as the underlying semantic model. In choosing CSP we have made an important trade-off: we are willing to reduce the class of dynamic systems that we can describe, in exchange for a formalism that is amenable to deep static analysis. In particular, the class of system excludes systems that inherently create an

infinite number of configurations during run time. (This rules out, for example, algorithms such as the Prime Sieve in which a new filtering process is created for each prime number).

On the other hand, we are able to account for the large (and we would argue, central) class of system where the collection of possible topologies is effectively bounded and known at system definition. This includes a broad class of client-server systems and many fault tolerant architectures (such as Simplex [RD95]).

We believe that this trade-off is worth making, and that we have identified an important place in the design space of architectural formalization. However, at present this is largely a conjecture. On-going research will hopefully shed further light on the strengths and weakness of the approach that we are advocating.

In addition to gaining further experience with the approach, there are a number of technical extensions that are worth exploring. First, we think the relationship between Wright specifications and the concrete implementations should be studied (e.g. automatic code skeleton generation, or conversely reverse ingineering extracting a Wright specification). Second, fault tolerance introduces notions (e.g. timeout, preemption) along with idiomatic encoding (replace the absence of event by a timeout event, set an interruptible interpretative cycle). Wright could be specialized, so that libraries of style with specialized transformations and analysis provide these idioms. Finally, all interesting properties can't be expressed in CSP. Our framework could be reused by replacing CSP by another formalism (e.g. temporal logic).

References

[AG97] R.J. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 1997.

[All97] R.J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1997.

[GMW] D. Garlan, R.T. Monroe, and D. Wile. Acme: An architecture description interchange language. Submitted for publication.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[LAK+95] D.C. Luckham, L.M. Augustin, J.J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 1995.

[Med97] N. Medvidovic. A classification and comparison framework for software architecture description languages. Technical report, University of California, Irvine, Department of Information and Computer Science, 1997.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[RD95] J.G. Rivera and A.A. Danylyszyn. Formalizing the uni-processor simplex architecture. Technical report, Carnegie Mellon University School of Computer Science, 1995.

[SDK+95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 1995.

8 Biography

Robert Allen recently completed his Ph.D. in Computer Science at Carnegie Mellon University. He has authored numerous papers on software architecture, and was the primary designer for the Wright Architectural Description Language. He is currently employed by IBM Microelectronics Division as a Software Engineer.

Rèmi Douence is an Inria post-doc student with David Garlan at Carnegie Mellon University. He was formerly involved in functional languages: his Ph.D. thesis (october 1997, Irisa Rennes, France) formally describes and compares the functional languages compilation techniques. He now focuses on applying formal methods to applied domains. He is mainly interested in declarative and high-level programming languages (semantics, implementation, analysis...), specifications, prototyping, debugging and teaching.

David Garlan is an Associate Professor in the Department of Computer Science at Carnegie Mellon University, where he heads the ABLE Project. Dr. Garlan's research interests include software architecture, formal methods, and software development environments. He recently co-authored (with Mary Shaw) the book "Software Architecture: Perspectives on an Emerging Discipline."