Architecture-driven Modelling and Analysis[1]

David Garlan and Bradley Schmerl

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213 USA

{garlan,schmerl}@cs.cmu.edu

Abstract

Over the past 15 years there has been increasing recognition that careful attention to the design of a system's software architecture is critical to satisfying its requirements for quality attributes such as performance, security, and dependability. As a consequence, during this period the field of software architecture has matured significantly. However, current practices of software architecture rely on relatively informal methods, limiting the potential for fully exploiting architectural designs to gain insight and improve the quality of the resulting system. In this paper we draw from a variety of research results to illustrate how formal approaches to software architecture can lead to enhancements in software quality, including improved clarity of design, support for analysis, and assurance that implementations conform to their intended architecture.

Keywords: Software Architecture, Architecture Analysis

1 Introduction

Software architecture is concerned with the high-level structures of a software system, the relationships among them, and their properties of interest. These high-level structures represent the loci of computation, communication, and implementation. Typical properties include emergent behaviour, such as the performance, reliability, security, maintainability, and so on (Garlan and Shaw 1996, Perry and Wolf, 1992).

Well designed architectures typically allow one to reason about satisfaction of key requirements and to make principled engineering tradeoffs. They can provide clear rationale of assignment of function to components, establish principles of conceptual integrity (Brookes 1975), and lead to considerable reduction in rework over the lifespan of a system (Boehm and Turner 1993). They can also permit reuse of architectural design idioms and patterns, reduction of development costs through product line approaches, and guidance to future maintainers of those systems.

Given the potential benefits of software architecture over the past decade and a half the field has received increasing attention and consequent progress. There are now numerous textbooks (Bass, Clements, and Kazman 2003, Rosanski and Woods 2005), review methods (Clements, Kazman, and Klein 2003), conferences (e.g., the Working IEEE/IFIP Conferences on Software Architecture and the European Workshops on Software Architecture), documentation standards (Clements et al 2002, IEEE 2000), handbooks (Buschmann et al. 1996), and courses covering the topic. Success stories detailing the economic benefits and practice of product lines abound (Bosch 2000 and Clements and Northrop 2001). Software development practices typically now incorporate architecture reviews, and software architects have titles and well-defined roles in many organizations.

Coupled with heightened awareness, and increasing maturity of practice, a number of standards bodies are now promoting notations and standards for software architecture. UML 2.0 from the Object Management Group, for example, now has improved capabilities to represent general component and connector architectures. The IEEE prescribes a meta-framework for architectural views. Some standards aim at more specific domains, such as resource constrained systems (e.g., AADL by SAE International, SysML by the Object Management Group). Other standards-based approaches, like "model driven architecture" from the Object Management Group, attempt to provide ways to move from architectural models to architecturally-consistent implementations. Finally, the presence of middleware and their corresponding architectural frameworks have led to considerable standardization and reuse within certain application domains, e.g., J2EE, Eclipse, ADO.NET.

However, despite notable progress and concern for ways to represent and use software architecture, specification of architectural designs remains relatively informal, relying on graphical notations with weak or non-existent semantics that are often limited to expressing only the basic of structural properties. As a consequence, it is almost impossible using today's common practices to (a) express architectural descriptions precisely and unambiguously; (b) provide soundness criteria and tools to check consistency of architectural designs; (c) analyse those designs to determine implied system properties; (d) exploit patterns and styles, and check whether a given architecture conforms to a given pattern; and (e) guarantee that the implementation of a system is consistent with its architectural design.

Luckily, however, research has developed techniques to address all of these shortcomings by providing more-formal approaches to architectural design. While these techniques may not be completely ready for full-scale adoption by industry, many of them are close to that level of maturity.

[1] This paper accompanies the keynote talk "Software Architecture for Highly Dependable Systems" by David Garlan.

In this paper we outline several such techniques and their associated tools, drawing particularly from research carried out at Carnegie Mellon University in the ABLE Project. While not a comprehensive survey of existing work on formal approaches to software architecture, this paper will give a flavour for the kinds of techniques being investigated by the research community, and the kinds of potential benefits that they bring to the field.

The remainder of this paper is organized as follows: Section 2 summarized how to specify architectural structure; in Section 3 we introduce architectural properties and illustrate how a flexible property mechanism can facilitate powerful architectural analysis; Section 4 shows how architectural behaviour can be specified; Section 5 introduces the concept of architectural style, and shows how this concept can be used to provide domain-specific architectural models; Section 6 presents a summary of our approaches to answering the question of implementation conformance to an architecture; finally, Sections 7 and 8 present related work and conclusions.

2 Modelling Architectural Structure

The starting point for any formal treatment of software architecture is the representation of architectural structure. However, this raises the question: what kinds of structure? Any complex software system may have many structures of interest: modules, run-time entities, development teams, physical devices and networks. Today we understand that the preferred way of addressing this complexity is to recognize that an architectural design must be described in terms of a number of distinct (but related) views. Each view represents an architectural perspective on the system, exposing certain system structures and their properties, to address a related set of concerns.

Following the approach of Clements et. al (2002), one can categorize the kinds of structures into three general categories. First, there are coding structures, such as modules, packages, and classes, with relationships like uses, depends-on, inherits, etc. Second, there are run-time structures: databases, clients, servers, and connectors indicating communication pathways. Third, there are allocation structures, which map elements of the first two views into non-software entities, such as the physical setting (networks, CPUs, etc) or development teams. These lead to allocation views, such as deployment views or work breakdown structures.

In this paper we will focus on modelling and analysis of run-time structures, or **component and connector** (C&C) views. This is because such structures are the ones that most directly convey critical properties related to dependability, such as reliability, security, and performance. These are also the class of views that are least well supported by existing notations and tools.

2.1 Components, connectors, and systems

We model a run-time C&C view of software architecture as a graph of components and connectors. Specifically, basic elements and relations of a C&C view are:

- **Components** model the principle computational elements of a system's run-time structure. They include things like databases, clients, servers, GUI's, etc. Each component has a set of **ports,** which model the run-time interfaces of that component, through which it interacts with other components (via connectors). For example, a server might have a number of service invocation ports, representing the set of run-time interactions with different clients.
- **Connectors** model the pathways of communication between components. They include things like pipes and client-server communication links. Connectors may be binary, such as pipes and client-server interactions, or N-ary, such as a publish-subscribe connector, which allows publisher component to interact with zero, one, or many subscribing components. Each connector has a set of r**oles,** which model the specifications of behaviour required of the components that use a given connector. For example, a pipe might have a reading and a writing role, while a publish-subscribe connector would have publish and subscribe roles.
- **Systems** model a graph of components and connectors in which the ports of a component fill the roles of a set of connectors to determine the interconnection topology.

Figure 1 illustrates these concepts. In addition, a component or a connector may have substructure, called a **representation** that further elaborates its internal structure.

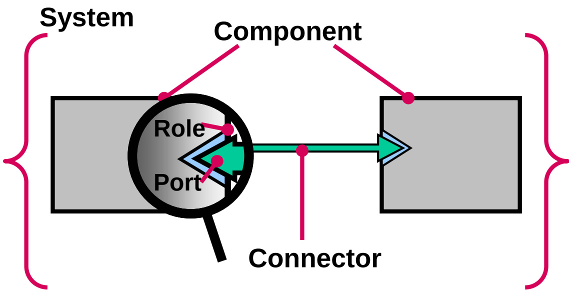


Figure 1. Component and Connector View.

This vocabulary allows one to model the box-and-and-line diagrams common to architectural descriptions, and generally corresponds to the primitive conceptual building blocks in most architectural description languages (ADLs). It is important to note, however, that unlike many informal diagrammatic depictions of architecture, the model indicated above explicitly identifies component interfaces, and represents connectors as first class model elements of the software architecture.

2.2 Acme

In order to support analysis of component and connector architecture models, it is necessary to have a machine-processable representation. In this paper we use the Acme ADL for this representation (Garlan et al. 2000).

Figure 2 shows the Acme specification of a simple client-server system consisting of a single client and a single server, interacting through a remote procedure-based

```
System simple-cs = {
  Component client = { port call-rpc; };
  Component server = { port rpc-request; };
  Connector rpc = {
    role client-side;
    role server-side;
  };
  Attachments = {
    client.call-rpc to rpc.client-side;
    server.rpc-request to rpc.server-side;
  }
}
```

Figure 2. Acme description for a simple client-server architecture.

connector. The system, named *simple-cs*, is declared in the first line of the specification. Following this are declarations of the two components, *client* and *server*, each with a single port (*call-rpc* and *rpc-request*, respectively). The connector, *rpc*, is declared to have two roles (*client-side* and *server-side*). Finally, the system is created by attaching the appropriate ports to the respective roles of the connector.[2]

2.3 Analysis of architectural structure

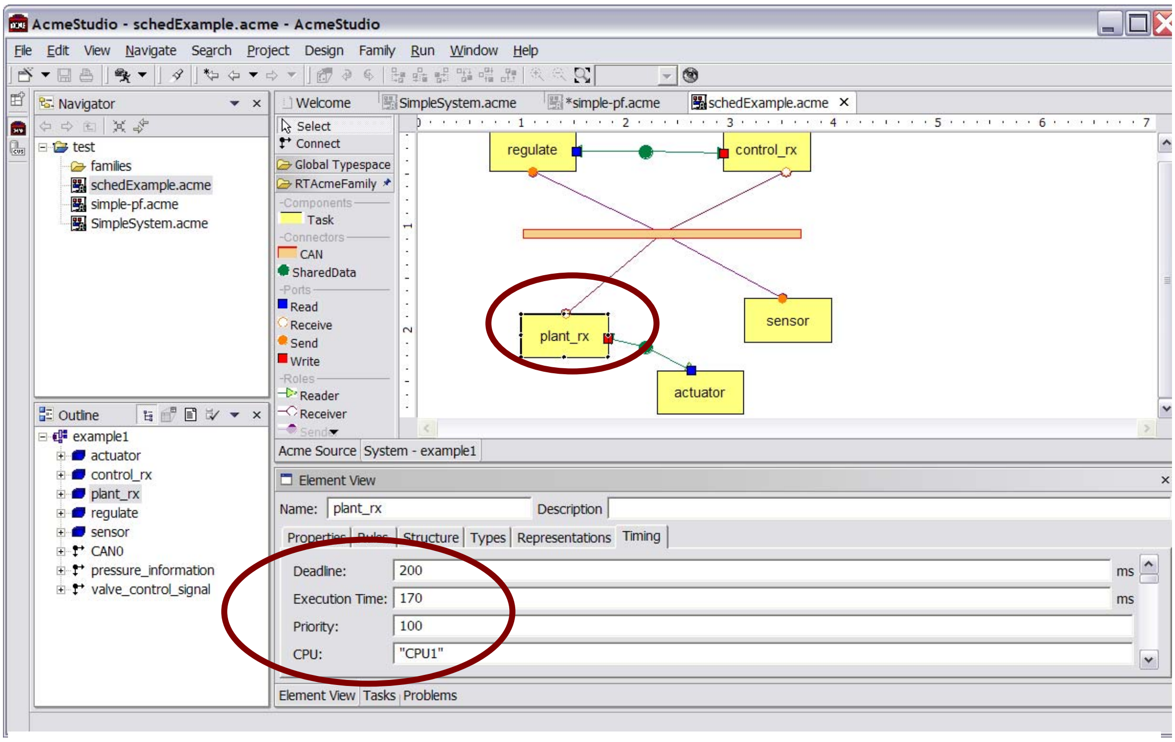
```
System simple-cs = {
  …
  Component server = {
          port rpc-request = {
            Property sync-requests : boolean
                                   = true;
    };
    Property max-transactions-per-sec : int = 5;
    Property max-clients-supported : int = 100;
  };
  Connector rpc = { …
    Property protocol : string = “aix-rpc”;
  }; …
};
```

Figure 3. Properties in Acme.

The textual representation of a graphical picture does little more than provide an alternative depiction. But there are, nonetheless, opportunities for analysis even with such simple models. For example, after parsing, we might check the model to determine whether any connectors have unattached roles, or whether every port of a component is attached to some connector. We can also check for naming conflicts (e.g., two ports of the same name on the same component).

3 Modelling Architectural Properties

While some analyses of pure structure are possible, to achieve significant analytic value from an architectural model we need to represent more of the semantics of the architecture. In Acme this is done by annotating the structure with **properties**.



Name: plant_rx

Deadline: 200 ms

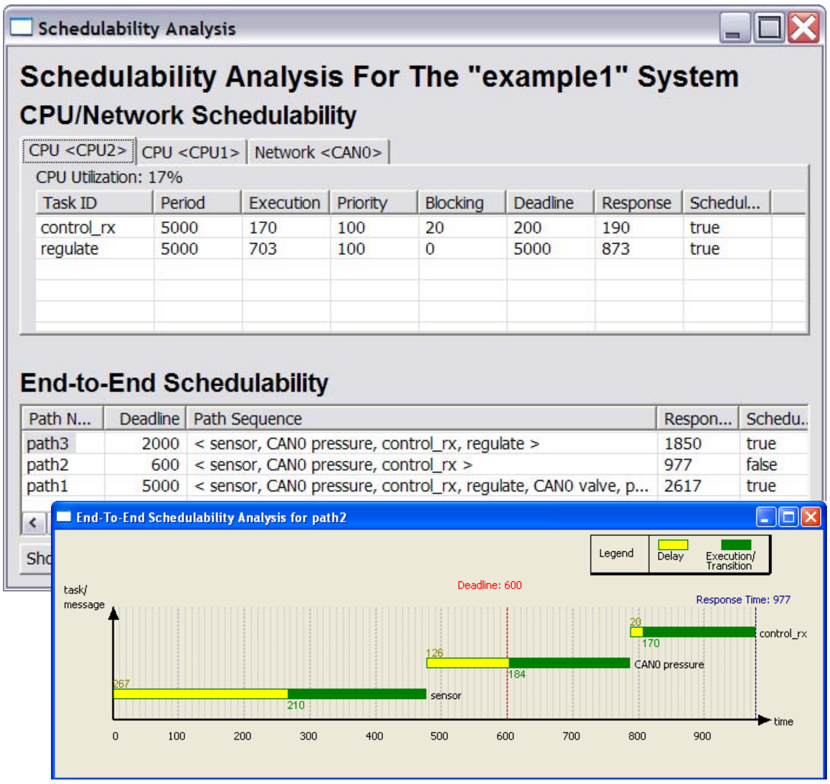
Execution Time: 170 ms

Priority: 100

CPU: "CPU1"

Figure 4. Specifying schedulabiluty properties in AcmeStudio.

[2] Although we don’t illustrate it in this simple example, at this structural level we could also provide representations of the client and server, elaborating each component’s architectural substructure. See Garlan et al. 2000 for details.



Schedulability Analysis For The "example1" System

CPU/Network Schedulability

CPU Utilization: 17%

Task ID	Period	Execution	Priority	Blocking	Deadline	Response	Schedul...
control_rx	5000	170	100	20	200	190	true
regulate	5000	703	100	0	5000	873	true

End-to-End Schedulability

Path N...	Deadline	Path Sequence	Respon...	Schedu..
path3	2000	< sensor, CAN0 pressure, control_rx, regulate >	1850	true
path2	600	< sensor, CAN0 pressure, control_rx >	977	false
path1	5000	< sensor, CAN0 pressure, control_rx, regulate, CAN0 valve, p...	2617	true

Figure 5. The results of the schedulability analysis.

3.1 Properties in Acme

Properties are simply typed name-value pairs that can be associated with any architectural element.[3] Types may be primitive (integer, boolean, etc.) or composite (sets, sequences and records).

Figure 3 illustrates the use of properties, elaborating Figure 2. This example illustrates properties associated with a port (indicating whether the client request is synchronous); a component (indicating the maximum number of transactions per second supported by the server), and a connector (indicating the name of the protocol that is expected to be used over it).

3.2 Analysing Architectural Properties

The meaning of properties is not specified in Acme, which does not provide native support for their analysis. However, such properties can be used by analysis tools to gain insight into the architecture by calculating global system properties from local properties of components and connectors using off-the-shelf analysis theory and algorithms. Such analyses can be a powerful aid to architectural design, allowing architects to identify design errors early in the process, helping the architect document the expected run-time properties of architectural elements, and facilitating tool support for providing feedback and comparisons of analysis.

We now illustrate these ideas with three examples: rate-monotonic analysis for automotive control systems, queueing theory-based analysis for detecting server overloads, and Monte Carlo style security simulation.

Example 1: Analysis of Real-time Schedulability,

Figure 4 illustrates a simple automotive system represented in AcmeStudio (Schmerl and Garlan 2004), a framework for creating architecture design environments. AcmeStudio, written as a plug-in to the Eclipse framework, permits one to define domain-specific architectural styles[4] and link in analysis tools that may be invoked by the user to analyse systems in those styles.

The style used in Figure 4 includes components that run as periodic tasks on a set of CPUs, and communicate directly with tasks on the same CPU, and with other CPUs using an automotive standard communication bus (here a CAN bus). An important question in the design of such systems is whether certain task scenarios (treated as paths through the architecture), can be scheduled on the available processors.

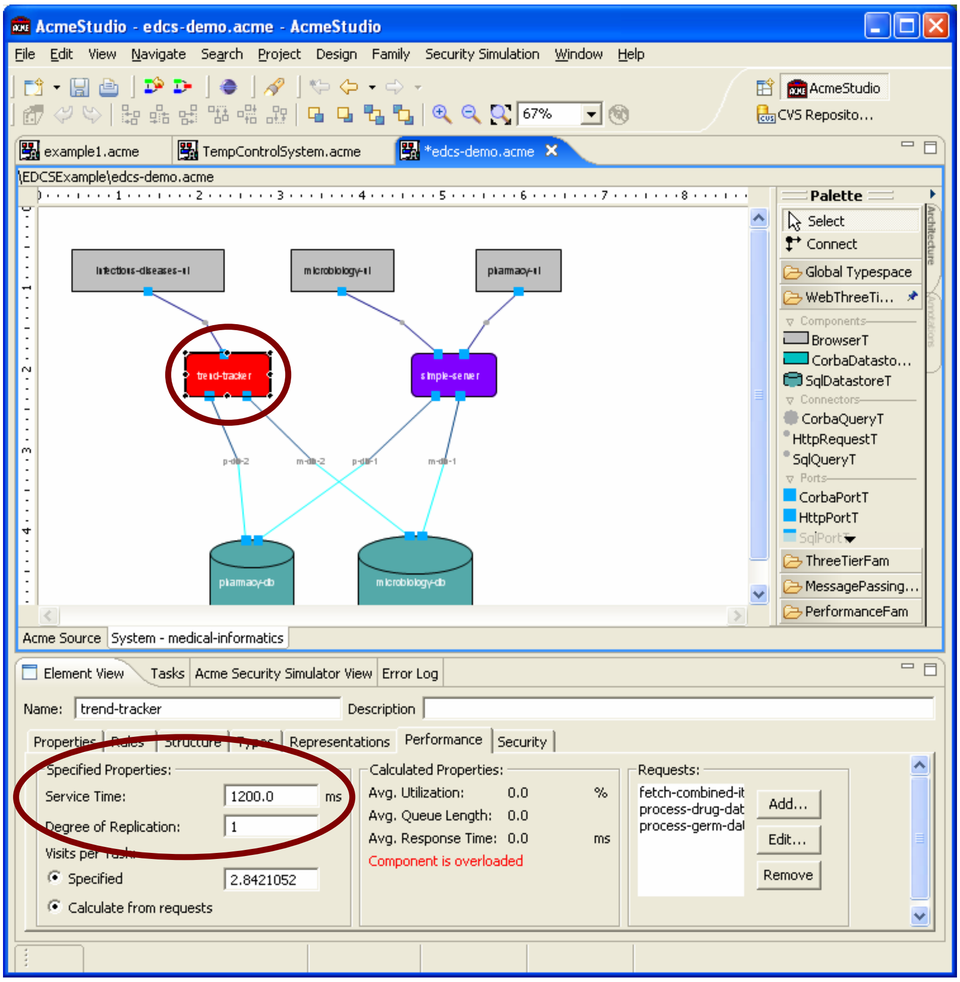
To evaluate this system-wide property, the style associates with each component a set of properties relevant to real-time schedulability. In the architectural style of this example these properties are modelled as its deadline, execution time, priority, and CPU. For example in Figure 4 the selected component, plant-rx, has values of 200, 170, 100, and CPU1 as its respective property values.

When all components have been annotated with these properties (and the connectors with similar properties), we can invoke a tool to evaluate the CPU utilization, and the schedulability of specific pathways. In the figure three pathways are specified. The resulting analysis prints out the results of applying rate monotonic analysis (Sha and Goodenough 1991), indicating which paths are schedulable (Figure 5).

It is important to note that the actual analysis of schedulability is carried out using completely standard, off-the-shelf algorithms for rate-monotonic real-time analysis. Moreover, AcmeStudio makes it relatively easy to add such an analysis using its own architecture analysis "plug-in" framework, which assists with creating specialized property editors (e.g., to specify pathways for

[3] We use the term architectural element to refer generally to components, connectors, ports, roles, representations, and systems.

[4] We discuss architectural styles in detail in Section 5; for now, consider a style as providing element types specifying the properties that must be defined for instances of the elements.



Name: trend-tracker

Description

Specified Properties:

Service Time: 1200.0 ms

Degree of Replication: 1

Visits per Task:

● Specified 2.8421052

● Calculate from requests

Calculated Properties:

Avg. Utilization: 0.0 %

Avg. Queue Length: 0.0

Avg. Response Time: 0.0 ms

Component is overloaded

Requests:

fetch-combined-it
process-drug-dat
process-germ-dal

Figure 6. Performance Analysis in AcmeStudio.

evaluation), invoking the tool through menus, passing the relevant data to it for analysis, and displaying the results back in the graphical editing environment.

Example 2: Analysis of server-load.

Of course, not all systems in need of performance analysis are real-time systems. To illustrate how the same general ideas can be supported for different application domains, consider Figure 6. Here we have an example of a system defined in a tiered style in which clients queue requests for database service at a set of servers that contain business logic to access a set of databases. The system model is shown in AcmeStudio specialized for a style that supports such systems.

To analyse performance of this system we take advantage of queuing theory to evaluate performance characteristics of such systems (Di Marco and Inverardi 2004, Spitznagel and Garlan 1998). To perform the analysis, we must first supply the values of a set of properties of the components and connectors, such as arrival rates (expressed as probability distributions), average service time for handing requests at a server, and degree of server replication. These properties are specified through a UI plugin to AcmeStudio specific to the performance analysis, at the bottom of Figure 6.

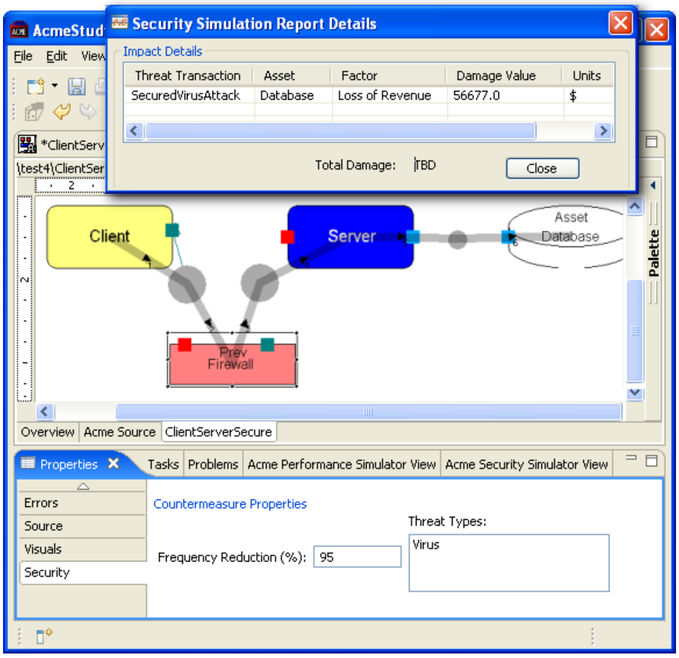
Once these kinds of properties have been defined, as before we can pass the model to an analysis tool, which in this case calculates for each server a set of derived properties, including average server utilization, queue length, response time. From these it can further indicate whether any servers are overloaded. In Figure 6, the analysis has determined that the circled component in the diagram is overloaded, and has changed its display to be red.

Example 3: Analysis of security

It is also possible to analyse the security of a system through architectural simulation. The Security Simulator plugin to AcmeStudio enables an architect to perform security simulations based on threat scenarios that she anticipates will affect the system. The main concepts in the security analysis are threat types, assets, and countermeasures; the simulation is based on the approach outlined in Butler (2002).

Threat types specify the possible threats that can affect the system, e.g., a virus or denial of service. Because different systems may be subject to different types of threats, the architect must specify each of the threat types that may be posed to the system.

Assets are components that may be damaged by particular threats. Assets are assigned a monetary value, and the particular threat types that may affect the asset are specified. For example, a database component may not be susceptible to password sniffing attacks, but may be vulnerable to data corruption as the result of a virus.



Security Simulation Report Details

Impact Details

Threat Transaction	Asset	Factor	Damage Value	Units
SecuredVirusAttack	Database	Loss of Revenue	56677.0	$

Total Damage: TBD

Countermeasure Properties

Frequency Reduction (%): 95

Threat Types: Virus

Figure 7. Security Simulation in AcmeStudio.

Three different types of countermeasures can be defined: *Preventative* components affect the frequency at which threats occur; *Monitoring* components and *recovery* components reduce the effect of a threat. The architect specifies each of the countermeasure's target threat types, and the effectiveness or reduction that the countermeasure has on the target threat.

Once the particular properties are specified, the architect must then define paths (consisting of components and connectors) through the architecture that particular threats may take. The threat type that affects that path and the frequency (as a stochastic function) of the threat type are specified. After the threat transaction is created, the assets in the path can be given outcome values. The outcome can be in terms of dollars, loss of life, loss of productivity, etc. A weight is assigned to each outcome factor.

Threat scenarios are composed of one or more transactions. A scenario is used as the main entry-point for the simulation, and specifies the amount of time that will be used in performing the simulation. The simulation takes into account the threat entering the transaction path, the frequency of the threat type and the countermeasures in the path. Monte Carlo simulation is performed to determine the most probable damage value to each of the assets in the threat transaction. The value obtained is multiplied by the frequency of the threat transaction and the simulation time. This gives the total damage for the particular threat outcome factor. The end result of the simulation is a report that details the threat scenario, threat transaction, and total damage to the assets in the threat transaction path.

Consider the simple architecture illustrated in Figure 7, where we define the database as an asset (giving an asset value of $100K), run a security simulation on a path originating at the client and going through the firewall and server to the database for a simulated virus attack. We define the scenario so that (1) the simulation time is two virtual months; and (2) a virus attack happens on average 5 times per day, with a maximum of 20 attacks per day. If the firewall is 95% effective against virus threats then running the scenario indicates that the damage is calculated as $56, 677. If we were to run the same simulation without the firewall, the simulation will indicate that the loss of revenue increases to $1,112,409.

Such a simulation allows the architect to evaluate different scenarios, and to evaluate the effectiveness of different countermeasures against different attacks.

Providing different sets of properties on an architectural model facilitates different analyses of that model. It is therefore possible to compare and trade-off different scenarios and quality attributes of the same architectural model, rather than have to use different environments and architectural models in potentially different architectural languages.

4 Modelling Architectural Behaviour

An important aspect of modelling software architectures is the specification of abstract behaviour. By knowing the behaviour of architectural elements we can significantly improve the clarity of architectural designs. We can also analyse these specifications, for example to spot protocol mismatches in which interactions between components can potentially lead to deadlock (Allen and Garlan 1994 and 1997, Allen, Garlan, and Ivers 1998).

To illustrate, consider the simple system consisting of a pipe that connects two filters, F1 and F2, illustrated in Figure 8. The intuition behind such a pipe-filter system is that components communicate through buffered streams, writing through their output ports and reading through their input ports.

While the intuition may seem simple at first glance, understanding the real meaning of the figure (for example to implement F1 and F2) depends on detailed understanding of the interactions defined by the pipe. For example, from the figure alone it is impossible to answer the following questions:

- Which is the reading/writing end of the pipe?
- Is writing synchronous? That is, assuming F1 is the writer, does it block after writing?
- What if F2 tries to read and the pipe is empty? Does it block, or can it continue with other processing?

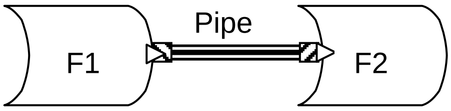


Figure 8. A simple pipe-filter system.

- Can F1 choose to stop writing?
- Can F2 choose to stop reading without consuming all of the data on the pipe?
- If F1 closes the pipe, can it start writing again at some future time?
- If F2 never reads, can F1 write indefinitely, or does F1 eventually block?

Note that there is no “correct” answer to these questions, since any set of answers could represent a possible pipe design. Indeed, in actual systems pipe implementations differ precisely along such dimensions of variability.

What is required is some way to specify the semantics of a pipe at the architectural level so that such questions can be answered easily. This would represent a marked improvement over existing practice in which decisions about such behaviour require one to examine the code of some implementation, existing examples of usage, or consult a human expert.

4.1 Representing architectural behaviour in Wright

There are many possible ways in which one might represent architectural behaviour. Indeed, practically any behaviour specification will do, including process algebras, state machines, and timed autonoma.To illustrate the general principles, we use the Wright specification language, one of the first to use formal modelling to specify architectural behaviour (Allen 1997, Allen and Garlan 1994).

Wright uses a subset CSP (Hoare 1985), a well-known process algebra, which defines behaviour in terms of patterns of events. Some of the constructs are listed in Figure 9. These include *events* (representing architecturally-relevant actions), processes (representing patterns of events), sequentiality (representing the ability to follow one behaviour by another), choice (representing the ability to branch), and parallel composition (representing the ability to compose partial descriptions). These CSP-based specifications can be associated with various architectural structures, including ports and roles.

```
Events:  e, request, read?y, write!5
Processes:        P, Reader, Writer, Client,
                  § (successful termination)
Sequence:         e → P,    P ; Q
Choice:  P |-| Q,    P [] Q
Composition:      P || Q
```

Figure 9. Behavior specifications in Wright.

4.2 Example

Figure 10 illustrates the basic ideas of behaviour description in Wright through a partial description of a pipe connector. Each role of the pipe (Reader and Writer) has an associated protocol defined in the subset of CSP summarized above. In addition, the connector has a “glue” specification (also a CSP process) that indicates how the roles interact through the connector itself.

```
Connector Pipe
  Role Writer = (write!x → Writer) ⊤ (close → §)
  Role Reader = Read ⊤ Exit
  where  Read = (read?x → Reader) [] (eof → Exit)
                     Exit = close → §
  Glue = Writer.write?x → Glue []
              Reader.read!y → Glue []
              Writer.close → ReadOnly []
              Reader.close → WriteOnly
  where ...
```

Figure 10. Partial Wright specification of a Pipe connector.

Such specifications, although compact, provide immediate answers to questions such as those posed above. For example, the specification in Figure 10 immediately tells us that a pipe writer can close at anytime, but cannot write again once it has close. A pipe reader can also close at any time, but if it chooses to read a value, it must be prepared to recognize an “end-of-file” marker and then immediately close.

Moreover, beyond clarification of design intent, specifications such as these permit a variety of analyses, including:

- Consistency of connectors: that the glue-mediated roles of a connector do not lead to a deadlocked state.
- Compatibility of component interface to connector interaction protocol: that a port satisfies the requirements of a connector role that it fills.
- Consistency of a component behaviour to its interfaces: that a port’s specification represents a correct projection of a component’s internal behaviour to that interface point.

Many of these checks can be performed semi-automatically by model checkers. See Allen (1997) for details.

5 Modelling Architectural Styles

One notable feature of software architecture is the ability to reuse styles and patterns. For example, many systems are described in terms like “client-server” system, “N-tiered system”, “pipe-filter system”, etc. Such terms refer to families of systems that share a common architectural design vocabulary (e.g., clients, servers, tiers, etc.) and a set of constraints on how that vocabulary can be used (e.g., that clients can’t talk directly to clients, or that connections don’t cross more than one tier).

```
Family PipeFilterFam = {
  Component Type filterT = {
  Ports {In,Out} ;
…} ;
Connector Type pipeT = {
  Role Reader = {Property datatype = …} ;
  Role Writer = {Property datatype = …} ;
  Invariant self.Reader.datatype ==
              self.Writer.datatype;
  …
}
System my-PF-System : PipeFilterFam = {
  Component F1: filterT = {…} ;
  Connector P:  pipeT = {…} ;
  …
}
```

Figure 11. Specification of a Pipe-Filter architectural style in Acme.

Important questions for architectural modelling and analysis are: How can we model an architectural style? How can we check that a given system has a given style? Can we combine several styles without leading to inconsistencies?

5.1 Architectural styles in Acme

We can specify styles by augmenting our architectural modelling notation with two things. First is the ability to define component, connector, and property types. These provide the basic vocabulary of design in that style. Second is the ability to define constraints on how instances of these types may be combined in a system description.[5]

For example, to define a pipe-filter style we would first need to define one or more filter component types and a pipe connector type. These would identify the kinds and number of ports on filters and roles on the connector. Additionally, we might define various property types, and indicate which properties are associated with which elements in the style. Next we would need to define constraints that might, for example, specify that there should be no dangling pipes or that a system should not have any cycles.

Figure 11 illustrates the basic ideas with a partial definition of a pipe-filter style, or *family*, as it is termed in Acme. Here we have defined a *Filter* component type, and specified that it must have at least an *In* and an *Out* port. We have also defined a *Pipe* connector type, and specified that it must have a *Reader* and a *Writer* role, and that each role must specify the datatype that is transmitted through that role.

The connector also includes a constraint, in this case an invariant that says the type of data written to a pipe must match the data read from it. Such specifications are written in a first-order predicate language (similar to UML's OCL), augmented with some functions that make it easier to refer to things like a component's ports, or the roles attached to a port.

With the pipe-filter family in hand, we can now use it to define a specific system in that style. In Figure 11 we illustrate the description of a system, *my-PF-system.* Components and connectors may now be declared as instances of the types defined in the family.

5.2 Example: Mission Data Systems

To illustrate the concepts of modelling and analysing style-oriented architectural description in more depth, we now describe a larger example: NASA's Mission Data System (MDS) (Dvorak and Reinholtz 2004). MDS includes an experimental architectural style for defining space systems. It consists of a set of component types (e.g., sensors, actuators, state variables), and connector types (e.g., sensor query). It also defines a number of rules that define legal combinations of those types. Figure 12 graphically illustrates the style, which consists of 7 component types, 12 connector types.

Figure 13 shows a screenshot of a simple MDS system displayed in AcmeStudio. The system represents a temperature control system consisting of a temperature sensor (*TSEN*), a temperature estimator (*TEST*), a heating actuator (*SACT*), a temperature state variable (*CTSV*), a heath state variable to indicate whether the sensor is behaving correctly (*SHSV*), a temperature controller (*TCON*) to issue commands to the actuator, and an executive that controls the value of the target temperature (*EXEC*). Appropriate connectors (of which there are 12 types) are used to define the interconnection topology.

The rules in MDS were initially defined in English and had to be hand translated into Acme constraints. A simple example of such a rule is

> *"For any given Sensor, the number of Measurement Notification ports must be equal to the number of Measurement Query ports (rule R5A)."*

This rule, which is a small part of a larger rule (see below) indicates that for every query port that a sensor provides, it must also provide an announcement port (and vice versa).

This rule was translated into the following constraint, which is associated with the sensor component type:

```
numberOfPorts (self,
                    MeasurementNotifReqrPortT)
==
numberOfPorts (self,
                    MeasurementQueryProvPortT)
```

[5] From a tooling perspective style definition may also entail specification of graphical conventions (shape, colour, layout) for the style, style-specific shortcuts for improving graphical editing (such as automatic creation of connectors based on naming conventions), and analysis tools to be included in an environment that uses the style.

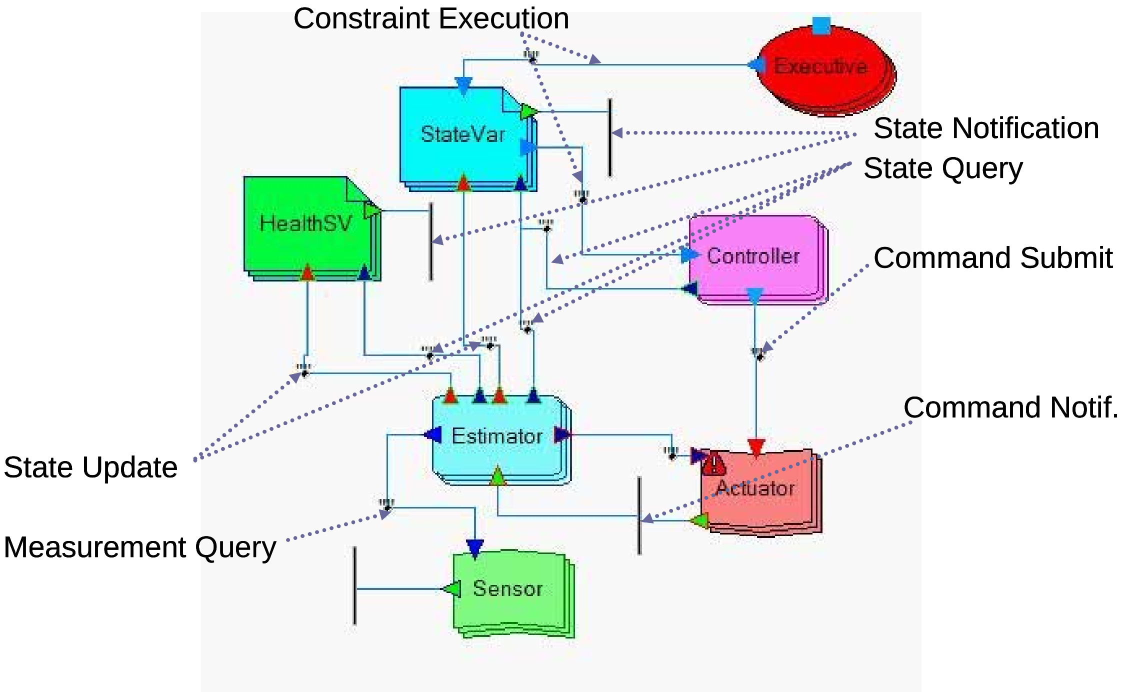


Figure 12. Definition of the MDS Architectural Style.

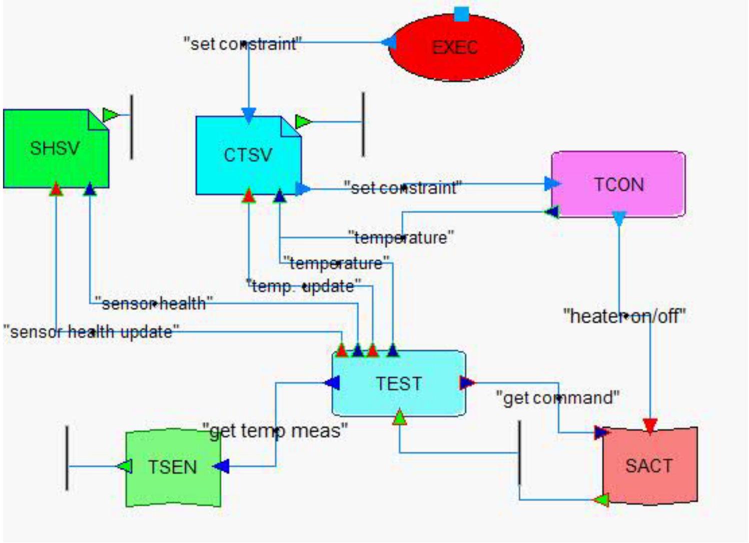


Figure 13. A simple control system in the MDS style.

Rules such as this one are continuously evaluated in AcmeStudio as the MDS architect creates an architectural description of an MDS System. If a rule is violated, the environment highlights the problem. Figure 14 illustrates how this appears to an architect.

Of course, checking rule satisfaction is relatively trivial for small systems and for such simple rules. Indeed, visual inspection could easily locate such rule violations. But in general MDS rules are much more complex, for example:

"Every estimator requires 0 or more Measurement Query ports. It can be 0 if estimator does not need/use measurements to make estimates, as in the case of estimation based solely on commands submitted and/or other states. Every sensor provides one or more Measurement Query ports. It can be more than one if the sensor has separate sub-sensors and there is a desire to manage the measurement histories separately. For each sensor provided port there can be zero or more estimators connected to it. It can be zero if the measurement is simply raw data to be transported such as a science image. It can be more than one if the measurements are informative in the estimation of more than one state variable."

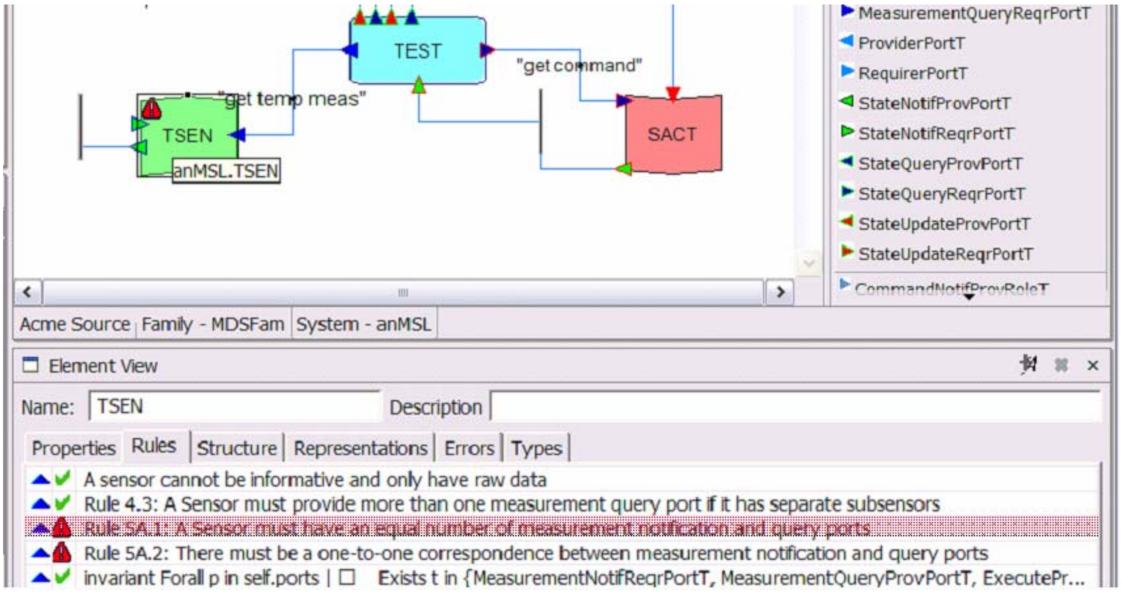
This is one of 12 such rules. Moreover, MDS architectures typically have hundreds of components. Complete checking of rule satisfaction in those situations becomes a significant problem for which formal style specification provides an effective solution.

5.3 Other Style-based Analysis

In addition to checking whether a given system conforms to a given style, it is often useful to investigate properties of styles themselves. For example, it is possible to define a style in which constraints lead to inconsistencies. For such systems it is impossible to create *any* system instances. Moreover, we may want to investigate whether the constraints of a style imply properties not explicitly modelled. For example, local constraints on attachments can be used to imply global connectedness.

To evaluate such properties we can interpret an Acme style description as a specification of a class of models, and use a model generator to check for the existence of such models.

Specifically, we can translate a style into an Alloy model and use the Alloy Analyser (Jackson 2002) to investigate



Name: TSEN

Description

- A sensor cannot be informative and only have raw data
- Rule 4.3: A Sensor must provide more than one measurement query port if it has separate subsensors
- Rule 5A.1: A Sensor must have an equal number of measurement notification and query ports
- Rule 5A.2: There must be a one-to-one correspondence between measurement notification and query ports
- invariant Forall p in self.ports | □ Exists t in {MeasurementNotifReqrPortT, MeasurementQueryProvPortT, ExecutePr...

Figure 14. Displaying problems to the architect.

properties of the resulting specification. Details of this analysis are beyond the scope of this paper, but the interested reader is referred to Kim and Garlan (2006).

6 Mapping between Architecture and Implementation

One of the difficult problems for an architect is ensuring that the implemented system is consistent with the intended architecture. Formal modelling and analysis can also help solve this problem

The problem for architectures is similar to the problem for any model-based method of ensuring that an implementation meets its specification. In general, there are two basic solutions. First, one can attempt to ensure satisfaction *by construction*. This can be done through a process of formal *refinement* in which a concrete model is obtained by applying well-founded refinement rules to a more-abstract model (or specification). Sometimes this process can be completely automated, in which case it is often termed *generation*. The second technique is to demonstrate that a lower-level model is consistent with a higher-level model by *comparison*. This is often done by providing a *mapping relation* between the two models. Both techniques can be used for software architectural models.

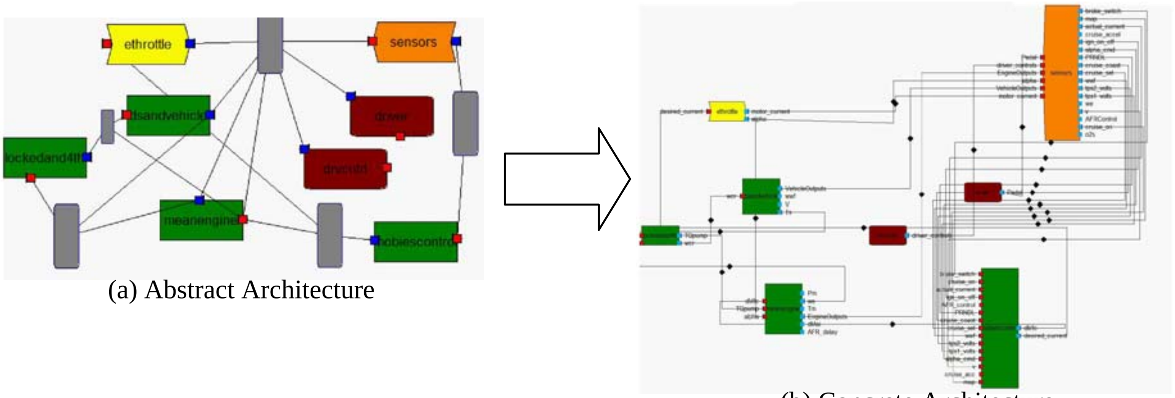
6.1 Refinement and Generation

Although using refinement in the most general case of software architecture is as difficult as any other form of model-based refinement, in many cases the problem is greatly simplified by exploiting architectural styles. That is to say, by limiting the problem to a specific class of systems and a specific class of implementations, it is often possible to build automated assistance for mapping architectures to implementations. The assistance can be in the form of automated transformations, or in the extreme case code generation of all or part of the target system.

We now illustrate this concept with two examples:

Example 1: Model Generation of Automotive Control Systems

A number of automotive companies have in place a component-based approach to control systems. Starting with an abstract architectural description, pre-specified components drawn from libraries are substituted to produce a



(a) Abstract Architecture

(b) Concrete Architecture

Figure 15. Mapping abstract automotive architecture to concrete automotive architecture.

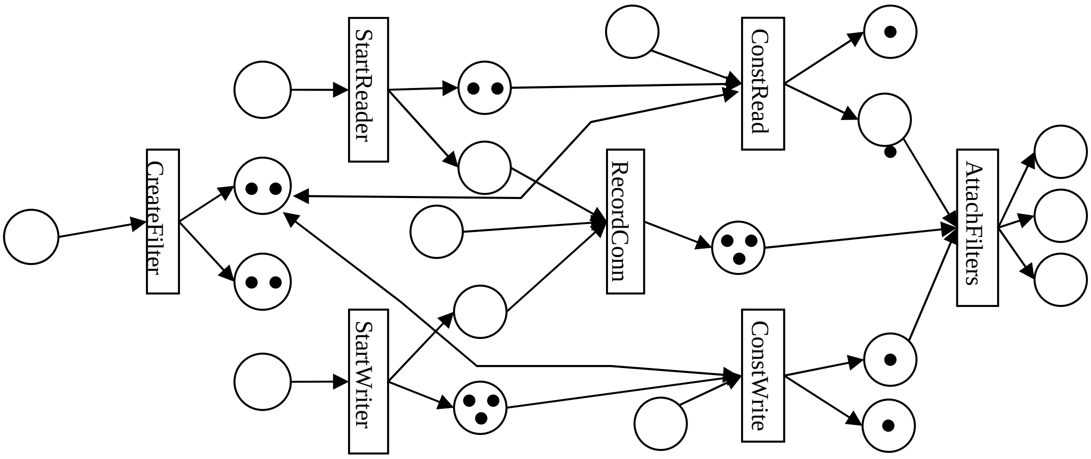


Figure 16. A DiscoTect Coloured Petri Net for Discovering Pipe-Filter Systems.

full system definition. In many cases the concrete components have formal models suitable for simulation, and in some cases code generation.

In Steppe et al. (2004) we describe a two-tiered approach that uses Acme architectural models of the architecture of an automotive system in two levels. At the higher (abstract) level, an architecture is described in terms of generic abstract components and simple virtual connectors (Figure 15a). In the lower (concrete) level model, concrete components are chosen from a repository of automotive components and substituted for the abstract ones, and detailed connections are made between them (Figure 15b). This concrete composition can then be sent to formal simulation tools for analysis.

While refinement of generic architectures to concrete architectures using component selection is a major step forward, one of the stumbling blocks is that refinement is done manually. In particular, the hooking up of concrete components, which may have dozens of ports is typically a time consuming process. Moreover, there are often dependencies between different components, so that choices of one component may affect others. Making sure that integrity rules of component composition are respected is a difficult, and again time-consuming, task.

However, it turns out that in many cases there are straightforward rules that can be applied to do most of the interconnecting. Indeed, in the case of automotive control systems when certain naming conventions are followed, almost all of the interconnecting can be done automatically. Further integrity rules can be specified as constraints in the style (as illustrated earlier). Indeed, the concrete version of the automotive software in Figure 15b was in fact generated directly using a plug-in to a version of AcmeStudio that had been specialized to model architectures in the two (abstract and concrete) styles.

Example 2: Code Generation for MDS Space Flight Systems

With certain modifications to the nature of the connectors in the MDS style we were able to provide a prototype code generator for MDS systems (Garlan et al. 2005). A key feature of that generator is the ability to target the resulting implementation to different platforms. For example, one platform might be the space environment, which requires power- and space-efficient code, while another platform might be the NASA testing environment in which resources are plentiful and there is a premium on support for debugging and monitoring.

The ability to generate retargetable implementations relies on the following:

1. There is a substantial body of reusable infrastructure code that supports inter-component communication, concurrency, and shared data.
2. It is possible to create a library of component implementations whose processing is not dependent on the implementation of the communication infrastructure. This code treats most components as input-output transformers, where the mechanisms for transporting code between components is irrelevant to the algorithms they implement.
3. There are a small set of attributes that determine the characteristics of the target platform. These attributes include the threading model, the amount and nature of debugging code, the target implementation language, and the task scheduler implementation.

Automatic generation of implementations in this domain allows engineers to work at a relatively high level of abstraction, in which the architectural principles of MDS are a primary focus at all times. The generator guarantees that the resulting implementation is consistent with the architectural model, and moreover does so in a way that is appropriate for the targeted run-time platform on which the system will be executed.

6.2 Direct Comparison

The second technique for ensuring compatibility between architecture and implementation is to find a way to compare the two. Since an implementation necessarily has considerably more detail than the architecture, the chief problem to solve is to abstract away the details of the implementation that are irrelevant to the architecture.

Two approaches are typically used. One is to perform *static analysis* on the code to infer high-level structure.

The other is to use *dynamic analysis* on the running system to capture actual run-time behavior and relate it to architectural models. Static analysis is particularly effective for recovering (or inferring) module-oriented structures, since, in general, determining dynamic behavior of a system (e.g., creating new components or connections) is undecidable. Dynamic analysis is particularly effective for inferring run-time structures, such as C&C views. For that reason we focus on dynamic analysis.

The basic model for dynamic analysis is a process involving a series of steps. First a system is monitored to extract low-level behavior, such as object and thread creation, method invocation, and variable assignment. Next, low-level, implementation-oriented events are processed to produce high-level, architecturally-relevant events. An architectural model is dynamically constructed by applying the abstract architectural events to an evolving model. Finally, the as-observed architectural model is compared to the as-designed architectural model (or style) to detect inconsistencies.

The main challenge in this process is the abstraction from low-level events to architectural events. This is difficult to do because it may be necessary to observe many low-level events before it is clear what architectural events have occurred. Moreover, these implementation events may be highly interleaved. For example, creating a pipe might involve creating both ends of the pipe and then joining them together. In this process it is possible that many writing ends of a set of pipes are created before any reading end is created.

To account for this complexity we need to define a formal mapping engine. In our own work we have developed the DiscoTect system to do this (Yan et al. 2004, Schmerl et al. 2006). At its core, DiscoTect represents a mapping engine that uses a formal mapping language to describe the relationship between patterns of low-level and high-level events. The output of a mapping description is a colored Petri net. After some filtering, low-level events enter the net as input tokens. Successive events may cause those tokens to move through the net, eventually emerging as output tokens representing architectural events.

Figure 16 shows the net that creates pipe-filter architectures from Java implementations that use Java pipe libraries, and represent filters as classes that adopt certain naming conventions. The tokens in the figure represent the current state of architectural reconstruction. Specifically, two filters have been constructed, one with a write port and one with a read port, and the pipe connection between them is about to be formed.

7 Related work

As noted in the Introduction, over 15 years there has been considerable research devoted to modeling and analysis of software architectures. This work falls into several categories.

7.1 Architecture description languages

A large number of ADLs and associated toolsets have been proposed by researchers (e.g., Balasubramaniam et al. 2004, Dashofy et al. 2002, Morconi and Riemenschneider 2007). Like the architectural modelling based on Acme described in this paper, most of these ADLs focus on component and connector structures and their properties. Several of them are specialized to specific architectural styles such as hierarchical publish-subscribe (Taylor et al. 1996), real-time control (Vestal 1996 and SAE International), or dataflow (Gorlick and Razouk 1991). Collectively they represent an impressive body of evidence about the utility of architectural modelling and analysis.

UML 2.0 by the Object Management Group provides an architectural modelling language for components and connectors that adopts many of the principles of Acme. However, these extensions are relatively new, and few tools have been developed to exploit them fully. Moreover, as a general-purpose modelling language UML is ill-suited to the problem of supporting domain-specific models that can take advantage of specialized analyses. However, several domain-specific profiles of UML have been proposed or are in the process of being ratified by the Object Management Group. Many of these have the benefits and power of the modelling approaches sketched in this paper.

7.2 Specification and analysis of arch behaviour

Wright, summarized in this paper, was one of the first modelling notations that attempted to provide behavioural modelling and analysis for software architecture (Garlan, Allen, and Ockerbloom 1994). Since then numerous behavioural formalisms have been used to provide complementary capabilities, including Chemical Abstract Machine (Inverardi and Wolf 1995), PO-Sets (Luckham 1996), Category Theory (Wermelinger 1998), Pi Calculus (Magee et al. 1995), and many others.

Most of these approaches share the goal of detecting mismatches in component compositions. The primary differences are the kinds of behaviour that can be modelled, and hence the kinds of mismatches that can be detected.

7.3 Refinement and Generation

Moriconi and colleagues were among the first to recognize the importance of formal mappings between architectures and implementations (Moriconi et al. 1995). Their approach uses structural transformation patterns to constructively create implementation-oriented models from architectural models.

UniCon, developed by Shaw et al. (1995) supports code generation from architectural models. Their approach creates a set of specialized compilation techniques for the various kinds of connectors that may go into an architecture. The goal is to provide a set of tools where any change to the implementation of a system must take place through the architecture. Other ADLs also have a certain amount of code generation capability (Luckham 1996, Taylor et al. 1997).

A number of projects have looked at reconstruction of architectures using static analysis. For example, Dali uses a variety of analysis techniques to create a high-level view of a system's implementation structures (Kazman and Carriere 1999). Since they focus on module-oriented views, they are complementary to the C&C-oriented approaches described in this paper.

ArchJava (Alrdich, Chambers, and Notkin 2002) augments Java with constructs for components and connectors, and uses typechecking to guarantee certain kinds of conformance between the component and connector levels of the system description and the lower-level implementation structures (classes, methods, etc.). In particular, the tools can guarantee that if two components are not connected at the architectural level, they cannot directly interact at the code level (e.g., through shared global variables).

A large number of people have become interested in "Model-driven Architecture" by the Object Management Group, which advocates a staged and automated approach to refinement of architectural designs to implementations. This is a natural complement to "Architecture-driven Models" – the theme of this paper. Much of the current work in MDA has focused on a staging in which platform dependencies are abstracted away in the high-level model, and bound during refinement. This is a special case of the approaches to refinement and generation outlined in this paper. Moreover,

8 Discussion and Conclusions

In this paper we have illustrated a number of ways in which formal architectural modelling and analysis can address important issues in software architecture, including clarifying design intent, supporting rich forms of analysis to enable detection of design flaws and make principled tradeoffs between quality of service goals, and allowing tools to help guarantee that implementations are consistent with the intent of their architectures. While the specific techniques described here draw heavily on research carried out by the Able Group at CMU over the past 15 years, many other research efforts have produced similar results.

There are several broad lessons that can be learned from this body of research.

1. **A little formality goes a long way.** The formalisms outlined in this paper are relatively simple. Simple structures with types, properties, relations, and behavioural descriptions can go a long way toward providing more improved capabilities for architectural design. Moreover, formal specification can be incremental: not all aspects of interest need be formalized or analysed.

2. **Reuse of existing methods.** The formal modelling and analysis techniques described in this paper rely on a large body of existing formal methods and tools, including model checkers, simulators, constraint checkers, and model generators. This is good news for software architects since it means that a large body of existing theory and tools can be applied with only minor modifications to the enterprise of software architecture design.

3. **One size does not fit all.** Architecture reveals a classic tradeoff between power and generality: the more general-purpose a model, the fewer opportunities for deep analysis. In our work we rely heavily on architectural style, and our ability to easily create style-specific tools, to exploit specific forms of analysis.

Although our ability to gain insight in software architectures through modelling and analysis has improved tremendously over the past decade, there remain a number of areas for which our techniques need to be improved. These include

- **Dynamic Architectures:** How can we reason about architectures whose structure changes dynamically? How can we determine when architecture changes can be performed safely on a system without restarting it? When can architectural changes be executed in parallel?

- **Software Architectures for Emerging Systems**: As technology advances so does our need to create systems that can take advantage of it. Today, for example, we are on the verge of ubiquitous computing systems that must work in the presence of hundreds of cooperating computational units, from cell phones, to sensors, to traditional computing platforms. What architectures are needed to handle such systems? Similarly, we are starting to see components whose behaviour is determined by machine learning. How can we specify what these components do and ensure that they are compatible with other components?

- **Managing Multiple Views**: So far, much tool support for architectural modelling focuses on a particular view, such as C&C views. How do we manage the relationships among multiple views of an architecture? To what extent can we ensure consistency between these views? How can we separate a particular architectural view into multiple views highlighting different concerns, to manage scalability?

Acknowledgements

The research described in this paper reflects work over the past 15 years funded by a variety of governmental agencies and corporations, including DARPA, NSF, ONR, ARO, Siemens, IBM, HP, and Microsoft. We gratefully acknowledge their support, and note that the opinions, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these funding agencies or corporations.

Numerous students and staff have contributed to the body of work summarized here, including Robert Allen, Elizabeth Bigelow, Shang-Wen Cheng, James Ivers, Jungsoo Kim, Andrew Kompanek, Charles Krueger, Ralph Melton, Bob Monroe, John Ockerbloom, Nicholas Sherman, and Bridget Spitznagel. Their hard work, insight, and inventiveness is largely responsible for the advances that

we have made, and we thank them for their varied but crucial support.

References

Aldrich, J., Chambers, C., and Notkin, D. (2002): ArchJava: Connecting Software Architecture to Implementation. Proc. ICSE 24, Orlando, Florida.

Allen, R. (1997): A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University School of Computer Science Technical Report CMU-CS-97-144.

Allen, R., Garlan, D. (1994): Formalizing Architectural Connection. *Proc. the 1994 International Conference on Software Engineering..*

Allen, R., Garlan, D., and Ivers, J. (1998): Formal modeling and analysis of the HLA Component Integration Standard. *Proc. the 6th International Symposium on the Foundations of Software Engineering (FSE-6),* Lake Beuna Vista, Florida.

Allen, R. and Garlan, D. (1997): A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, **6**(3).

Balasubramaniam, D., Morrison, R., Kirby, G.N.C, Mickan, K., and Norcross, S. (2004): ArchWare ADL Release 1 User Reference Manual. ArchWare Project IST-2001-32360 Report D4.3.

Bass, L., Clements, P., and Kazman, R. (2003): Software Architecture in Practice, 2nd Edition, Addison-Wesley.

Boehm, B., and Turner, R. (2003): *Balancing Agility and Discipline: A Guide for the Perplexed.* Addison Wesley Professional.

Bosch. J. (2000): *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach.* Addison Wesley.

Brookes, F. (1975): *The Mythical Man Month: Essays on Software Engineering*. Addison Wesley Professional.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal. M. (1996): *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns,* Wiley.

Butler, S. (2002): Security Attribute Evaluation Method: A Cost-Benefit Approach. *Proc. 24th International Conference on Software Engineering (ICSE2002).* Orlando, Florida, pp. 232-240.

Clements, P., Kazman, R., and Klein, M. (2001): *Evaluating Software Architectures.* Addison Wesley Professional: The SEI Series in Software Engineering.

Clements, P. and Northrop, L. (2001): *Software Product Lines: Practices and Patterns.* Addison Wesley SEI Series in Software Engineering,

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002): *Documenting Software Architectures: Views and Beyond,* Addison Wesley.

Dashofy, E., van der Hoek, A., and Taylor, R.N. (2002) An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *Proc. 24th International Conference on Software Engineering (ICSE2002).* Orlando, Florida.

Di Marco, A. and Inverardi, P. (2004): Compositional Generation of Software Architecture Performance QN Models. *Proc. the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04).* Oslo, Norway.

Dvorak, D., and Reinholtz, W.K. (2004): Separating Essential from Incidentals, An Execution Architecture for Real-Time Control Systems. *Proc. 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Austria.

Garlan, D., Monroe, R., and Wile, D. (2000)"Acme: Architectural Description of Component-Based Systems." In *Foundations of Component-Based Systems,* Cambridge University Press.

Garlan, D., Allen, R.J., and Ockerbloom, J. (1994): Exploiting Style in Architectural Design, *Proc. of ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*.

Garlan, D.; Kompanek, A. J.; & Cheng, S.-W. (2002): Reconciling the Needs of Architectural Description with Object Modeling Notations. *Science of Computer Programming 44,* 1, pp. 23-49.

Garlan, D., Reinholtz, W.K., Schmerl, B., Sherman, N., and Tseng, T. (2005): Bridging the Gap between Systems Design and Space Systems Software. *Proc. 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29),* Greenbelt, MD.

Gorlick, M.M. and Razouk, R.R. (1991): Using Weaves for Software Construction and Analysis. *Proc. 13th International Conference on Software Engineering (ICSE13).* IEEE Computer Society Press.

Hoare, C.A.R. (1995): Communicating Sequential Processes. Prentice Hall.

IEEE. (2000): *IEEE Recommended Practice for Architectural Description of Software Intensive Systems (IEEE Std 1471-2000).*

Inverardi, P. and Wolf, A. (1995): Formal Specification and Analysis of Software Architecture Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering* **21**(4).

Jackson, D. (2002): Alloy: A Lightweight Object Modeling Notation. *IEEE Transactions on Software Engineering and Methodology* **11**(2).

Jensen, K. (1994): An Introduction to the Theoretical Aspects of Coloured Petri Nets. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science vol. 803, Springer-Verlag, pp. 230-272.

Kazman, R. and Carriere. S.J. (1999): Playing Detective: Reconstructing Software Architecture from Available Evidence, *Journal of Automated Software Engineering* 6(2), 1999.

Kim, J.S. and Garlan, D. (2006): Analyzing Architectural Styles with Alloy. *Proc. Workshop on the Role of Software Architecture for Testing and Analysis 2006 (ROSATEA 2006)*, Portland, ME.

Luckham, D.C. (1996): Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events, *Proc. of DIMACS Partial Order Methods Workshop*.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995): Specifying Distributed Software Architectures. *Proc. 5th European Software Engineering Conference (ESEC 95)*.

Moriconi, M., Quian, X., and Riemenschneider, R. (1995): Correct Architecture Refinement. *IEEE Trans. Soft. Eng*. **21**(4).

Moriconi, M. and Reimenschneider, R. (1997): Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. *Technical Report SRI-CSL-97-01*, SRI International.

Object Management Group. MDA: The Architecture of Choice for a Changing World. http://www.omg.org/mda. Accessed November 29, 2006.

Object Management Group. *Unified Modeling Language (UML), Version 2.0*. http://www.omg.org/technology/documents/formal/uml.htm. Accessed November 29, 2006.

Object Management Group. OMG SysML Specification. http://www.sysml.org/docs/specs/OMGSysML-FAS-06-05-04.pdf. Accessed November 29, 2006.

Perry, D. and Wolf, A. (1992): Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, **17**(4):40-52.

Rasmussen, R. (2001): Goal-Based Fault Tolerance for Space Systems using the Mission Data Systems. Proc. 2001 IEEE Aerospace Conference, Big Sky, MT.

Rosanski, N. and Woods, E. (2005): *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives.* Addison Wesley.

SAE International. (2004): *Architecture Analysis and Design Language (AADL).* Document Number AS5506.

Schmerl, B. and Garlan, D. (2004): Supporting Style-Centered Architecture Development. *ICSE 26*, Edinburgh, Scotland.

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan. H. (2006): Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering* **32**(7).

Sha, K. and Goodenough, J. (1991): Rate Monotonic Analysis for Real-Time Systems. *Foundations of Real-Time Computing: Scheduling and Resource Management*, pp. 129-155. Kluwer Academic Publishers.

Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G. (1995): Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314-335.

Shaw, M. and Garlan, D. (1996): Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall.

Spitznagel, B. and Garlan, D. (1998): Architecture-Based Performance Analysis. *Proc. 1998 Conference on Software Engineering and Knowledge Engineering,* San Francisco.

Steppe, K., Bylenok, G., Garlan, D., Schmerl, B., Abirov, K., and Shevchenko, N. (2004): Two-tiered Architectural Design for Automotive Control Systems: An Experience Report. *Proc. Automotive Software Workshop on Future Generation Software Archtiecture in the Automotive Domain,* San Diego, CA.

Steppe, K., Bylenok, G., Garlan, D., Schmerl, B., and Shevchenko, N. (2004): Tool Support for Model Based Architectural Design for Automotive Control Systems. *Proc. 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application.* Enschede, The Netherlands.

Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, E.J., Nies, K.A., Oriezy, P., and Dubrow, D. (1996): “A Component- and Message-Based Architectural Style for GUI Software,” IEEE Transactions on Software Engineering **22**(6).

Vestel, S. (1996): “MetaH Programmer’s Manual, Version 1.09.” Technical Report, Honeywell Technology Center.

Vieira, M., Dias, M., Richardson. D.J. (2001): Software Architecture based on Statechart Semantics, *Proc. of the 10th International Workshop on Component Based Software Engineering*.

Wermelinger, M. and Fiadeiro, J.L. (1998): Towards and algebra of architectural connectors: A case study on synchronization for mobility. *Proc. 9th Workshop on Software Specification and Design.*

Yan, H., Garlan, D., Schmerl, B., Aldrich, J., and Kazman, R. (2004): DiscoTect: A System for Discovering Architectures from Running Systems," *Proc. of 26th International Conference on Software Engineering*, Edinburgh, Scotland.