# Architecture-Based Software Engineering

Judith A. Stafford and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{judys,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-891-99 November 1999

## ABSTRACT

*This paper provides an overview of the major issues and trends in architecture-based software engineering. While all software systems can be described in terms of components and interconnections, such descriptions are not always documented. Explicit description of software architecture provides a foundation for understanding and reasoning about both the functionality and quality of software systems very early in the development process as well as at a high level of abstraction. In this paper, we discuss the formalization of architectural specification, including a review of several languages developed especially for architectural description, and follow this with a discussion of architectural analysis techniques that can be applied to architectures described in these languages. Additionally, we discuss several other emerging concepts in software architecture that are having an influence on the general utility of the field.*

## 1 Introduction

All systems built from components exhibit structures that binds those components together. Consider, for example, the domain of civil engineering and, in particular, the structure that binds building materials into domestic dwellings. Civil architects are trained to apply their general knowledge of building materials, building codes, and aesthetic laws to a set of specific customer needs and budgetary constraints, such that the resulting dwellings are safe, functional, and affordable. Moreover, civil architects must be able to succinctly communicate their designs to a wide audience that includes customers, builders, and inspectors.

To make the task tractable, civil architects work by assembling their designs from familiar components that are well understood by architects, customers, builders, and inspectors. Rooms are defined by walls and interconnected by doorways; floors of rooms are interconnected by stairways leading to and from hallways; bathrooms on the same floor are placed near to each other and those on different floors are placed above each other. In essence, the design of a domestic dwelling is defined by its particular combination and interconnection of rooms.

But the arrangement of rooms in a domestic dwelling is only one structure of interest. Another is the plumbing structure and yet another is the electrical structure. The details of these structures, which involve different kinds of components and interconnections, are probably of little interest to the occupant, except to the extent that they affect the arrangement of rooms, such as the location of bathrooms. On the other hand, these structures are critical to plumbers and electricians. In fact, their interplay is also important and can affect the overall design.

The notion of software architecture can be understood by analogy to civil architecture. In the domain of software engineering, the structure of components is also the means by which a system is understood and evaluated. And here, too, there are several structures that are of interest and that involve different kinds of components and interconnections. For example, there are the procedure components and their interconnection through invocation, there are the subsystem modules and their interconnection through data and control communication, and there are the source files and their interconnection through compilation dependencies.

A software system's architecture is the arrangement of its components into one or more structures defined by the functional role played by each component and the interaction relationships exhibited by the components. The software architect is responsible for designing, modifying, documenting, evaluating, and enforcing a software system's architecture.

To make the software design task tractable, skilled software architects will make use of well-understood components and interconnections arranged into familiar and reliable structures. An extremely simple example is the compiler architecture known as the *multi-phase translator*. In this architecture, compilation is structured into the phases of lexical, syntactic, and semantic analyses, followed by optimization and target-code generation. Each phase is a translation of the abstract program from one form to another, such as the lexical analysis that causes characters to become tokens. Around this architecture has developed a formal theory and a toolset for each individual phase. In fact, it is the adoption of this common architecture for compilation that has allowed advances in individual phases, such as the invention of generator tools like LEX and YACC.

All software systems, just as all domestic dwellings, have an architecture. Moreover, software architectures are present at many levels within a system, just as a room can have an architecture separate from the house in which it is situated. To understand a software architecture we have to understand the context in which it is being described. For example, the multi-phase translator architecture is described above in terms of the role of functional components. A somewhat different architectural view is to say how the functional components are arranged in terms of their commu-

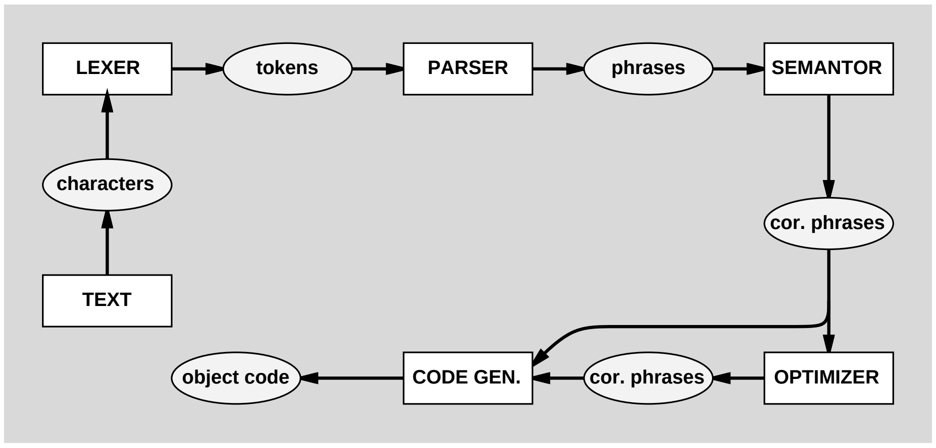


Figure 1: A Compiler Structured as a Pipe and Filter Architecture.

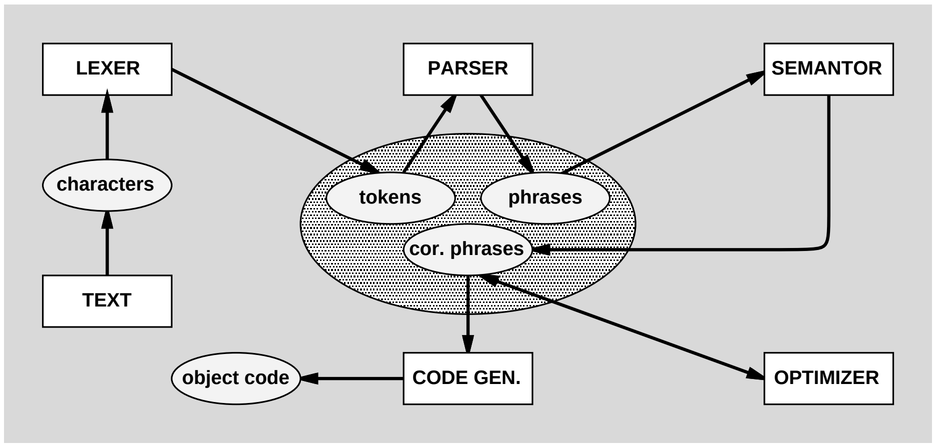


Figure 2: A Compiler Structured as a Shared Repository Architecture.

nication interconnection. One possibility is to say that they form a *pipe and filter* architecture, similar to the way commands can be strung together in the UNIX operating system (Figure 1). Another is to say that the components communicate and synchronize through a *shared repository* (Figure 2). The choice has a significant impact on the deepest design of the system.

There are many software architectures, at a variety of levels, that have been recognized as being commonly useful. They range from general-purpose architectures, such as the pipe and filter and client/server, to domain-specific architectures, such as telephone call processing and flight dynamics. The choice of an architecture depends on many factors, both functional and extra-functional. The functional criteria are almost obvious; the architecture must satisfy the functional requirements for the system. The extra-functional criteria are less obvious, but are in some sense more critical to the choice. These include the likely reliability, performance, and security of an implementation, the extensibility of the architecture, the availability of ready-made components, and the collective experience with the architecture.

Software architecture specifications are used for several different purposes, and these different purposes lead to rather different forms of description. For example, pictures are a convenient way of expressing the essence of the component and interconnection structure of a system, but are poor at expressing intended dynamic behavior. Prose descriptions are easy to write, but their ambiguity makes them useless for performing any kind of serious analysis.

One important reason for being explicit and precise about a system's architecture is to avoid a phenomenon known as *drift* [41]. Consider a domestic dwelling where a shaft is found to run between floors and so act as a convenient conduit for adding a new electrical line. If it is not clear in the architecture that the shaft was designed to be used solely as a chimney, then an unintended violation of the architecture (in this case, the interconnection structure between floors) could have disastrous results. It is not hard to think of analogous situations in software architectures where communication channels are similarly abused.

Whether an architecture is an instance of some commonly adopted architecture, has been made explicit, continues to be adhered to, and is of high quality, are all important questions. The emerging area of software architecture research is attempting to make the process of architecting software systems more rigorous, and the result of architecting more reliable and reusable, by bringing to bear powerful specification and analysis techniques.

This paper provides an overview of the major issues and trends in architecture-based software engineering. We begin by further discussing the importance to the development process of an architectural perspective. We then discuss the formalization of architectural specification, including a review of several languages developed especially for architectural description, and follow this with a discussion of architectural analysis techniques that can be applied to architectures described in these languages. Next, we briefly discuss several other emerging concepts in software architecture that are having an influence on the general utility of the area. We conclude with some thoughts about future directions.

## 2 Importance to the Development Process

There are basically two ways to attack any problem. One is to begin by considering various low-level details, and then, when convinced that those details can all be dealt with, proceed to higher levels of the solution. The other is to begin by exploring highest-level solutions, and then, after selecting a promising candidate, attempt to find a means for lower-level realization.

Many of the world's greatest design achievements begin with a high-level vision that has no obvious support in low-level engineering practice. When this is the case, realization of the vision is dependent on the development of new technologies. In some cases, these dependencies drive advances in the state of practice. In others, the advances are deemed to be impossible or unreasonably costly, in which case the vision must be altered. Vision and engineering are thus tightly coupled, each guiding and shaping the other.

Likewise, approaching the design of a software system at a highly abstract level allows the greatest amount of creativity, but incurs the greatest amount of risk. Software design based on abstract thinking must therefore be supported by experience-proven architectural styles and solid analytical techniques. Architecting is the central activity in the planning stage of software development, as depicted in Figure 3. The resultant architecture is important, but in the end, its utility depends on the quality of the planning that went into its creation.

The goal of software system development is to produce the best system that provides the functionality required by the commissioners of the system. There are many ways to define "best", depending on the system being built. For a missile guidance system, a higher value will likely

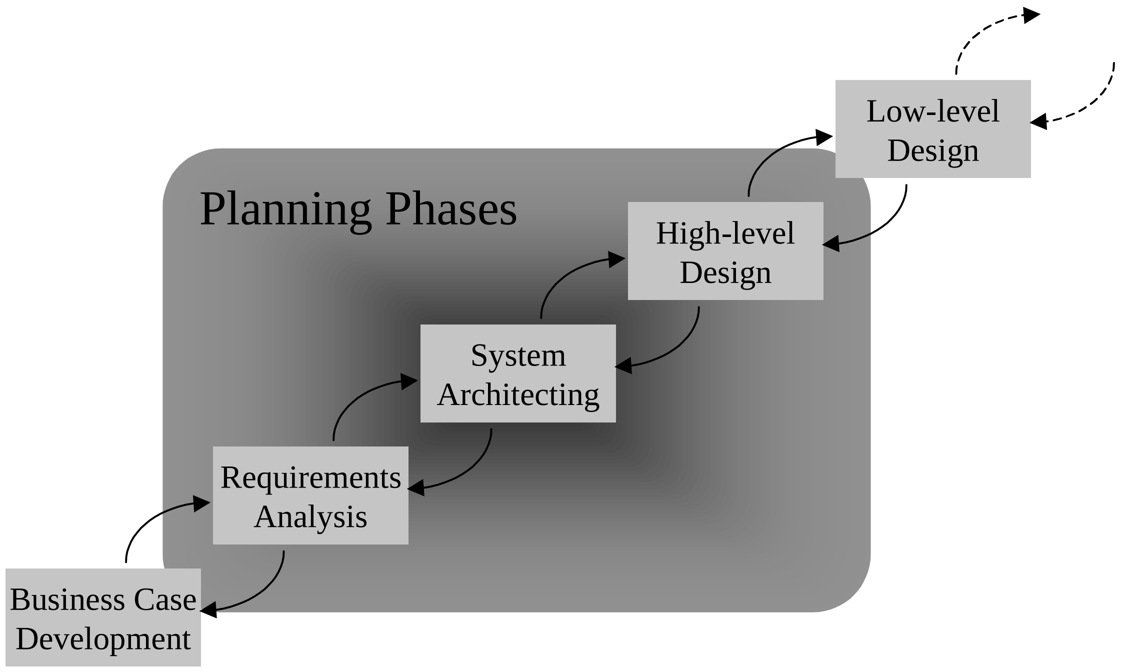


Figure 3: Architecting is the Central Planning Activity.

be placed on reliability than on time to market. On the other hand, the producer of shrink-wrapped commodity software, such as a word processor, often values time to market and feature differentiation over quality, until such time that the product has captured sufficient market share. Determining what constitutes the appropriate evaluation criteria for a system design is of prime concern during requirements analysis. This understanding influences the designer in the choice of an architecture. Decisions at this level are critical to the success of the development process, since these decisions directly affect all other aspects of the development process.

Traditionally, the software development process has been modeled as incremental steps involving refinement, beginning with an assessment of the problem to be solved and concluding with system maintenance (Figure 4). Iteration between steps is expected. For instance, during the testing stage, it might be discovered that an error exists that requires a major change in the design of the system. This will necessitate a cycling back to the design stage and repetition of all later stages. An explicitly and formally captured software architecture provides a medium for early analysis of the correctness and completeness of a design's satisfaction of system requirements, thus reducing the numbers and costs of problems encountered in later stages of development and reducing the time spent revisiting prior stages to correct those problems. In fact, the system view provided by an architectural description is useful throughout the life of a system as a high-level reference point for evaluation and change.

Below, we discuss two primary benefits of the inclusion of an explicit architecting stage in the software life cycle: first, increased abstraction for understanding large, complex systems and, second, the ability to perform high-level analysis.

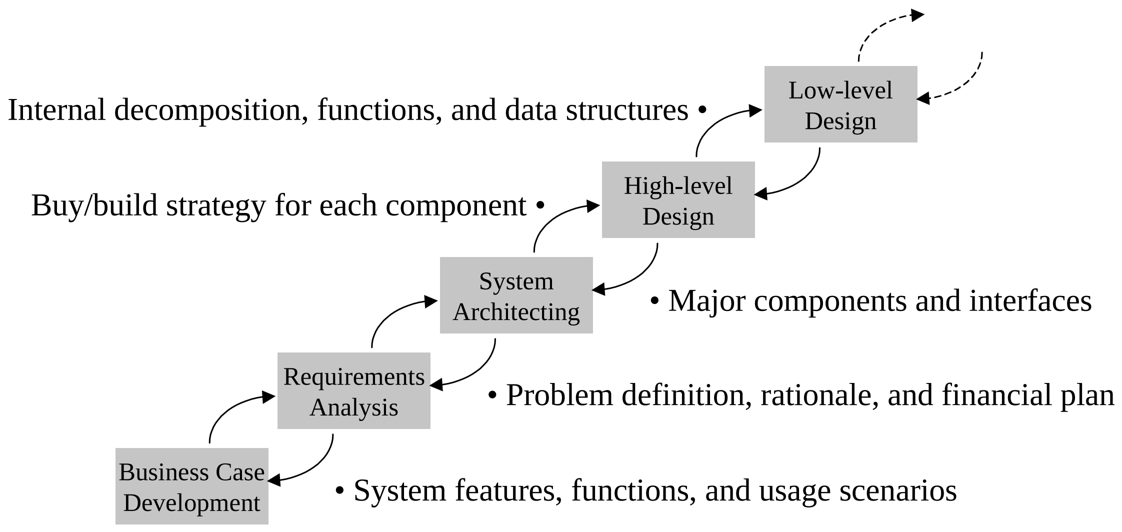

Internal decomposition, functions, and data structures •
Low-level Design
Buy/build strategy for each component •
High-level Design
System Architecting
• Major components and interfaces
Requirements Analysis
• Problem definition, rationale, and financial plan
Business Case Development
• System features, functions, and usage scenarios

Figure 4: The System Development Process Proceeds in Stages.

## 2.1 Abstraction

Abstraction is a modeling activity that, in essence, amounts to the selective highlighting (and, correspondingly, selective hiding) of particular characteristics of a system. If one considers a software architecture to be an abstract model of a system, then what are those highlighted characteristics? In general, they are *components*, *connections*, and the *behaviors* that regulate the interaction of components through connections. Viewing a system from an architectural perspective encourages this abstract way of thinking about system development, which results in benefits that include improved communication among the stakeholders of the system and improved software understanding of the system and its development process.

### 2.1.1 Improved Communication Among Stakeholders

Communication among stakeholders during the development of a system is crucial to success. It is also very difficult. Stakeholders, such as funding agencies, inspectors, system commissions, and system developers, must be able to communicate about their needs and ability to meet the needs of others. The differences in views and backgrounds is greatest among the stakeholders involved in the high-level design of a system [18]. For instance, at that critical point in development, it is important for the system commissioners with knowledge of the desired system functionality but, perhaps, no knowledge of system development, to be able to communicate with system developers who know how to build systems but may know little about the application domain where it is intended to be used.

Architectural description provides a vehicle for such communication. At the simplest level, a picture of the system can be drawn that is comprised of boxes and arrows representing the functionalities of the system and how they would be expected to interconnect. This level of architectural description allows the commissioners of the system to see that their needs have been recorded and it allows the developers to consider different ways in which that functionality might be implemented. At a more detailed level, a formal description of components, connections, and behaviors provides an object of shared introspection that can be formally analyzed, and perhaps even simulated.

### 2.1.2 Software Understanding

Code-based system understanding becomes difficult in moderately sized systems and impossible in large systems. The ability to reason about the implementation of a system based instead on the architectural description can greatly improve system understanding. Successful use of architecture for these purposes requires the ability to map between levels of abstraction of the system. But because of the phenomenon of architectural drift, it is critical that the mapping somehow be kept legitimate. Three proposed solutions to this problem are architecture recovery, forced consistency, and system generation. Forced consistency and system generation are useful when building new systems, while architecture recovery is an alternative for use with systems for which an architectural description does not exist or for systems whose architectures are known to have freely drifted over time.

#### Architecture Recovery

Architecture recovery methods use a combined top-down and bottom-up approach to discover the architecture hidden within an implementation. They are concerned with helping system engineers increase their understanding of very large, implemented systems for the purposes of change analysis. As examples, Gall et al. [62] use domain knowledge combined with existing reverse engineering techniques to discover the current architecture in the ARES project. Murphy, Notkin, and Sullivan [36] create a mapping from the actual source onto an architectural description that was drawn up by a person familiar with the overall functionality of the system.

Weide et al. [63] make the argument that reverse engineering based on architecture recovery is intractable. They do not claim that this intractability should imply that all work in the area should cease, but rather that software engineers must concentrate on building new systems better since most software systems have yet to be built. Of course, this view is not helpful to long-lived systems that are too expensive to replace, such as the air traffic control system or the global telecommunications and banking infrastructures, which must continue to evolve for decades to come.

#### Forced Consistency

Consistency can be accomplished when a change in one system artifact is propagated to all related artifacts. Forced consistency is a primary goal of Gestalt [47]. The authors of that language consider inconsistency between architecture and code to be the prime concern for improving the usefulness of software architectures. The actual structure of a C program is captured and revised into a Gestalt architecture based on preprocessor directives (e.g., `#include`) found in the source code files. After the architecture is constructed, changes in one are propagated to the other. van der Hoek et al. [59] are studying the use of versioned software architecture as a possible forced mapping solution to the architecture/implementation mapping problem. With the goal of achieving run-time change for systems, the C2 architectural style [39] requires an inter-level mapping. In the case of C2, changes at the architectural level should naturally be propagated to the implementation level. They assume that implementation-level changes will be restricted by a "preconceived set of application invariants".

### System Generation

System generation is the automatic creation of an implementation given a high-level system description. Methods of system generation vary but are generally based on the selection and composition of predefined, coarse-grained components. The generation of systems is raised to an even higher level by GenVoca [4], which is a system for generating application-specific system generators. Any system that is built via the selection of components rather than implementation coding should provide increased reliability of the mapping between source code and architecture.

It would seem that there should never be a reason to access automatically generated source code. However, engineers (even hardware engineers) are continuely haunted by undocumented implementation-level changes to their implemented systems. Thus, software generation will need to provide a means for assuring that the system can only be modified through architecture-level changes.

System generation techniques are discussed further in Section 5.3.

## 2.2 Architecture-Level Analysis

The goal of software analysis is to determine the degree to which a system satisfies its requirements.[1] There are many ways of distinguishing analysis techniques. The distinctions generally center around tradeoffs between the accuracy of the method and the effort required to perform the analysis [65] because, in general, proving the correctness of systems is intractable. An additional distinction can be made between techniques for detecting the presence of incorrectness and techniques for locating the cause of incorrectness. Most automated architectural analysis techniques that have been developed or suggested to date are of the former kind [2, 22, 31, 37, 44]. Recently, dependence analysis has been suggested as a basis for architecture-level fault localization and impact analysis of the latter kind [54]. Finally, there is an important distinction to be drawn between analysis of functional and extra-functional properties. Again, most existing techniques are of the former kind.

Many system properties can be analyzed for faults during the early stages in development. For example, Inverardi et al. [22] demonstrate the advantage of early analysis in the discovery of functional component mismatch, where the methods of data exchange expected between two architectural components were incompatible. Magee et al. [32] reveal the potential for deadlock in an architectural specification of the Olivetti Active Badge System. Naumovich et al. [37] apply an analysis tool to an architectural description of the classical gas station example to detect a potential race condition between customers trying to pump gas. Stafford et al. [54] apply a dependence analysis technique to discover a failure and locate the fault in a different architectural specification of the same gas station example.

In general, detecting faults during the early stages, including during architecting, results in fewer errors being propagated forward, which in turn results in increased reliability and reduced maintenance costs, as depicted in Figure 5. It is not unusual for nearly 40% of development expenses to be incurred in testing and maintenance of a system [43]. In a case study involving the prediction of software quality based on the development of the JStar ground surveillance system [26], the number of faults created during design was 32%, yet the number of faults discovered before or during design was a mere 1%. The expectation is that a solid architectural design can reduce the propagation of errors to late stages of development by providing a vehicle for communication between stakeholders in the system and by providing support for early analysis. Among the properties that can benefit from early analysis are the following.

[1]This form of analysis should not be confused with "requirements analysis", which is intended to examine a problem domain in order to elicit a set of requirements.

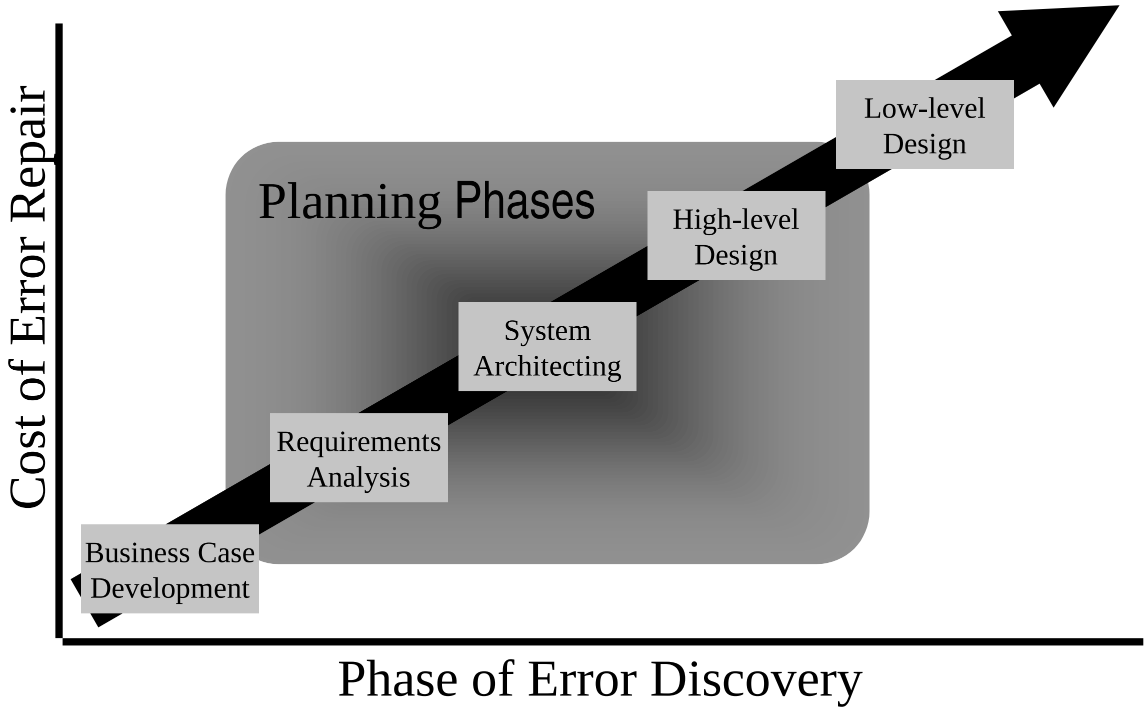


Figure 5: Early Analysis for Errors Reduces Maintenance Costs.

- *Completeness*—the entire range of functionality specified in requirement documents are met.
- *Liveness*—intended system behavior (e.g., computational progress) eventually occurs.
- *Safety*—no unintended behavior (e.g., deadlock) ever occurs.
- *Component interaction*—connected component interfaces and functionalities are compatible.
- *Performance*—system performance requirements are met.

A course-grained view of a system, which allows one to think about the system in large chunks, provides important benefits even after system design. Many issues, when addressed at the implementation level, are intractable for anything more than small (less than a few thousand lines of code), single-author systems. When dealing in the large, an architecture can provide assistance in performing tasks such as the following.

- *Fault Localization*—After a failure has been identified in a system, it is necessary to investigate the cause of the failure. This is normally done by systematically reducing the code to be inspected until the fault is eventually isolated to a specific set of statements. In a large system, the first approximation to this reduction is provided by the component structure of the system's architecture.
- *Regression Testing*—When a modification is made to a piece of software, it is necessary to rerun tests to assure that no faults were introduced during the modification process. Minimizing the tests to just those affected by the change helps reduce the cost of regression

testing. Again, in a large system, the first approximation to this minimal set is provided by associating the tests with the component structure inherent in the architecture.

- *Detection of Software Drift*—Over time the structure of a system tends to drift due to undocumented enhancements and repairs. The architecture provides a touchstone against which drift can be detected and measured.
- *Reuse*—When some component of a system is recognized as being usable in other applications, then that component, as well as any other components that it requires, are extracted and used in other settings. This requires an understanding of the architectural context and assumptions within which the component was conceived.
- *Reverse Engineering*—When a system has been in service for many years, then it often becomes difficult to make modifications because of lost engineering information describing the system. A typical, and tractable starting point for recovering this information is to recapture the architecture.

Each of these tasks currently involves detailed sifting (whether manual or computer assisted) through source code in order to achieve the particular goal. The architecture can be used to segregate large chunks of code to either be included or ignored during this examination, thus enhancing the practicality of the process.

## 3 Formalization and Software Architecture

Traditionally, software architectures have been described informally as natural-language documents. Box and arrow diagrams are sometimes used to bring more precision to descriptions of software architectures, but while they can reveal some ambiguous and missing requirements, they are not capable of modeling all the information provided in the natural-language specification, such as system behavior. Formalization, as applied to software development at the architectural level, involves the application of mathematically based modeling languages to capture structural and behavioral properties of systems. Above all, these languages provide support for rigorous analysis of a system early in the life cycle and/or at a high level of abstraction. But additional benefits accrue as well.

- A formally described system is a vehicle for precise and unambiguous description supporting communication among stakeholders in system development.
- Well-understood and well-documented components and connections provide a means to exploit commonalities among architectures.
- Explicit association of features and properties of architectures can be an aid in choosing among design alternatives.
- Increased modifiability of a system is possible due to well documented component interfaces and the availability of reliable techniques to analyze the possible effects of modifications to a system.
- Formal description provides a means to capture domain-specific properties, requirements, and idiosyncrasies in ways that support domain-specific architectural generalizations.

- Increasing the granularity of descriptive elements improves scalability, and thus support for the development of large-scale systems.

The creation of a formal architecture description naturally uncovers ambiguities and omissions. Moriconi et al. [35] discuss the need for formality in high-level descriptions in order to support the ability to create and use hierarchical abstractions and mappings between them. Shaw et al. [49] discuss differences in the abstractions necessary for describing architectures and those required for instantiation of systems. Most discussions that include the topic of architecture descriptions also mention the need for support tools to aid in analysis. But development of automated analysis tools depends first on the availability of formal languages that support system description.

One of the goals of the ISO 9000-3 guideline for application of the ISO 9001 standard to software development is formalization at all stages of the software life cycle. Oskarsson and Glass provide a practical alternative to building ISO 9000 compliant software [40]. They are supportive of formal approaches as an aid to building quality software. However, they claim that the focus on the use of formal methods is misdirected and raise two primary objections. The first is based on the evolutionary nature of software development. They state that the ISO emphasis on "complete and unambiguous" specification will best be implemented through the use of formal specification languages. However, they equate this use with the inhibition of evolution in software. Oskarsson and Glass's second objection is based on the fact that formal methods are beyond the understanding of most, if not all, customers and users as well as many developers.

An emphasis on "complete and unambiguous" specification need not imply that a system become fixed in time. What it does mean is that all system artifacts should be kept in step throughout the life of the system. Advances in analysis techniques, in fact, could even enhance system evolution by isolating the effects of changes and by raising the developer's confidence in those changes. As to the assertion that formal methods are beyond understanding, this denies the experience of other engineering disciplines, which regularly use more sophisticated techniques than those found in software engineering. Indeed, the essence of most formal methods are a normal part of higher education: set theory, graph theory, and logic. What is true is that the community has not yet settled on which specific techniques should be taught, and so most software engineers currently leave their studies ill prepared to learn advanced formal methods.

## 3.1 Architecture Description Languages

Languages for describing high-level designs have been in development and use for over two decades. The recent interest in software architectures has spawned the development of a new generation of such languages, which are now referred to as ADLs (architecture description languages). Most are intended to support some sort of formal analysis. Most are also still rather immature and few descriptions of actual system architectures have been written using them. However, this is an active area of research focused on providing languages and tools that are practical and useful.

Table 1 provides a list of several representative ADLs. These ADLs vary widely in their level of expressiveness. They also vary widely in maturity, as well as in the degree to which the structure and behavior of the system can be modeled. For instance, Darwin [31] concentrates its modeling on the structure of a system, providing support for behavioral specification and analysis through separate tools, while CHAM [21] concentrates more on directly capturing system behavior than on system structure. Rapide [56] and Wright [2] are particularly rich in describing both structure and behavior. In the case of Rapide, the goal is early system simulation, while in the case of Wright, the goal is to study the formal specification and analysis of architectural connection.

| ADL | Organization | Ref. | Design Goals |
|---|---|---|---|
| Acme | Carnegie Mellon University | [14] | Provide an interchange language to facilitate sharing of architectural components and analysis tools. |
| ArTek | Teknowledge Corporation | [58] | Describe the architecture of evolving large-scale components as a communication and coordination device within multi-contractor projects, to capture the architecture of a delivered system for documentation purposes, and to provide support comparing and contrasting existing architectures. |
| CAPS | Naval Postgraduate School | [51] | Provide a software development environment for graphical specification and prototyping of real-time systems. |
| CHAM | University of L'Aquila<br>University of Colorado | [21] | Investigate a formalism for description of architectural components and the interaction among them for the purposes of system analysis. |
| Gestalt | Siemens Corporate Research | [47] | Provide support for describing the structure systems, abstractions needed by developers, and communication mechanisms used in industrial software, as well as automatically checking of consistency with source code. |
| MetaH | Honeywell Technology Center | [61] | Specify real-time and concurrent aspects of the software and hardware of a system, and provide support for analysis of functional and extra-functional system properties. |
| Modechart | University of Texas at Austin | [25] | Provide a formalism for the specification, analysis and simulation of real-time systems |
| Rapide | Stanford University | [55] | Provide an executable ADL based on a rule-event execution model for prototyping, simulating, and analyzing of software systems. |
| RESOLVE | Ohio State University | [19] | Provide a framework, discipline, and language for specifying, developing, and composing reusable software components. |
| SADL | SRI | [35] | Provide support for specifying the structure and the semantics of an architecture through explicit mappings among architectures, architectural styles, and architecture refinement patterns. |
| UniCon | Carnegie Mellon University | [66] | Support style-based architectural construction by interconnecting predefined or user-defined architectural components. |
| Wright | Carnegie Mellon University | [2] | Provide a formalism that focuses on explicit connector types and analyses associated with architectural connection. |

Table 1: Comparison of Several Representative Architecture Description Languages.

## 3.2 Classification of ADLs

As is true for implementation languages, no one ADL is the best choice for all systems. Several classification schemes have been suggested based on different aspects of the decision process [33, 34, 41]. One primary difference among ADLs is the model they support in a representation. Shaw and Garlan [50] review a set of model types that appear regularly in software: structural, framework, dynamic, process, and functional. They describe how each model is more or less well supported by various ADLs. Vessey and Glass discuss the notion of "strong" versus "weak" approaches to system development [60]. Strong systems development implies a tight fit between the methods being used and the problem being solved. Weak development methods are applicable to many problems but not particularly good for any specific problem domain. Vessey and Glass feel that there needs to be more work done in the area of determining what models are best suited to solving problems in particular domains. Association of ADLs with application domains should improve selection of an ADL for describing the architecture of a particular system.

One can imagine classifications based on the formality of the description, the types of architectural entities emphasized, the support for various types of analyses, the intended specifier, and intended audience of specifications captured in the ADL. The most complete attempt to classify ADLs to date is presented by Medvidovic [33]. He offers a definition of an ADL and then proceeds to classify and compare the languages that fall within the bounds of that definition. The basis for his classification is the support provided for various aspects of components, connections, and configurations as well as the types of tool support provided with the language. The classification is not intended to suggest certain languages for specific tasks but rather to provide an overview of what the various languages provide and how they compare in support for various features he considers essential to an ADL.

## 3.3 Overview of Several ADLs

We now present an overview of several ADLs that support different aspects of high-level software description.[2] CHAM [21] provides support for analysis of system behavior. Darwin [31] is based on the premise that simple is better, and directly provides support for only the structural aspects of a system. Rapide [55] is intended as a simulation language for event-driven systems. MetaH [61] provides support for specifying both the hardware and software components of a system along with the relationships among them. Unicon [66] supports composition of systems from predefined components. Wright [2] explores formal aspects of architectural connection. Acme [14] attempts to model the constructs generally considered to be essential to all ADLs.

For each of the languages, we reproduce an example drawn from the published literature describing the language. Space does not permit us to discuss the examples in detail. Our intent is simply to give a flavor of each language; the reader is referred to the literature for more specific information.

### 3.3.1 CHAM

The Chemical Abstract Machine (CHAM) is an operational model based on the theory of term rewriting. It uses the metaphors of molecules to model components, chemical solutions to model system states, and chemical reactions to model system state changes [5]. Structured in a particular way, a CHAM provides a means for concisely describing the components and behavior of a software system at the architectural level [7, 21, 22]. Using its associated analysis techniques of structural

[2]Exclusion of other ADLs from this overview does not indicate a lack of importance, merely a lack of space.

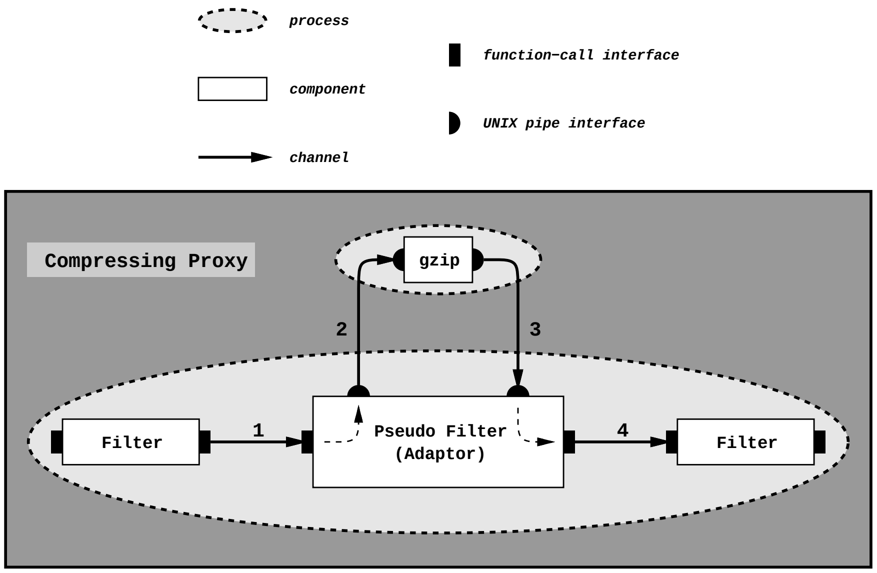


Figure 6: The Compressing Proxy [7].

induction and transition system construction, CHAMs can reveal incorrect global system behaviors as well as component behavioral mismatches.

Figure 6 depicts a system modeled using a CHAM, a portion of a CERN HTTP (Hyper Text Transfer Protocol) server that uses the **gzip** compression/decompression program to increase transfer rates [13]. A CERN HTTP server is structured as a series of filters to allow easy extension, as in the case of the **gzip** addition. A CHAM description for the compressing proxy architecture [7] is shown in Figure 7 and includes molecules, solutions, and reaction rules (also called transformation rules). Transitions in the state of the system are described through rewriting of terms, where the terms of an expression are transformed into a new expression based on the rules defined for the system.

A software architecture described as a CHAM is structured into three parts:

- a *syntax* for representing the system components;
- a set of *reaction rules* for describing the system behavior; and
- an *initial solution* for representing the original, static configuration of the system.

Following Perry and Wolf [41], CHAM descriptions distinguish three types of design elements: data, processing, and connecting. The data elements contain information, the processing elements transform the information, and the connecting elements glue the components of an architecture together.

The syntax shown in Figure 7 consists of the set $P$ representing three kinds of processing elements. The connecting elements for the architecture are given by a second set $C$ consisting of two operations, $i$ (for input) and $o$ (for output), that act on the elements of a third set $N$. This third set is used to define the topology of the system in terms of the communication channels connecting the components, and correspond to the numbers given in Figure 6. A fourth set $E$ introduces the control signals used in the communication between **gzip** and its adaptor. The set $F$ contains the

```
/* Molecule Syntax Definition */
```

$M$ ::= $P \mid C \mid E \mid M \diamond M$
$P$ ::= $F \mid \mathbf{AD} \mid \mathbf{GZ}$
$C$ ::= $i(N) \mid o(N)$
$N$ ::= $1 \mid 2 \mid 3 \mid 4$
$E$ ::= $\mathbf{end}_i \mid \mathbf{end}_o$
$F$ ::= $\mathbf{CF}_u \mid \mathbf{CF}_d$

```
/* Initial Solution Definition */
```

$$
\begin{aligned}
S_0 = \; & \mathbf{CF}_u \diamond o(1), \\
& \mathbf{CF}_d \diamond i(4), \\
& i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}, \\
& i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}
\end{aligned}
$$

```
/* Transformation Rules */
```

$$
\begin{aligned}
T_1 &\equiv i(x) \diamond m_1,\ o(x) \diamond m_2 \longrightarrow m_1 \diamond i(x),\ m_2 \diamond o(x) \\
T_2 &\equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m \\
T_3 &\equiv \mathbf{end}_o \diamond m_1 \diamond o(x),\ \mathbf{end}_i \diamond m_2 \diamond i(x) \longrightarrow m_1 \diamond o(x) \diamond \mathbf{end}_o,\ m_2 \diamond i(x) \diamond \mathbf{end}_i \\
T_4 &\equiv \mathbf{end}_i \diamond m_1 \diamond \mathbf{GZ} \diamond m_2 \longrightarrow m_1 \diamond \mathbf{GZ} \diamond m_2 \diamond \mathbf{end}_i \\
T_5 &\equiv \mathbf{end}_o \diamond \mathbf{GZ} \diamond m \longrightarrow \mathbf{GZ} \diamond m \diamond \mathbf{end}_o \\
T_6 &\equiv \mathbf{GZ} \diamond m \longrightarrow m \diamond \mathbf{GZ} \\
T_7 &\equiv f \diamond c \longrightarrow c \diamond f \\
T_8 &\equiv \mathbf{AD} \diamond i(1) \diamond m \longrightarrow i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD} \\
T_9 &\equiv \mathbf{AD} \diamond i(3) \diamond m \longrightarrow i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}
\end{aligned}
$$

Figure 7: CHAM Description of the Compressing Proxy [7].

representation of the "upstream" and the "downstream" CERN filters between which is placed the adaptor for **gzip**. By convention, the infix operator "$\diamond$" is used to express the status of a processing element in terms of its possible sequence of behaviors.

Notice that at the level of abstraction used in this particular description, the concern is not with the actual data transferred between the components, but simply the protocol by which the components communicate. In general, CHAM descriptions can be used to specify data aspects as well.

The next step in specifying an architecture is to define an initial solution $S_0$. This solution is a subset of all possible molecules that can be constructed under $\Sigma_b$ and corresponds to the initial configuration of a system conforming to the architecture. The solution establishes the basic connectivity of the components and their initial status. For example, $S_0$ establishes that **GZ** is initially in the state of accepting data along channel 2 (from **AD**). It can then end its input and enter a state of offering data along channel 3 (to **AD**), after which it can end that output. **AD** is initially in the state of accepting data along channel 1 (from an upstream filter) and must wait until it has stopped accepting the data before it can offer data on channel 2 (to **GZ**). It can then end its output. The full meaning of the initial state becomes apparent when combined with the transformation rules.

There are eight transformation rules that define the complete behavior of the compressing proxy at this level of architectural modeling, where $m, m_1, m_2 \in M$, $x \in N$, $c \in C$, $e \in E$, and $f \in F$. Rule $T_1$ is a general inter-element communication rule, rules $T_2$ through $T_5$ capture the communication protocol between **gzip** and its adaptor, rule $T_6$ enables the iteration of **GZ**, and rule $T_7$ describes the activation of the upstream and downstream filters. Rules $T_8$ and $T_9$ describe the behavior of the adaptor.

We now trace through just a few applications of the transformation rules to illustrate how the formulation captures the essence of the architecture. First, data to be compressed must be available, and therefore the solution must be "heated" by rule $T_7$ acting on the molecule $\mathbf{CF}_u \diamond o(1)$.

$$\begin{aligned} S_0 &\xrightarrow{T_7} S_1, \text{ where} \\ S_1 &= o(1) \diamond \mathbf{CF}_u, \\ &\quad \mathbf{CF}_d \diamond i(4), \\ &\quad i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}, \\ &\quad i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \end{aligned}$$

Now a reaction can occur within the subsolution consisting of molecules $o(1) \diamond \mathbf{CF}_u$ and $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$. This reaction is governed by $T_1$ and represents the initial transfer of data from $\mathbf{CF}_u$ to **AD**.

$$\begin{aligned} S_1 &\xrightarrow{T_1} S_2, \text{ where} \\ S_2 &= \mathbf{CF}_u \diamond o(1), \\ &\quad \mathbf{CF}_d \diamond i(4), \\ &\quad o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1), \\ &\quad i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \end{aligned}$$

The data transfer has occurred through a single reaction, and $\mathbf{CF}_u$ is now in a state in which $T_7$ is required to activate it once again for a further data transfer. Although $T_7$ can be applied, for brevity we do not consider this possibility.

At this point, reaction $T_1$ can occur again, modeling the passing of data from **AD** to **GZ** for compression. $T_1$, in this case, acts upon the subsolution consisting of molecules $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$.

$$\begin{aligned} &S_2 \xrightarrow{T_1} S_3, \text{ where} \\ &S_3 = \mathbf{CF}_u \diamond o(1), \\ &\quad\;\; \mathbf{CF}_d \diamond i(4), \\ &\quad\;\; \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2), \\ &\quad\;\; \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \end{aligned}$$

From this state, any one of the three reactions $T_2$, $T_3$, $T_4$, or $T_7$ can occur nondeterministically.

In summary, this CHAM models an abstract machine that evolves dynamically from one admissible state to another, starting from the initial solution $S_0$ representing the initial configuration of the system and using the transformation rules $T_1$ through $T_9$ to model the possible behaviors of the system. This kind of architectural description lends itself naturally to formal examinations of behavioral properties.

### 3.3.2 Darwin

Darwin is intended to be used for the specification, construction, and management of software systems, with emphasis on the description of concurrent, distributed, and dynamic systems. Studies have been conducted involving configuration of distributed user programs, multiservice networks, and embedded systems.

Darwin describes a system topology as a hierarchical structure comprised of composite components. At the lowest level, a composite component will be composed of primitive components. Darwin does not directly provide a means for modeling component interaction or overall system behavior. In addition to the basic structural view described in Darwin, behavioral, service, and performance views are suggested for analysis and as aids during the implementation decision-making process.

- The *structural view* describes components and their interconnections. Components are specified as sets of portals (the Darwin term for "ports"), instantiations of components, and bindings between the portals of the instantiated components.
- The *behavioral view* is specified in FSP notation and analyzed for safety and liveness properties using the TRACTA [17] compositional reachability analysis technique. The Labelled Transition System Analyzer (LTSA) is used to perform the TRACTA analysis. LTSA is further discussed in Section 4.1.2.
- The *service view* is a refined view of the architecture aimed toward implementation. It includes information about the portals, such as binding patterns and whether a portal provides or requires a service.
- The *performance view* is suggested as an important area for future work.

Darwin specifications can be given as simple box and arrow diagrams or can be more elaborately described in the Darwin textual notation, which provides conditionals and iterators. Examples of both types of description are shown in figures 8 and 9.

Kramer and Magee [28] used Darwin to study the the Ring Database problem [45]. In this problem it is possible for inconsistent data to result from timing issues. Figure 9 contains a Darwin

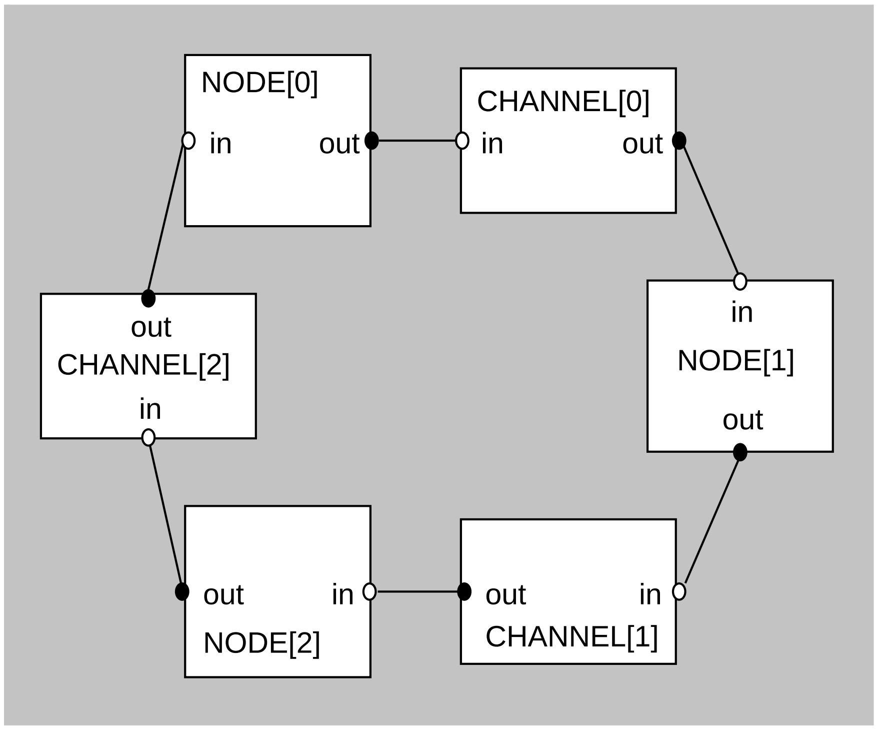


Figure 8: The Ring Database.

specification for a ring database. A graphical version is shown in Figure 8 and a behavioral description is given in Figure 10. The system RING is described as a composite component that binds together an unspecified number of primitive components CHANNEL and NODE. Communication between the components is defined in the interface UPDATE. The inter-component connections are described as a sequence of bindings between interface portals named "in" and "out" of the two types of components. The interface passes the name of the originating node and a value through the portals.

The LTS behavioral description for an instance of RING consisting of three nodes and three channels is given in Figure 10. As noted above, an LTS is described in the FSP notation. In order to provide a basic understanding of the FSP notation, we describe the semantics of the following symbols used in the example.

- $x \rightarrow P$ indicates that when a process engages in action $x$ it will then behave as process $P$.
- $x \rightarrow P|y \rightarrow Q$ indicates that a process may either engage in $x$ or $y$ and the resultant behavior will be $P$ in the former case and $Q$ in the latter case.
- In the behavioral description of NODE, the *quiet1*, *quiet2*, and *display* actions are used to define when the values of local variables can be read and used.
- The last action of component NODE is a guarded command. The operator "|" indicates that if one of the five conditions listed becomes true, the resultant behavior will be that described to the right of the $\rightarrow$.

```
component RING {
forall i = 0 to N-1 {
  inst  node[i]:NODE; channel[i]:CHANNEL;
  bind  node[i].out--channel[i].in;
        node[i].in--channel[((i-1)+N)%N].out
        }
}

interface
   UPDATE{Nodes; Value;}

component CHANNEL
   portal in:UPDATE;
   portal out:UPDATE;
}

component NODE
   portal in:UPDATE;
   portal out:UPDATE;
}
```

Figure 9: Darwin Description of the Ring Database Structure [28].

- The notation "||" used before the behavioral description of RING indicates that this is to be a parallel composition of its components NODE and CHANNEL.
- The symbol "/" denotes a relabeling that is sometimes necessary, since bindings in Darwin are modeled as shared actions in an LTS.

The Software Architect's Assistant (SAA) [38] is a tool that provides support for both a textual and a graphical description of components and connections. As noted above, textual descriptions can include the use of conditionals and iteration in order to vary the topology of a system during run time. Work is in progress to provide support for behavioral analysis through integration of LTSA into SAA.

### 3.3.3 Rapide

Rapide is a high-level, event-based simulation language that provides support for the dynamic addition and deletion of predeclared components. Rapide descriptions are composed of type specifications for component interfaces and architecture specifications for permissible connections among the components of a system.

System behavior is described through architectural connection rules, state transition rules, and patterns of events required to generate events that activate the rules. System behavior can be simulated through execution of the Rapide description. The results of a simulation of system behavior can be studied using a representation called a *poset*. A poset is a partially ordered set of events captured during a single simulation of a system.

Components are defined in terms of their interfaces. Three types of components are described in Figure 11, which is the Rapide description of the familiar gas station problem. The component types are a pump, a customer, and an operator. In this simple example we see that interfaces

```
const N      = 3 \\ number of nodes
range Nodes  = 0..N-1
const Max    = 2 \\ Update values
range Value  = 0..Max-1

const False  = 0
const True   = 1
range Bool   = False..True

CHANNEL      = (in[j:Nodes][x:Value]
             → out[j][x] → CHANNEL).

NODE(I=0) = NODE[0][FALSE],
NODE[v:Value][updt:Bool] =
   /* one local update at time - set value to u */
   (when (!updt) local[u:Value] → out[I][u] → NODE[u][True]

   /* locally passive - display consistent value */
   |when (!updt) quiet1 → display[v] → quiet2 → NODE[v][updt]

   /* receive update from originator j and value x */
   | in[j:Nodes][x:Value] →
      ( when (I==j && updt)    complete   → NODE[v][False]
      | when (I==j && !updt)   oops       → ERROR
      | when (I<j && updt)     discard    → NODE[v][updt]
      | when (I>j && updt)        out[j][x]  → NODE[x][False]
      | when (!(I==j) && !updt)   out[j][x]  → NODE[x][False]
      )
   ).

|| RING = ( node[i:Nodes]:NODE(i)
        || channel[Nodes]:CHANNEL)
   /{forall[i:Nodes] {
   node[i].out/channel[i].in,
   node[i].in/channel[((i-1)+N)%N].out,
   quiet1/node[i].quiet1,
   quiet2/node[i].quiet2
        }
   }.
```

Figure 10: FSP LTS Description of the Ring Database Behavior [28].

```
type Dollars is integer; -- enum 0, 1, 2, 3 end enum;
type Gallons is integer; -- enum 0, 1, 2, 3 end enum;

type Pump is interface
action in  O(), Off(), Activate(Cost : Dollars);
       out Report(Amount : Gallons, Cost : Dollars);
behavior
    Free : var Boolean := True;
    Reading, Limit : var Dollars := 0;
    action In_Use(), Done();
begin
    (?X : Dollars)(On ~ Activate(?X)) where
    $Free ||> Free := False; Limit := ?X; In_Use;;
    In_Use ||> Reading := $Limit; Done;;
    Off or Done ||> Free := True; Report($Reading);;
end Pump;

type Customer is interface
action in  Okay(), Change(Cost : Dollars);
       out Pre_Pay(Cost : Dollars)Okay(), Turn_On(), Walk(), Turn_Off();
behavior
       D : Dollars is 10;
begin
       start ||> Pre_Pay(D);;
       Okay ||> Walk;;
       Walk ||> Turn_On;;
end Customer;

type Operator is interface
action in  Request(Cost : Dollars), Result(Cost : Dollars);
       out Schedule(Cost : Dollars), Remit(Change : Dollars);
behavior
       Payment : var Dollars := 0;
begin
       (?X : Dollars)Request(?X) ||> Payment := ?X; Schedule(?X);;
       (?X : Dollars)Result(?X) ||> Remit($Payment - ?X);;
end;

architecture gas_station() return root is
    O : Operator;
    P : Pump;
    C1, C2 : Customer;
connect
    (?C : Customer; ?X : Dollars) ?C.Pre_Pay(?X) ||> O.Request(?X);
    (?X : Dollars) O.Schedule(?X) ||> P.Activate(?X);
    (?X : Dollars) O.Schedule(?X) ||> C1.Okay;
    (?C : Customer) ?C.Turn_On ||> P.On;
    (?C : Customer) ?C.Turn_Off ||> P.Off;
    (?X : Gallons; ?Y : Dollars)P.Report(?X, ?Y) ||> O.Result(?Y);
end gas_station;
```

Figure 11: Rapide Description of the Gas Station Example [56].

specify several aspects of the component's interactions with other components. The declaration of *in* and *out* actions specify the component's ability to observe or emit particular events. Implicitly declared actions represent events generated in the environment of the system that are emitted by or watched for in an interface; the event `start` in the first transition rule of the customer interface in Figure 11 is an example. Behaviors, which may involve local variables, describe the computation performed by the component, including how the component reacts to *in* actions and generates *out* actions. Computations are defined in an event pattern language [57], where a pattern is a set of events together with their partial ordering. The partial order of events is represented as a poset.

In addition to the concepts of actions and behaviors, interfaces can be described using several other constructs, including functions and constraints. Functions describe normal synchronous behavior. Constraints provide an opportunity to describe the patterns of events that either must or must not have occurred prior to the triggering of some event.

Posets are a critical element of Rapide. The poset is used both to specify in an interface an allowable pattern of events and to record the actual events occurring in a particular simulation. Posets represent causal and timing dependencies and are the basis for the dynamic behavioral analysis performed by the Rapide tools. A Rapide poset representing a simulation of the gas station example is shown in Figure 13. Support for static dependence analysis is provided for Rapide specifications by Aladdin [54].

In Rapide, component types are instantiated and then connected to form architectures. The architecture declaration at the bottom of Figure 11 instantiates the gas station components shown graphically in Figure 12. In this graphical view, the dotted arrows within the components portray the summarization of intra-component behavior that is described in the behavior section of the interface definition.

The semantics of connections between architectural elements are specified through rules. Connection rules have a trigger, an operator, and a body. Rapide provides four kinds of connections in connection rules. The example shown here uses only the agent connection (written syntactically as "`||>`"). In an agent connection, the observation of the pattern described in the trigger stimulates the events in the body.

### 3.3.4 MetaH

MetaH is tailored to the development of real-time, fault tolerant systems with primary emphasis on avionics applications. MetaH provides support for the specification of entire systems, including software modules, hardware components, and the bindings and interactions among them. The style of specification is iterative, beginning with partial specifications based on system requirements and continuing to lower levels of refinement in the form of source objects. While MetaH is similar to other structured design and object-oriented notations, significant additional capabilities are supported, such as hierarchical specification of both software and hardware components, and automatic generation of the glue code to combine predefined software and hardware components into a complete application.

A key aspect of the MetaH language is its support for analysis at the earliest stages of application development based on information supplied by system architects about expected or required properties, both functional and extra-functional. These properties are supplied in the form of *attributes*, and concern schedulability, reliability, safety, and security.

Each MetaH object is defined in terms of its interface and an implementation of that interface. Object implementations can be hierarchical, containing other objects and connections among them. Three basic MetaH language constructs include packages, processes, and applications. An example

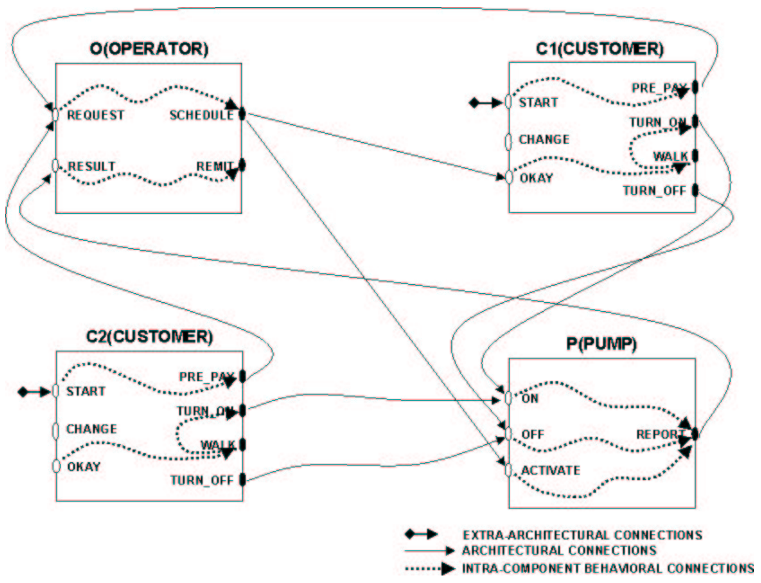


Figure 12: Rapide Graphical View of the Gas Station Example.

application description is shown in Figure 14. A type package, two processes, and an application are specified. The graphical description is shown in Figure 15. The application is specified after all of its components have been specified or have been predefined and included in the set of available components. An application is described in terms of the software that is to be deployed on a specific type of hardware. In this case the software is a macro, a set of processes, called `M.SIMPLE`, and the hardware is an I80960MC.CVME_TIMED processor. `M.SIMPLE` is an implementation of the `M` macro interface. It is composed of two *periodic* processes, `P1` and `P2`. A periodic process is one that is dispatched on a regular basis. In contrast, an *aperiodic* process is one that is dispatched upon receiving an event from another process or a hardware component.

Components describe processes, hardware, in/out ports, in/out events, and types. Implementation specifications and application specifications are defined as precisely as desired in terms of connections among the ports and events of components, execution pathways, and attributes. In `SIMPLE` each process defines three attributes, the macro defines the connections between the two processes, and the application defines an additional application-level attribute for the processor clock period. Over 100 predeclared attributes have been defined in the language for describing the allowed behavior of software and hardware components. These attributes are the primary source of information for analyses supported by MetaH.

### 3.3.5 Unicon

Unicon provides support for the composition of systems from components and connectors in a "plug and play" style of architecting. Like Darwin, Unicon places emphasis on the structural aspects of system description. The overall objectives for the development of Unicon are to provide a basis for studying a linguistic approach to architecture description and to gain knowledge of the design trade offs in various system properties.

The language contains four basic elements: components, connectors, players, and roles. Components are specified through interfaces and connectors are specified through protocols. Component

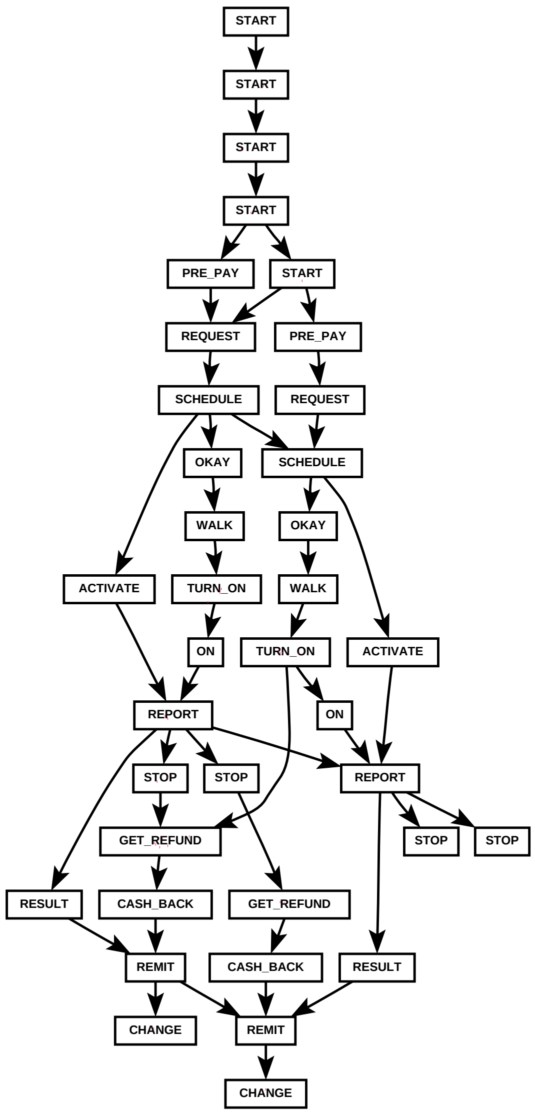
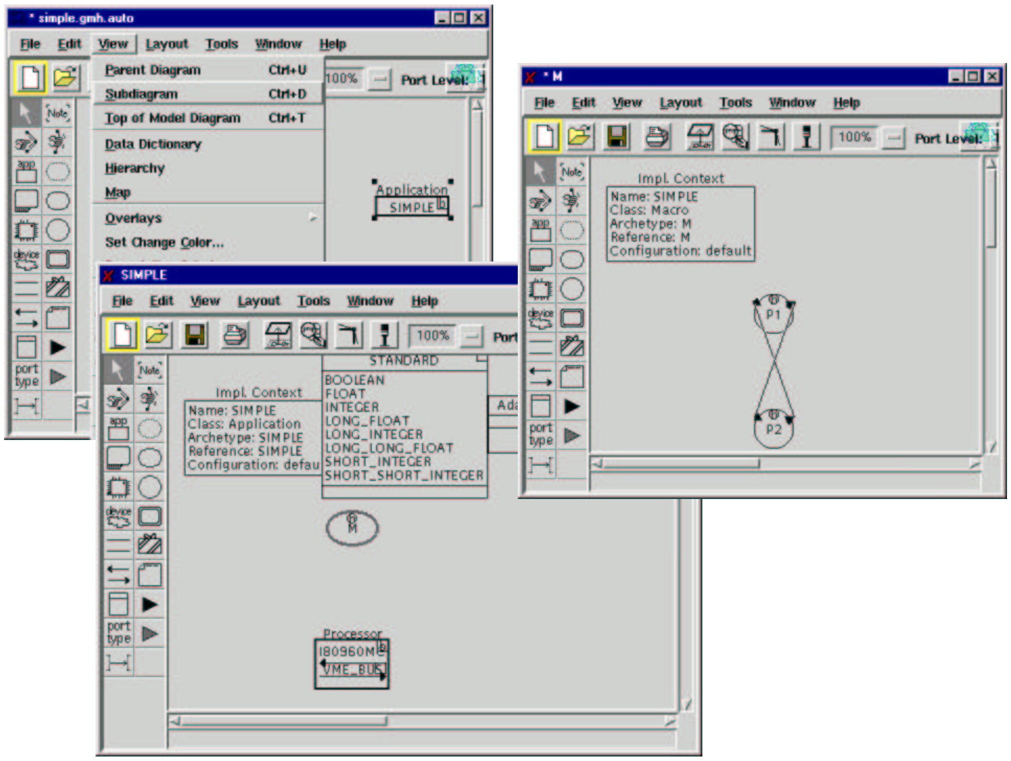


Figure 13: Rapide Poset for a Simulation of the Gas Station Example.

```
type package DOMAIN_TYPES is
   INTEGER_TYPE: type;
end DOMAIN_TYPES;

with type package DOMAIN_TYPES;
process P1 is
   FROM_P2 : in port DOMAIN_TYPES.INTEGER_TYPE;
   TO_P2 : out port DOMAIN_TYPES.INTEGER_TYPE;
end P1;

periodic process implementation P1.SIMPLE is attributes
   self'SourceTime := 100 us;
   self'Period := 1 sec;
   self'SourceFile := "p1.a";
end P1.SIMPLE;

with type package DOMAIN_TYPES;
process P2 is
   FROM_P1 : in port DOMAIN_TYPES.INTEGER_TYPE;
   TO_P1 : out port DOMAIN_TYPES.INTEGER_TYPE;
end P2;

periodic process implementation P2.SIMPLE is attributes
   self'SourceTime := 50 us;
   self'Period := 1 sec;
   self'SourceFile := "p2.a";
end P2.SIMPLE;

macro M is end M;

macro implementation M.SIMPLE is
   P1 : periodic process P1.SIMPLE;
   P2 : periodic process P2.SIMPLE;
connections
   P1.FROM_P2 <- P2.TO_P1;
   P2.FROM_P1 <- P1.TO_P2;
end M.SIMPLE;

application SIMPLE is
   macro M.SIMPLE on processor I80960MC.CVME_TIMED;
attributes
   I80960MC'ClockPeriod := 500 ms;
end SIMPLE;
```

Figure 14: MetaH Description for SIMPLE [61].



Impl. Context
Name: SIMPLE
Class: Application
Archetype: SIMPLE
Reference: SIMPLE
Configuration: defau

Impl. Context
Name: SIMPLE
Class: Macro
Archetype: M
Reference: M
Configuration: default

Figure 15: Graphical Description of SIMPLE Using DoME.

interfaces and connector protocols contain their type, their means of interaction with other architectural elements, and sets of properties. They are also associated with C language implementations. Components may be primitive or may be composite instantiations of systems. Components interact through their players and connectors interact through roles. Thus, the players and roles are the source of interaction among the components and connectors of the system. Properties specify attributes, assertions, or constraints related to the language elements.

Components (but not connectors) can be defined by the designer, thus supporting flexibility in design. Unicon provides both graphical and textual interfaces for describing architectures, and these interfaces are inter-convertible. The graphical interface allows a designer to select components and connectors from predefined menus. Some support is provided for interface compatibility checking.

The example shown in Figure 16 describes a Stack component [66]. A Stack is a component of type Computation. Other predefined component types include Filter, SharedData, SeqFile, Process, SchedProcess, and General. The component types describe behavior, functionality, performance characteristics, and expected interaction style. The interface section of a component description may contain a list of properties and a list of players. Stack has no properties but has four players. Stack is evidently implemented through the use of the "stack.c" Source file. Other implementation types available are Object, Executable, ObjectLibrary, Include, and Data.

The definition of Stack is a subcomponent for the Reverser component that is described in Figure 17 as a filter whose interface has three players. Its implementation is a composite implementation that describes the use of three components (reverse, stack, and libc) and one connector (datause). The implementation also contains a list of connections, role-player associations, and bindings.

### 3.3.6 Wright

Wright is intended to provide a formal basis for describing and analyzing software architectures. The primary objectives of the language are to provide a practical language that is usable to describe real systems by real system designers, to be solidly founded on a theoretic basis, to include abstraction for system behavior as well as structure, and to provide a basis for describing families of systems.

A Wright description provides for the specification of both the structure and the behavior of a system. The structural specification is comprised of a set of components, a set of connectors, and the system configuration. Components describe computations, connectors describe means of component interaction, and configurations describe how the components and connectors are attached. Each component has one or more *ports* that represent points of interaction with other components. Connectors have special ports called *roles*. Interaction occurs between two or more components by placing a connector between them and by associating each port in a component with a role in the connector.

The high-level structure of a simple client-server relationship is depicted in Figure 18. The Wright description of the client-server is shown in Figure 19. This example is adapted from an example given by Allen and Garlan [1]. The client-server example describes two components, one connector, and a configuration. Only the behavior of C-S-connector is shown in the figure.

Wright behavioral descriptions are given in a subset of CSP. CSP is an event-based specification language for describing concurrent processes [20]. The symbols used in this simple example describe communication entities basic to the description of software systems. The meaning of these symbols is as follows.

- $e?x$ (input operator)—$x$ is an input value for event $e$, where $e$ can be interpreted as a communication channel.

```
COMPONENT Stack {
     INTERFACE IS
          TYPE Computation

          PLAYER stack_init IS RoutineDef
               SIGNATURE (;‘‘void’’)
          END stack_init
          PLAYER stack_is_empty IS RoutineDef
               SIGNATURE (; ‘‘int’’)
          END stack_is_empty
          PLAYER push IS RoutineDef
               SIGNATURE (‘‘char *’’; ‘‘void’’)
          END push
          PLAYER pop IS RoutineDef
               SIGNATURE (‘‘char * *’’; ‘‘void’’)
          END pop
     END INTERFACE

     IMPLEMENTATION IS
          VARIANT stack IN ‘‘stack.c’’
               IMPLTYPE (Source)
          END stack
     END IMPLEMENTATION
END Stack
```

Figure 16: Unicon Description of Stack [66].

```
COMPONENT Reverser {
     INTERFACE IS
          TYPE Filter

          PLAYER input IS StreamIn
               SIGNATURE (``line'')
               PORTBINDING (stdin)
          END input
          PLAYER output IS StreamOut
               SIGNATURE (``line'')
               PORTBINDING (stdout)
          END output
          PLAYER error IS StreamOut
               SIGNATURE (``line'')
               PORTBINDING (stderr)
          END error
     END INTERFACE

     IMPLEMENTATION IS
               ⋮
          USES stack INTERFACE Stack
               ⋮
          CONNECT reverse._iob TO datause.user
               ⋮
          ESTABLISH C-proc-call WITH
               reverse.stack_init AS caller
               stack.stack_init AS definer
          END C-proc-call
               ⋮
          BIND output TO ABSTRACTION
               MAPSTO (reverse.fprintf)
          END output
     END IMPLEMENTATION
END Reverser
```

Figure 17: Unicon Description of Reverser [66].

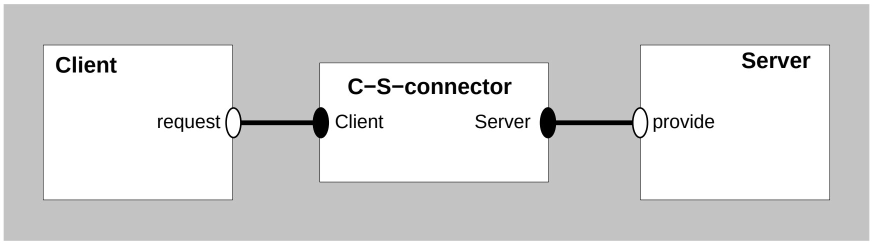


Figure 18: A Simple Client-Server Relationship.

```
System Simple Example
    Component Server
        Port provide {provide protocol}
        Spec {Server specification}
    Component Client
        Port request {request protocol}
        Spec {Client specification}
    Connector C-S-connector
        role Client = request!x → result?y → Client ⊓ §
        role Server = invoke?x → return!y → Server[]§
        glue = Client.request?x → Server.invoke!x
                  → Server.return?y → Client.result!y → glue[]§
Instances
    s:  Server
    c:  Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client;
end SimpleExample.
```

Figure 19: Wright Description of Simple Client-Server [1].

- $e!y$ (output operator)—$y$ is an output value for event $e$, where $e$ can be interpreted as a communication channel.
- $e \to P$ (prefixing operator)—a process engages in event $e$ and becomes process $P$.
- $P \sqcap Q$ (decision operator)—a process nondeterministically behaves like process $P$ or process $Q$.
- $P [] Q$ (alternative operator)—a process behaves like process $P$ or process $Q$ with the choice made by other processes that interact with the process.
- §—(termination operator) a process is able to terminate.

A notion of consistency is introduced via a behavioral equivalence between the CSP agents describing the semantics of corresponding ports and roles.

In the client-server example, the client sends a request with data element $x$ that produces a result with data element $y$. Because the operator $\sqcap$ is used, the client itself chooses, nondeterministically, whether to repeat the process or to terminate. On the other hand, the operator $[]$ appearing in the declaration of Server states that the determination of whether it is repeatedly invoked or terminated is made by other processes in its environment, namely Client. The glue describes the allowed sequence of events between Client and Server. If the client makes a request of the server it expects a result to be returned. After the client receives the result it decides whether to make another request or to terminate the session. The server will provide results as long as the client makes requests.

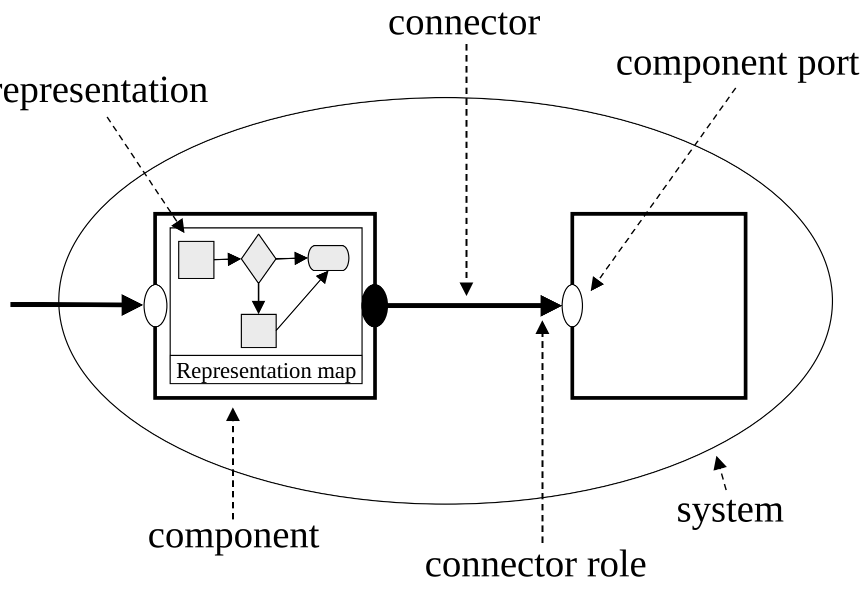


Figure 20: Seven Elements of an Acme Architecture.

### 3.3.7 Acme

The development of ADLs is still at a point where there is a general lack of agreement on the full set of linguistic concepts required to describe software architectures. Nevertheless, there is an emerging consensus on a core set of concepts that primarily have to do with the structural aspects of software architectures. This is quite evident even from the brief review of the five ADLs given above.

Recognizing this emerging consensus, the designers of the Acme language are attempting to represent a useful intersection of existing ADLs as a means to support some degree of interoperability among their associated tools [14]. Additional goals for the language include providing a descriptive standard for architectural tools, assistance in the development of new ADLs, and a language that is accessible to most system developers.

The language provides seven basic constructs for describing software architectures (Figure 20).

- *System*—describes the configuration of the components and connectors.
- *Component*—represents the primary computational elements and data stores of a system.
- *Component port*—identifies a point of interaction between a component and its environment.
- *Connector*—represents the means of interaction among components.
- *Connector role*—defines a participant of the interaction represented by a connector.
- *Representation*—represents hierarchical architectural descriptions.
- *Representation map*—a mapping from a representation's interface to that of the system component or connector that it represents.

To some extent, the seven basic Acme constructs serve as a “least-common denominator” and, therefore, some important language-specific concepts cannot be directly represented. As mentioned above, the orientation of the constructs is toward structure, so the missing concepts center on behavior. An additional Acme construct, *Properties*, allows an architect (or developer of an ADL-to-Acme translator) to describe properties that may be useful for analysis purposes, but not representable using the other constructs. As an example, if one wishes to use Aladdin [54] to reason about the dependencies in a system, a path property can be specified to indicate the ability of an input to contribute to the stimulation of an output in some way.

Garlan and Wang investigated the use of Acme as an architecture interchange language [16]. The purpose of their case study was to investigate the feasibility of combining the individual analytical strengths of Aesop, Wright, and Rapide in order to provide a variety of analysis capabilities in one environment. Aesop provides analysis of architectural styles, Wright provides static analysis techniques for proving properties, and Rapide provides for simulation and dynamic analysis. Difficulties were encountered involving differences in structural representation as well as semantics of the languages. While Acme, Aesop, and Wright share similar semantics, difficulties arose between these languages and Rapide. Among these difference is lack of support for defining connector types in Rapide and the lack of support for dynamic addition and deletion of components in Wright. The authors concluded that it would not be possible to achieve total interchange between Wright and Rapide. They chose to concentrate their efforts on including as large a subset of the features of all three languages as possible. To do this they limited the types of systems that would be appropriate for interchange, created a semantic-based translator to map Wright specifications into Rapide, but did not attempt to create a Rapide to Wright translation. The authors conclude that while there remain many challenges in this area of research, architecture interchange based on Acme shows promise.

## 4 Formal Analysis of Software Architectures

Formal software architecture description languages allow one to reason about functional and extra-functional properties of software systems early in the software life cycle as well as at high levels of abstraction. Research in architectural analysis centers on determining which specific properties are appropriate for this level of analysis, and on developing techniques to carry out those analyses. The premise underlying this work is that the confidence gained through analysis at an architectural level will translate into confidence in other levels of the system.

Many techniques for analyzing software systems have been developed over the past decades. Most, however, are ineffective for analyzing large systems. This is particularly true for techniques aimed at analyzing concurrent systems, where state explosion problems are especially acute. To make them more tractable, traditional specification and analysis techniques have been enhanced in a variety of ways. Software architecture can be seen as another approach to attacking the problem by providing a particular method for abstraction and modularization.

Automated analysis techniques can differ in the levels of assurance they provide. In general, the techniques trade off efficiency and tractability against precision and completeness. For instance, it may be possible to guarantee some properties only under certain assumptions or conditions. Carefully chosen, those assumptions and conditions can match well with the context in which the system is anticipated to operate, and thus the analysis can provide useful information.

A desirable characteristic of any imprecise or incomplete analysis technique used to examine a property is that it give no false positive results concerning that property. In other words, it should never indicate the absence of a problem when, in fact, there is a problem. On the other hand, it is reasonable to allow a technique to indicate the possible presence of a problem, even if none truly exists, and defer further analysis to some other automated analysis technique or to the human. This characteristic is commonly referred to as *conservatism*. Clearly, the most conservative analysis technique is one that indicates the possible presence of an error in all situations. Such an absurd technique, while highly efficient (it can be implemented using a constant function), is not of use. One goal of analysis research is to increase the precision of conservative techniques such that they are both efficient and useful.

Techniques to analyze the structural properties of a system, such as the proper connection of "out" ports to "in" ports, are well known and understood. In this section we concentrate on behavioral properties, such as deadlock freedom in communication protocols, by briefly reviewing recent research in three categories of analysis techniques: *proofs*, *sampling*, and *reduction*. We also briefly review recent work in analyzing extra-functional properties, such as performance.

As in our discussion of languages in Section 3, we present information about architectural analysis in breadth rather than at depth. The techniques are rather complex, making it impossible for us to convey more than just a sense of what the techniques can do, how they are used, and why they are applicable to software architectures. Our goal is to provide a basic survey of recent trends in research on architectural analysis as a starting point for a further examination of the topic. In presenting the analysis techniques, we mainly base the discussion on the languages of Section 3. Note that although we discuss the techniques in terms of their application to specific languages, they are generally applicable to whole classes of languages.

## 4.1 Proof Techniques

Proof techniques are typically used to examine properties having to do with system safety and liveness. Safety properties say that "nothing bad will ever happen",[3] whereas liveness properties say that "eventually something good will happen". The standard safety property of interest in concurrent systems is freedom from deadlock. Given a compositional approach to system construction, deadlock freedom becomes a serious challenge at the architectural level. The standard liveness property is continued computational progress of all, not just some, processing elements.

There are two main approaches to proving safety and liveness properties: algebraic techniques and transition-system techniques.

### 4.1.1 Algebraic Analysis

Algebraic analysis involves the application of structural induction techniques to an algebraic description a system.

#### CHAM

Compare et al. [7] used algebraic analysis to prove the presence of deadlock in the CHAM description of the compressing proxy shown in Figure 7. The deadlock in the system arises because of a mismatch in the protocol between **gzip** and its adaptor. In particular, **gzip** uses a one-pass compression algorithm and may attempt to write a portion of the compressed data (perhaps because an internal buffer is full) before the adaptor is ready, thus blocking. With **gzip** blocked, the adaptor

[3]Safety properties in this context should not be confused with properties associated with so-called "safety-critical systems". While safety-critical systems and their properties are important, very little work in software architecture has been targeted toward addressing their specific needs.

also becomes blocked when it attempts to pass on more of the data to **gzip**, leaving the system in deadlock.

The proof of the deadlock proceeds by reasoning about the algebraic structure of the specification. For example the first step is to observe that the application of any rule other than $T_8$ does not change the number of molecules or kind of processing elements in a solution but only transforms the state of the processing elements mentioned in the left-hand side of the applied rule. This fact is used to show that every derived solution will have exactly the same number of molecules as the initial solution, namely four, one for each of the four processing elements of the specification. The argument continues by reasoning about what it means for a derivation to terminate and that every derivation that terminates involves the application of rule $T_4$ or $T_5$. These rules model the situations in which **gzip** autonomously decides to end its input or output, thus eventually leading to a deadlock.

Of course, as general experience has shown, algebraic analyses can be tedious and complicated. But they are also highly informative, since they maintain information about the structure of the system, which can be useful in actually discovering the source of a problem. Moreover, when the system under specification leads to an explosion in the state space, then the use of algebraic techniques is the only practical approach to use.

### 4.1.2 Transition-System Analysis

Transition-system analysis is performed as a search over a graph-based model of a system's state space. As mentioned above, such an approach can be expensive due to the large numbers of states and possible state transitions that are characteristic of any reasonably sized system. Generally, an abstraction method is used to reduce the size of the model. After the model is constructed, properties, generally expressed as some type of automaton or in some form of temporal logic, are proved about the modeled system.

#### CHAM

One of the strengths of the CHAM formalism is that it admits, not only to algebraic analysis, but also to transition-system analysis [7]. In particular, the operational semantics of the CHAM can be used to automatically derive a transition system, where a node in the graph represents a unique solution (i.e., a state) and the arcs in the graph represent the application of a rule (i.e., a transition). A portion of the transition system for the compressing proxy system described in Figure 7 is depicted in Figure 21. The graph was produced by a tool developed to generate transition systems from CHAM architectural specifications [46].

Notice the node labeled $S_{76}$, which has no outgoing arcs and so represents a solution that from which no further progress can be made. This node represents a deadlock in the system. Examination of the full graph reveals the fact that all paths leading from $S_0$, the initial solution, to $S_{76}$ involve an arc labeled $T_4$ or $T_5$. This confirms the result achieved using algebraic analysis that the application of $T_4$ or of $T_5$ leads to deadlock.

#### Darwin

As discussed in Section 3.3, the Darwin ADL directly provides only for the specification of structural aspects of an architecture. The system's behavioral aspects can be described using a labeled transition system (LTS) in the FSP notation and analyzed using LTSA. An LTS is a graph whose nodes represent system states and whose arcs represent the ability to transition from one state to

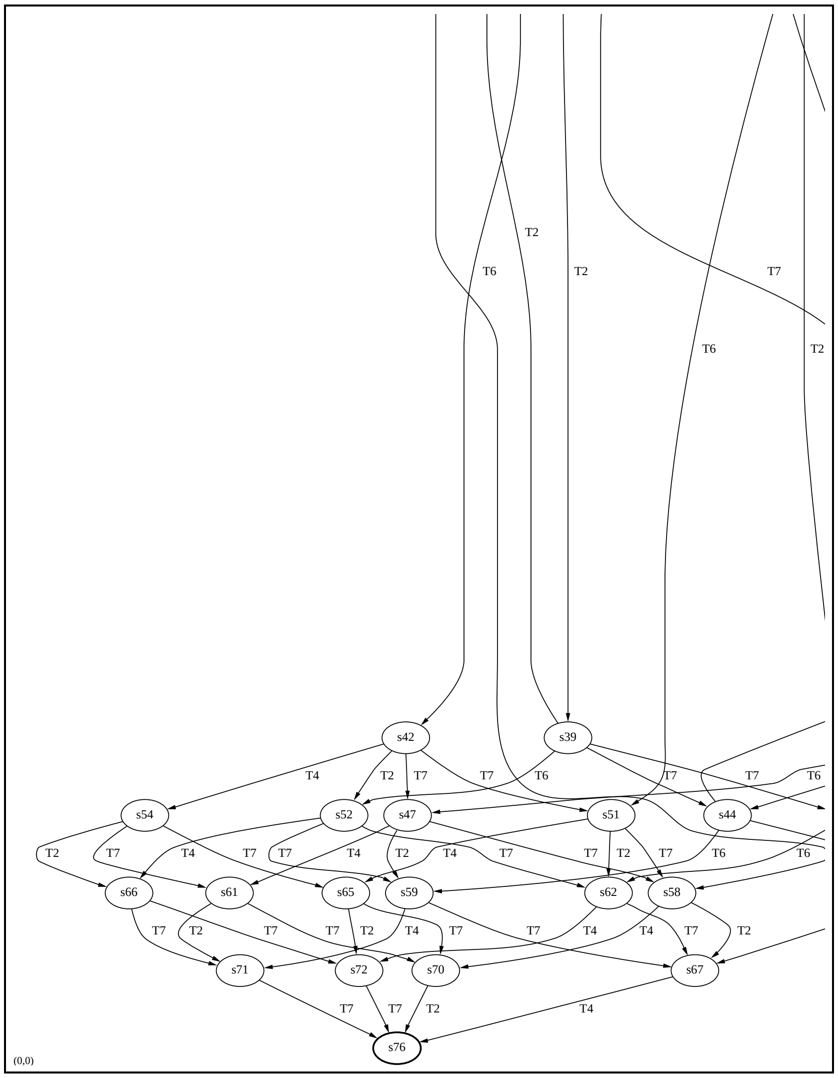


Figure 21: Excerpt from the Compressing Proxy Transition System [7].

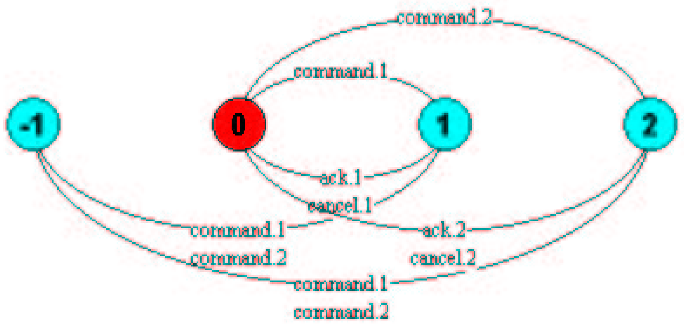


Figure 22: Results of LTSA Analysis for an Active Badge System [32].

another. The states are numbered beginning at zero, with the special state "-1" indicating an error state. Arcs are labeled by the name of the action that allows the transition between the states.

LTSA employs compositional reachability analysis (CRA) [64] as a means of managing state-space explosion. In CRA, components of a system are analyzed one at a time. Global analysis is performed by combining the results of the individual component analyses. For example, consider the description of a system that could include a word processor as a component. The interface of the word processor describes its allowable interactions with other components, such as the file system, invocation of a spell checker, and the like. The internal behavior of the word processor consists of various responses to inputs and internal actions. This internal behavior can be verified independently of the rest of the system, promising that if required inputs are available, the text processor will respond as in the outputs of its interface.

LTSA provides support for proving safety and liveness properties about a specification. Safety properties may be proved either explicitly by defining properties or by checking for error conditions. Liveness properties are specified in terms of Buchi automata and traces are checked for cycles that indicate the potential for liveness violations. Darwin and associated tools have been used to study the properties of several distributed systems [27, 28, 32]. Figure 22 shows the analyzed LTS for a portion of the Active Badge System [32]. In this system, it is an error if two commands may be processed at the same time. Notice that it is possible for `command 2` to be received before `command 1` is acknowledged or canceled, as indicated by the transitions from states 1 and 2 to the error state -1.

LTSA also provides another, less expensive global analysis technique. This alternative performs a depth-first search at the component level of the system description. The analysis can identify property violations and potential for deadlock, but does not provide information about the cause of the violation.

## Wright

An important aspect of the Wright language is the use of CSP in connector specifications to provide a mathematically sound basis for reasoning about the correctness of communication protocols. In particular, Wright provides a method for specifying and proving properties of individual connectors, and defining compatibility relationships through refinement. Provided that a connector is deadlock free, the compatibility relationship between roles and ports guarantees the preservation of deadlock

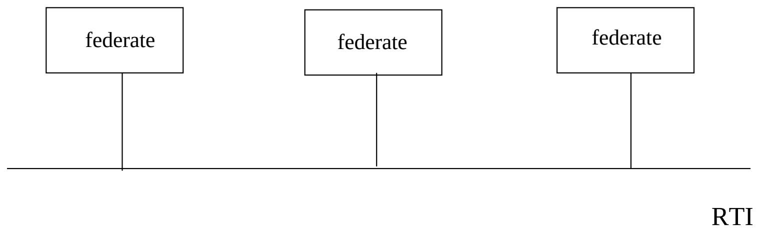


Figure 23: The HLA Integration Standard [3].

```
Connector RTI($nfeds$ : 1..)
   Role Fed$_{1..nfeds}$ = FederateInterface
   Glue = RTIBehavior

FederateInterface =
  FedMgmt || DeclMgmt || ObjMgmt || OwnMgmt || TimeMgmt
       || DataMgmt || FedJoined || ControlPause
where
  FedMgmt = ...
  DeclMgmt = ...
  ...
```

Figure 24: Extract from Wright Specification of RTI Connector [3].

freedom for that individual connector. To prove global deadlock freedom, the strategy followed in Wright is to first translate a Wright system configuration into a single CSP specification of the entire system and then to build the automata model from the CSP specification. Wright specifications can be analyzed for behavioral correctness using the FDR [10] model checker. FDR is a commercial tool that supports verification of specific properties in CSP process specifications.

The utility of connector types as an aid to high-level analysis and design is highlighted by Allen, Garlan, and Ivers [3] in a study involving the High Level Architecture (HLA), which was developed by the Defense Modeling and Simulation Office (DMSO) for building distributed simulation systems. A graphical depiction of the HLA is shown in Figure 23. Structurally, this is a very simple architecture that involves some number of *federates* (simulations) communicating in restricted ways over a so-called simulation bus. The Run Time Interface (RTI) defines the allowed interactions among federates. An extract from the Wright description of the RTI connector is shown in Figure 24, where FederateInterface defines how an individual federate can communicate and RTIBehavior defines the inter-federate interactions. An excerpt of the behavioral description of the RTI is shown in Figure 25.

An analysis of the Wright specification identified several faults in the architecture, including race conditions and the potential for deadlock. A graphical depiction of a trace of events found during FDR analysis is shown in Figure 26. This trace represents the cause of a race condition involving PauseProtocol and HandlePauseResume, defined in Figure 25. The numbers in parenthesis indicate the event order for the trace. The problem, in this particular situation, is that event 4, a federate resignation (i.e., the removal of a federate from the simulation) occurred after event 2, which determines the federates currently joined, and before event 5, the actual call for a pause.

```
RTIBehavior = HandleMembership || JoinedFeds$_{\{\}}$ ||MiniProtocols
where
    HandleMembership = ...
    JoinedFeds$_S$ = (whoIsJoined!S $\rightarrow$ JoinedFeds$_S$)
      [] ([] $i:(1..nfeds)$@ Fed$_i$.joinFedExecution $\rightarrow$JoinedFeds$_{S\cup\{i\}}$)
      [] ([] $i:(1..nfeds)$@ Fed$_i$.resignFedExecution $\rightarrow$JoinedFeds$_{S\setminus\{i\}}$)
      [] §

MiniProtocols =
    FederationProtocols || DeclarationProtocols || ObjectProtocols
    || OwnershipProtocols || TimeProtocols || DataDistributionProtocols

FederationProtocols = PauseProtocol || ...
PauseProtocol = HandlePauseResume || PausedFeds$_{\{\}}$
HandlePauseResume =
      ([] $i:(1..nfeds)$@ Fed$_i$.requestPause $\rightarrow\overline{\text{whoIsJoined?S}}$ $\rightarrow$ $\overline{\text{whoIsPaused?T}}$ $\rightarrow$
          (; $j:(S\setminus T)$@ $\overline{\text{Fed}_j\text{.initiatePause}}$ $\rightarrow$ §) ; HandlePauseResume)
    [] ([] $i:(1..nfeds)$@ Fed$_i$.requestResume $\rightarrow$ $\overline{\text{whoIsJoined?S}}$ $\rightarrow$
          $\overline{\text{whoIsPaused?T}}$ $\rightarrow$ ResumeResponse$_{S==T,\ T}$)
    [] §
ResumeResponse$_{true,S}$ =
    (; $i:S$@ $\overline{\text{Fed}_i\text{.initiateResume}}$ $\rightarrow$ §) ; HandlePauseResume
ResumeResponse$_{false,S}$ = HandlePauseResume
PausedFeds$_S$ = ...

ObjectProtocols = HandleRegistrations || HandleRemoves ||HandleAttrOutOfScopes || ...

HandleRegistrations = ...
HandleRemoves =
      ([] $i:(1..nfeds)$@ Fed$_i$.deleteObject $\rightarrow$ $\overline{\text{whoIsJoined?S}}$ $\rightarrow$
          (; $j:(S\setminus\{i\})$@ DecideIfRemoveNeeded$_j$) ; HandleRemoves)
    [] ([]$i:(1..nfeds)$@ Fed$_i$.attrsOutOfScope $\rightarrow$
          DecideIfRemoveNeeded$_i$ ; HandleRemoves)
    [] (implicitOutOfScope?i $\rightarrow$ DecideIfRemoveNeeded$_i$ ; HandleRemoves)
    [] §
DecideIfRemoveNeeded$_i$ = § $\sqcap$ $\overline{\text{Fed}_i\text{.removeObject}}$ $\rightarrow$ §

HandleAttrOutOfScopes =
      ([] $i:(1..nfeds)$@ Fed$_i$.subscribeObjClassAttr $\rightarrow$
          DecideImplOutOfScope$_i$ ; HandleAttrOutOfScopes)

...

          DecideImplOutOfScope$_i$ ; HandleAttrOutOfScopes)
    [] ([] $i:(1..nfeds)$@ Fed$_i$.publishObjClass $\rightarrow\overline{\text{whoIsJoined?S}}$ $\rightarrow$
          (; $j:(S\setminus\{i\})$@ DecideOutOfScope$_j$) ;
          HandleAttrOutOfScopes)

...
```

Figure 25: Extract from Wright Specification of RTIBehavior [3].

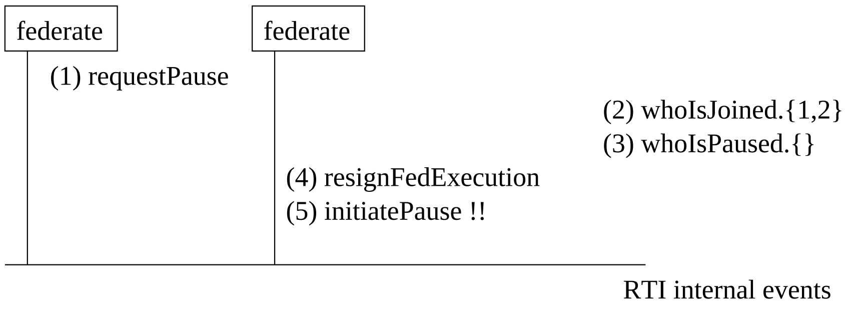


Figure 26: Race Condition with Resigning Federates [3].

This situation would create an attempt to communicate with the federate that is no longer joined. Potential deadlock was identified in two situations, one involved giving federates the right to refuse to pause and the other related to the right of a paused federate to decide whether it wishes to resume or not.

## 4.2 Sampling Techniques

When one wishes to gain confidence in the correctness of a system, but a full proof is too costly, or when one wishes to understand how a system will react under specific conditions, then sampling techniques provide an attractive alternative to the more general proving techniques discussed above. There are two main sampling techniques that have been applied to the analysis of software architectures: simulation and testing.

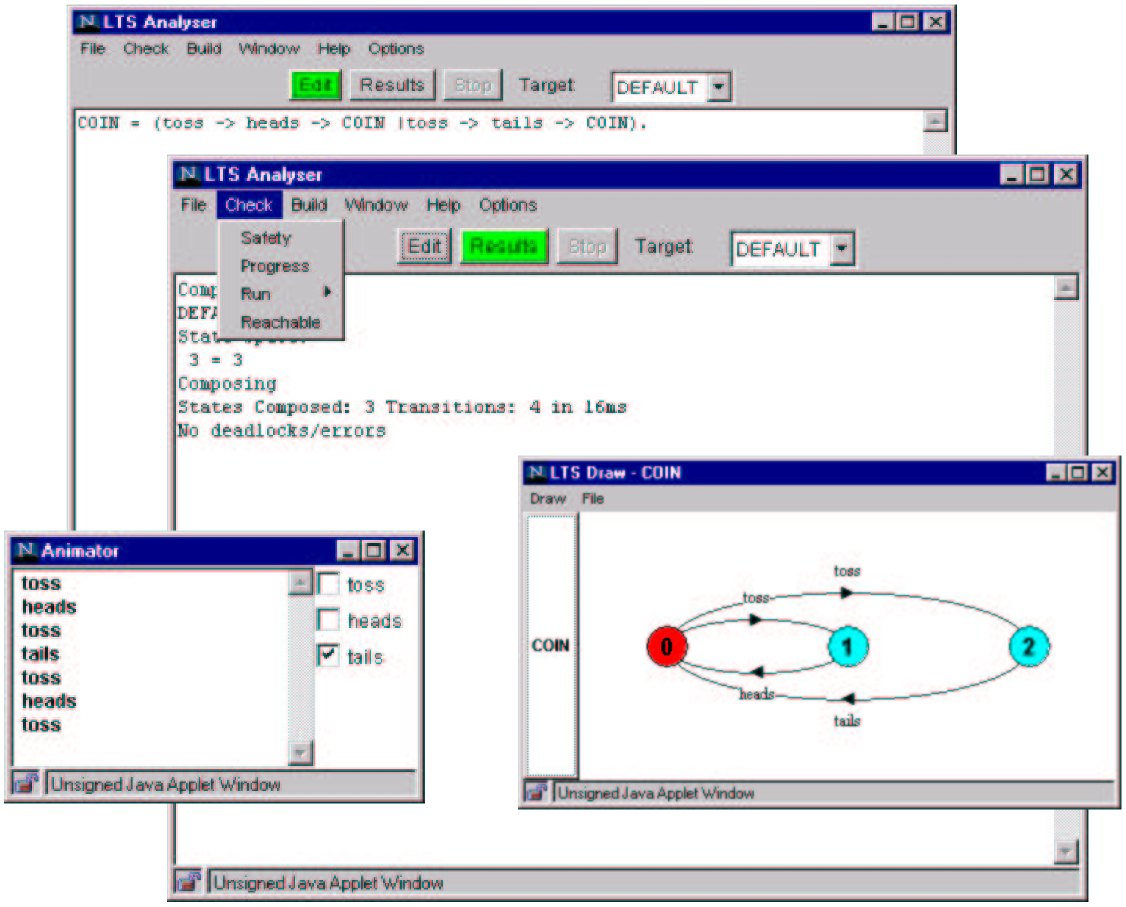
### 4.2.1 Simulation

Simulation allows one to directly observe the abstract behavior of a system without incurring the cost of a full implementation. Of course, because it is a simulation and not a real behavior, certain assumptions are being made to support the execution. The quality of the results of a simulation analysis are therefore heavily dependent on the quality of those assumptions.

**Darwin**

In addition to the proof analysis described in Section 4.1.2, LTSA supports a less expensive, scenario-based analysis. We use the simple example of a coin-toss system, taken from LTSA's user manual, to provide a flavor of this type of analysis. The FSP notation for the coin toss is entered into the editor window of LTSA. This is shown as the window to the back in Figure 27. The coin-toss system exhibits nondeterministic choice using the operator "`|`". We see from the specification that a toss of a coin will nondeterministically result in `heads` or `tails`, after which another toss of the coin can be made.

After the code is entered into the editor, the build menu can be used to create a transition system. The analyst can then use the check menu to run different analyses. The line "`No deadlocks/errors`" appearing in another of the windows shown in Figure 27 is the result of choosing "Safety" from the menu (see Section 4.1.2). The window "LTS Draw" shows a graphical version of the LTS.



```
COIN = (toss -> heads -> COIN |toss -> tails -> COIN).
```

```
Comp
DEFA
Stat
 3 = 3
Composing
States Composed: 3 Transitions: 4 in 16ms
No deadlocks/errors
```

```
toss
heads
toss
tails
toss
heads
toss
```

Figure 27: LTSA Simulation Analysis of a Coin-Toss System.

To simulate the coin-toss system, the analyst chooses "Run" from the check menu. This causes a tool called Animator to execute in its own window. Animator allows the user to step through a simulated execution, with nondeterminism controlled internally through a random-number generator. The list appearing on the left side of the animation window indicates the sequence of transitions taken to a given point during a particular simulation run. The choices on the right side of the animation window show the possible inputs to the system. A check mark indicates a valid input at that point in the simulation. Clicking on a checked item causes the input to be generated. The result of clicking on "tails" in the scenario shown will lead to the appearance of the word `tails` at the end of the list on the left side of the animator window. The check mark will then appear next to "toss", indicating that the coin is ready to be tossed again.

## Rapide

Rapide allows a designer to specify the behavior of a system either in the behavior section within an interface or, more elaborately, as a separate module that implements an interface. In either case, the Rapide description can be compiled and executed, which provides a simulation of the system's behavior under specific conditions. The resultant computation is captured as a partially ordered

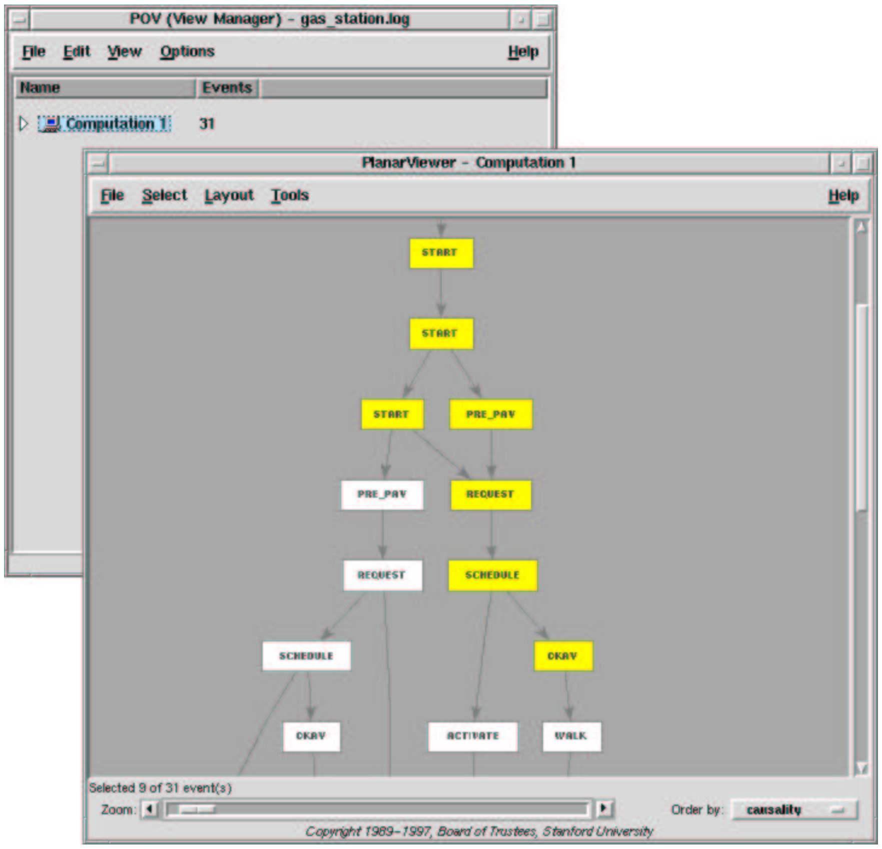

POV (View Manager) - gas_station.log
File Edit View Options Help
Name Events
Computation 1 31
PlanarViewer - Computation 1
File Select Layout Tools Help
START
START
START
PRE_PAY
PRE_PAY
REQUEST
REQUEST
SCHEDULE
SCHEDULE
OKAY
OKAY
ACTIVATE
WALK
Selected 9 of 31 event(s)
Zoom:
Order by: causality
Copyright 1989-1997, Board of Trustees, Stanford University

Figure 28: Rapide POV Tool Displaying the Poset for a Simulated Run of the Gas Station System.

set of events, the poset (see Section 3.3.3).

The poset is used to support simulation review, either as an animation over the structural view of the system in the tool Raptor, or as graphical representation in the tool POV. Figure 28 shows the use of POV to analyze an architectural description of a version of the gas station example. The file menu is used to select a log file that contains the poset resulting from a simulation run. Several different layouts are available for viewing, and other Rapide tools are accessible via the layout and tools menus. Once a file and a layout are chosen, a node can be selected and then the analyst can request to view the causal past or the causal future of the selected event. In the case shown in Figure 28, the causal past of event "OKAY" is highlighted, appearing as the darkened nodes in the poset.

### 4.2.2 Testing

The target artifact of traditional testing techniques is, of course, the implementation of a system. An architectural description can also be a suitable target if it contains enough behavioral information to permit simulated executions. The primary difference between simulation analysis, discussed above, and testing, discussed below, is essentially a matter of how the simulated executions are used. In particular, simulation analysis focuses on broadly illustrating or exemplifying system behavior, whereas testing analysis is aimed at carefully exercising the system behavior.

Testing techniques are typically applied at a variety of levels. Unit testing is performed on individual components to check their internal correctness. System testing is performed on a whole system to check its adherence to requirements. Between these two levels is integration testing, which checks for errors in the interworking of components in terms of design correctness, rather than requirements correctness. This is the level that appears to be most appropriate for use with architectural descriptions.

Because exhaustive testing is, in general, impractical, the analyst needs some way to objectively decide when testing should be terminated. Thus, the concern in testing is to achieve a *coverage* goal that can indicate the degree to which the system has been exercised. A critical challenge in developing testing analyses for software architecture descriptions is therefore the formulation of coverage criteria appropriate to that level of description [44].

#### CHAM

Bertolino et al. [6] devised a technique for using architectural descriptions as the basis for integration testing. The technique is applicable to any model from which a transition system can be derived. For their purposes, they found it convenient to present the technique in terms of the CHAM formalism.

The basic approach is to form subgraphs of the transition system graph that represent different views of the system behavior of interest to the analyst. Not coincidentally, the subgraphs represent different coverage criteria applied to the transition system graph. A completely covered subgraph indicates a satisfied criterion. Three of the subgraphs are the following.

- *Concurrent-reduced graph.* The CHAM model can lead to reaction rules being applicable to multiple, independent subsolutions of a given solution, which means that the behavior represented by the rules can occur either concurrently or in an arbitrarily interleaved fashion. The concurrent-reduced graph removes the interleaved transitions in favor of the concurrent one.

- *Input-reduced graph.* The behavior of a CHAM is typically regulated by one or more rules representing input to the system. Iteration of these rules govern the "amount" of input and, in many cases, the degree of concurrency in the system. Consider a pipeline architecture that is given only one portion of input to process versus two or more portions. The more portions of input fed to the system, the more potential for concurrency, up to some intrinsic maximum. The full transition system graph represents this maximum, while anything less than the maximum is a so-called input-reduced graph.

- *Regression-reduced graph.* This subgraph of the transition system graph is simply formed by considering only those states and transitions relevant to some particular component.

Bertolino et al. note that the input-reduced graph is a basis for a form of stress testing, while the regression-reduced graph is a basis for regression testing of components (hence its name).

## 4.3 Reduction Techniques

One way to reduce the cost of analyzing a large system is to simply reduce the amount of the system that needs to be analyzed. The reduction process is itself a form of analysis, but one that is more appropriately characterized as a "meta-analysis", since its result is intended to be input to another analysis technique, rather than a result in its own right. For example, at the heart of the testing technique of Bertolino et al., mentioned above, is a reduction meta-analysis. Of importance, however, is the fact that their technique is not intended to be conservative, but simply practical; it is possible that one of their reductions could hide an error that would otherwise be found. Here we are concerned with conservative reduction techniques.

The test of any conservative reduction technique is then a question simply stated and simply answered, but difficult to predict: will the cost of performing the meta-analysis be paid back by a commensurate reduction in the cost of the basic analysis? Often there is no general answer to this question, and answering the question on a case-by-case basis defeats its purpose. The uncertainty arises from the fact that the benefits of a reduction technique depend on the characteristics of the particular system being analyzed. What this leads to is a "meta-meta-analysis" that judges the suitability of a given system for reduction. In the end, to avoid an infinite and clearly unproductive recursion, the decision about whether to apply a reduction technique must be left to the skill and experience of the analyst.

### 4.3.1 Dependence Analysis

Dependence analysis involves the identification of interdependent elements of a system. It is a reduction technique in the sense that the interdependent elements induced by a given inter-element relationship forms a subset of the system. It has been widely studied for purposes such as code restructuring during optimization, automatic program parallelization, test-case generation, and debugging. Dependencies can be identified based on syntactic information readily available in a formal specification. This type of analysis generally ignores state information, but may incorporate some knowledge of the semantics of a language to improve the precision of the results [42].

Dependence analysis as applied to program code is based on the relationships among statements and variables in a program. Techniques for identifying and exploiting dependence relations at the architectural level have also been developed [53, 61, 67]. Dependence relationships at the architectural level arise from the connections among components and the constraints on their interactions. These relationships may involve some form of control or data flow, but more generally involve source structure and behavior. Source structure (or structure, for short) has to do with system dependencies such as "imports", while behavior has to do with dynamic interaction dependencies such as "causes". Structural dependencies allow one to locate source specifications that contribute to the description of some state or interaction. Behavioral dependencies allow one to relate states or interactions to other states or interactions. Both structural and behavioral dependencies are important to capture and understand when analyzing an architecture.

Aladdin [54] is a tool developed at the University of Colorado that identifies dependencies in software architectures. It can be used as a stand-alone tool or in conjunction with ADLs. It was designed to be easily integrated with ADL tool kits developed elsewhere, and is currently available for use in analyzing Acme and Rapide architectural descriptions.

If one thinks of an architectural description as a set of boxes and arrows in a diagram, where the arrows represent the ability for a box, or some port into or out of that box, to communicate with another box in the diagram, then one can think about Aladdin as walking forwards or backwards from a given box, traversing arrows either from heads to tails or vice versa. In Aladdin, the arrows are called *links* and the process of walking (i.e., performing a transitive closure) over the links is called *chaining*.

If there is no knowledge about how a box's input ports behaviorally relate to it output ports, then a forward (backward) walk must include leaps from each input (output) port that is reached to all output (input) ports. In that case, the analysis is essentially being performed in a conservative

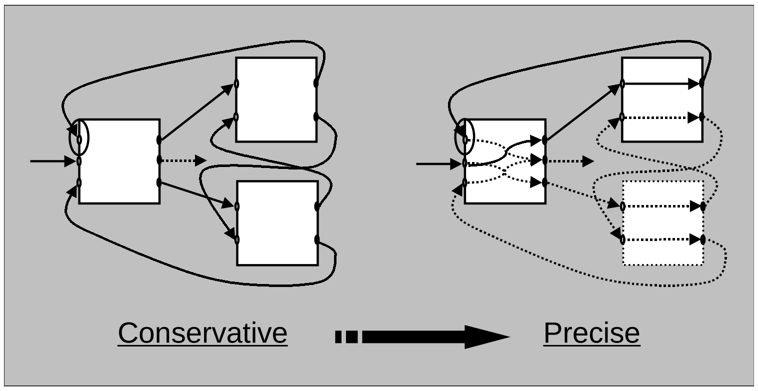


Figure 29: Increasing Precision of Dependence Analysis.

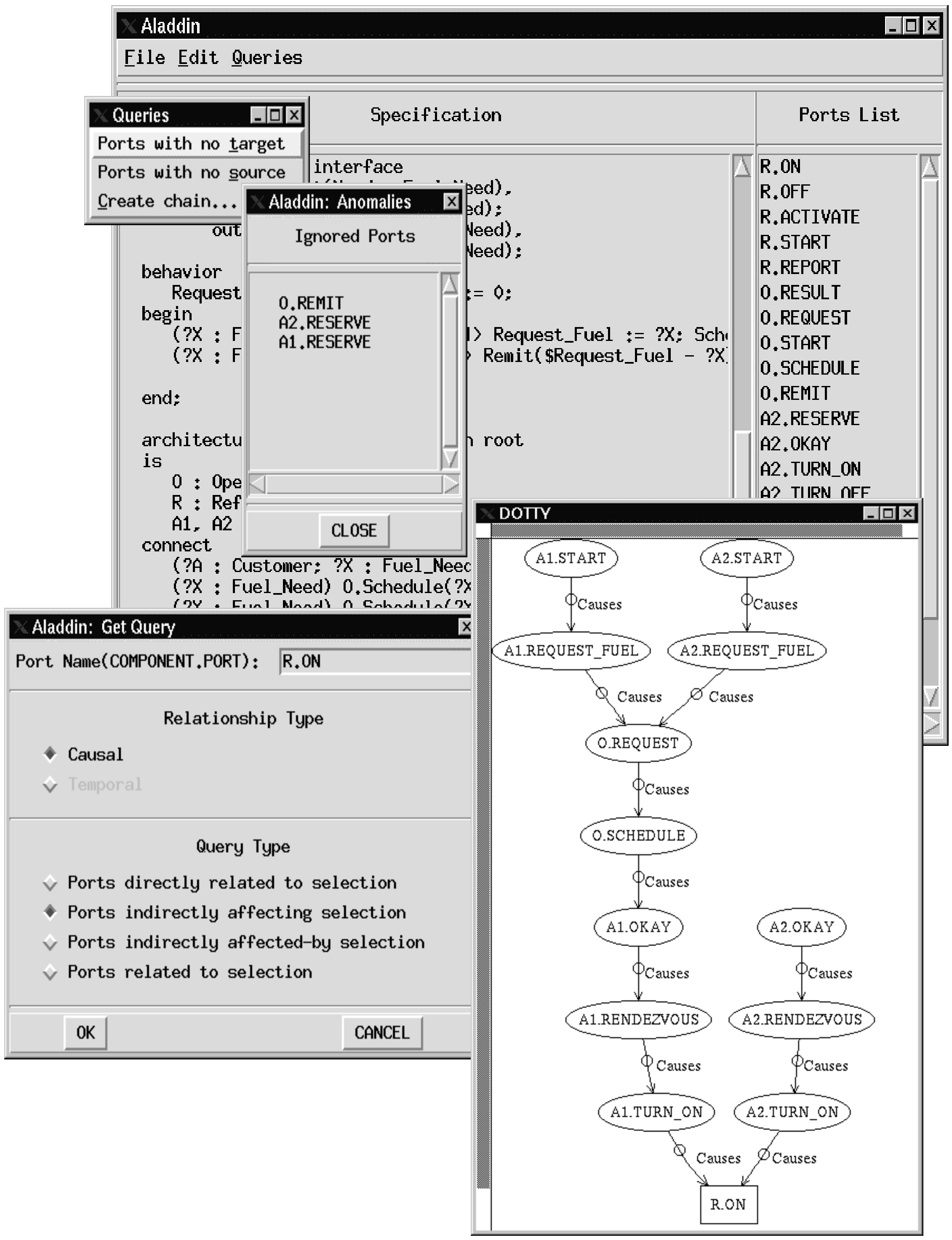
manor at the component level, which can lead to a high degree of false dependencies. If, instead, the designer makes a precise statement about how input and output ports are related, presumably using an appropriately rich ADL, then Aladdin can take advantage of this information to produce a more precise reduction set.

The behavioral relationship among the input and output ports of a component define the interaction behavior of that component. It is important to note that the interaction behavior is not intended to capture the functional behavior of the component. For example, the description of how a server interacts with its clients is independent of the computation carried out by the server on behalf of its clients. Aladdin uses a summarization algorithm operating on the description of a component's interaction behavior to identify possible relationships between pairs of input and output ports. The resulting connections are called *transitional connections.*

Figure 29 illustrates the improvement in precision that can be gained when transitional connections are included in the information used to determine possible dependencies. The solid arcs in this figure denote arcs that must be traversed in order to identify a conservative set of dependencies. In the view of the system shown on the left, the transitional connections are unknown. Therefore, when tracing back from the circled port, one must assume that any stimulus applied to input port could have contributed to a response on any output port. The lack of information on the interaction behavior of the component forces the analysis to include all components of the system in the dependency set. The existence of the transitional connections in the view of the system on the right provides information that allows the analysis to eliminate the component connected only by the dashed arcs.

Rather than constructing a complete dependence graph, Aladdin's analysis is performed on demand in response to an analysts query. The query might request information about the existence of certain specific kinds of anomalous dependence relationships, or might request information about the parts of the system that could affect or be affected by a specific port in the architecture. A view of Aladdin's interface is shown in Figure 30. A file containing a Rapide architectural specification is selected using the file menu. In this figure a specification for a variant of the gas station example was selected. The specification is displayed in the left pane of the main Aladdin window. The right pane displays the list of component ports that have been identified from the architectural description.

The analyst can select to perform any of several queries. The queries window shown at the top



```
interface
                     eed),
                     ed);
      out            Need),
                     Need);

  behavior
     Request         := 0;
  begin
     (?X : F         ) Request_Fuel := ?X; Sch
     (?X : F         > Remit($Request_Fuel - ?X)

  end;

  architectu         root
  is
     O : Ope
     R : Ref
     A1, A2
  connect
     (?A : Customer; ?X : Fuel_Need
     (?X : Fuel_Need) O.Schedule(?X
```

Ports List:

```
R.ON
R.OFF
R.ACTIVATE
R.START
R.REPORT
O.RESULT
O.REQUEST
O.START
O.SCHEDULE
O.REMIT
A2.RESERVE
A2.OKAY
A2.TURN_ON
A2.TURN_OFF
```

Ignored Ports:

```
O.REMIT
A2.RESERVE
A1.RESERVE
```

Port Name(COMPONENT.PORT): R.ON

Relationship Type: [✓ Causal] Temporal

Query Type:
- Ports directly related to selection
- [✓ Ports indirectly affecting selection]
- Ports indirectly affected-by selection
- Ports related to selection

Figure 30: Use of Aladdin to Identify Anomalies and Perform Port-Based Queries.

left in the figure appears when the analyst selects the "Queries" menu item. The analyst can choose to see a list of ports with no source or those with no target, which are two kinds of port-related anomalies. The small window to the right of the window "Queries" contains a list of all the ports in the specification that do not have targets. Ports with no source or no target may indicate an unspecified connection or they may indicate a function of the component that is not used in this particular architecture.

The analyst can also choose to create a chain. If "Create chain..." is selected, then the window "Get Query" appears. The analyst selects a query, in this case the analyst wanted to see a chain of all the ports in the architecture that could causally affect port `R.ON`. Dotty [11], a graph layout tool, is used to display the resultant chain, which appears in the window "Dotty". The chain is displayed as a directed graph rooted at the node representing the specified port of interest, in this case the node `R.ON` at the bottom of the graph. The arcs are labeled with a relationship type and represent direct (or perhaps summarized) dependence relationships between pairs of ports. The nodes of the graph represent all ports that could cause, directly or indirectly, the port of interest, the event `R.ON`, to be triggered.

This query was performed in order to help identify the cause of a failure in a Rapide simulation of the gas station. In the simulation it was discovered that `A2` was never allowed to refuel. The cause of this is apparent from viewing the chain, and in fact could have been discovered through running an anomaly check prior to simulation, since the event `A2.OKAY` has no source. Through examination of the chain, the analyst determines that the problem occurs because `O.REQUEST` must record the source of a request so that the appropriate `OKAY` can be triggered.

Aladdin takes advantage of the behavior section of Rapide interface definitions. Aladdin applies a summarization algorithm to the behavioral description in order to identify the transitional connections in the Rapide description. Aladdin can also be used in conjunction with Rapide's simulation tools. If a specification error is detected during a simulation, Aladdin can be used to identify a reduced set of description elements.

As discussed in Section 3.1, Acme supports specification of language-specific properties when used as an interchange language. The Acme version of Aladdin is based on the analyst's use of Acme properties to specify the links within and among the components and connectors of a system. In this case, Aladdin takes the Acme specification as input and identifies the links automatically.

Aladdin can also be used independently of any particular ADL. The analyst can manually define links by using, for example, an informal graphical notation. When all the connections have been identified, the analyst can make queries about the relationship of specific ports to other ports in the architecture, as described above. In this way it supports Jackson and Wing's notion of "lightweight formal methods" [24] in a manner similar to Feather's use of a database [9].

## 4.4 Extra-Functional Property Analysis

Analysis depends upon description. Our techniques for describing extra-functional properties are significantly less advanced than our techniques for describing functionality. Therefore, the analysis techniques that are available to formally analyze the extra-functional properties of software architectures are not as well developed as those described above. Nevertheless, they show promise and are likely to be the focus of the greatest attention in the near future.

Assuming one can reasonably describe the extra-functional properties of individual components, the key issue in performing an extra-functional analysis is to understand how to compose the properties based on the interactions evident in the architecture. This is illustrated in the example below.

**MetaH**

An interesting form of analysis is provided with the MetaH description language developed by Honeywell [61]. MetaH descriptions capture information about both software and hardware components through an attribution mechanism. Analysis tools are provided for determining which components can affect or be affected by other components in the system depending on the values of the attributes.

- *Safety/security level analysis.* Safety/security attributes are assigned based on the criticality of a component. A safety violation occurs if a defect in a component can potentially affect a component having a higher desired safety level. A security violation occurs if a component could potentially receive information from a more secure component.
- *Schedulability analysis.* Time attributes can be associated with components. These are combined to compute predicted execution time for execution paths through the system. The analyst is given a choice between computing the average path time span and the maximum path time span, the latter being of particular interest for determining guaranteed schedulability.
- *Reliability analysis.* Error attributes, a model of the types of errors that might occur in a system, and the system's expected reaction to those errors are combined to provide a form of reliability analysis. The probabilities of error occurrences are modeled as Markov chains that can be analyzed by any of several common tools.

While MetaH analyses are exhaustive, the are generally less expensive to perform than the types of proof techniques discussed in Section 4.1. In particular, MetaH analyses perform a transitive closure over the static structural relationships defined in the architecture and then examine the attributes of only those components that are shown to possibly be affected.

## 5 Other Architectural Concepts

This paper has concentrated on the role of software architecture in the design process, and the characteristics of various languages and analysis techniques intended to support the software architect. Aside from linguistic and analytical capabilities, several other concepts have emerged to increase the utility of software architecture. In this section we present brief reviews of four of these: architectural styles, domain-specific software architectures, system generation and refinement, and architectural views.

### 5.1 Architectural Styles

Architectural styles provide a standardized vocabulary of high-level structures for refined communication among stakeholders. In the domain of civil architecture, style names are evocative of the building being discussed. Analogously, software architectural styles provide a succinct description of the kinds of components in a system and the constraints on the ways that the components can interact [15, 34, 41, 48].

An architectural style defines a family of systems in which each member of the family shares certain properties with all other family members. Among these properties are its allowed structural elements, constraints on the interactions of those elements, invariants, underlying computational model, and shared experience with systems built in the style. Styles can be arbitrarily specialized

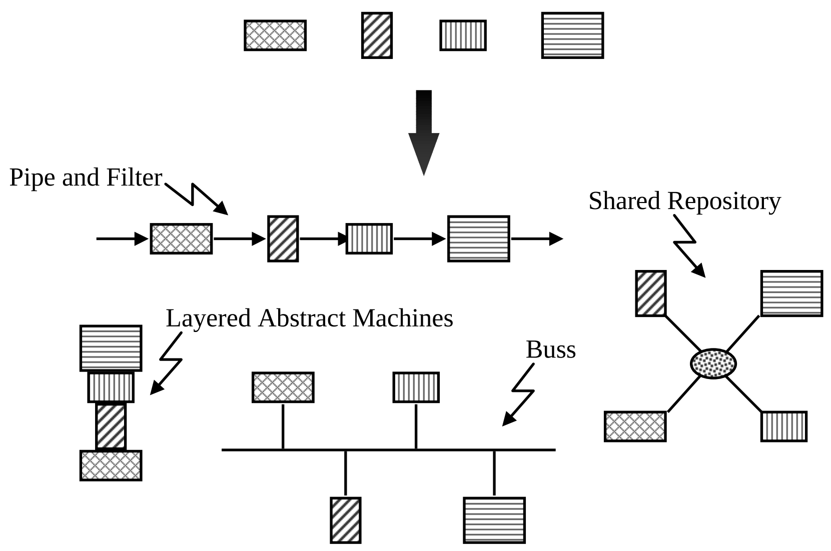


Figure 31: Common Architectural Styles used in Computer Systems Software.

by (further) restricting properties such as acceptable topologies and allowable methods of communication. For example, Le Métayer [34] uses graph grammars to describe constraints on styles.

The choice of a particular style has implications for both the engineering techniques used and the resources needed to build the system. Thus, an early, high-level discussion of style can provide a forum for decisions as to what general form is most appropriate for the system under consideration. In general, different application domains, such as telecommunications, transaction processing, or avionics, develop architectural styles appropriate to that domain. Figure 31 depicts several common styles found in the domain of computer systems software (e.g., compilers, operating systems, database management systems, and the like). Their basic structures can be described quite succinctly.

- *Pipe and Filter*—input to one component is processed and then the resulting output is passed to another component for processing.
- *Shared Repository*—a repository serves as a store house for data that may be accessed by variable numbers of other components.
- *Layered Abstract Machine*—components are stratified into layers, where data produced at one layer is available to layers above but not below.
- *Buss—data are broadcast over a shared communication medium from which components can choose to withdraw the data or ignore the data.*

## Style Classification

Classification techniques, such as taxonomies, are useful for improving understanding and communication in scientific disciplines. In the case of architectural styles, they provide a uniform descriptive standard for styles, a repository for information about styles, a basis for discrimination among styles, and ultimately assistance in the choice of styles. One effort directed at simplifying the choice of architectural style is a classification proposed by Shaw and Clements [48] in which component types, connector types, data communication, control and data interactions, and compatible reasoning types are used as the bases for discrimination among styles. It is their hope that the existence of a classification will aid in the creation of analysis tools to augment a style-based design environment.

#### Style-based Design Environment

Architectural style is being used to simplify the architecting process. Further simplification can be achieved through automation. An example is Aesop, a system for generating design environments for families of software systems [12]. Given an architectural style specification containing information such as style elements and style rules, a design environment is generated. The generated environment is specialized to support the design of architectures in the given style. The environment consists of a graphical user interface in which a system can be constructed from predefined style-specific components and connectors. Aesop can be thought of as an environment for creating new architectural styles.

#### Style-based Analysis

As described in Section 4, many analysis techniques for use at the architectural level have been suggested. These techniques are intended for use on specific architectural instances. Nitpick [23] is an example of an automated analysis technique for use on architectural styles. The technique attempts to prove the absence of specific system properties for all possible instances of a style. If the property, such as deadlock, is found to be possible in any instance, Nitpick produces an example instance-architecture that contains the violation. The style designer can use the example to discover the source of the violation. Information about the probability of the existence of certain system properties can be used when making style choices.

### 5.2 Domain-Specific Software Architectures

Application domains tend to exhibit software-related, domain-specific characteristics. Although the existence of domain characteristics is easy to explain and well accepted, the means for capturing a precise understanding of the specific characteristics for a given domain is new. Domain-Specific Software Architectures (DSSAs) are intended, at least in part, to support this process. They provide a forum for the modeling and definition of domain-specific characteristics that are used to provide a reference architecture from which specific applications within that domain can be created.

An early approach was the Two Life-Cycle Model for system development defined by the Software for Adaptable, Reliable Systems (STARS) program of the U.S. Department of Defense [8]. The first life cycle is for domain engineering and the second is for application engineering. The domain-specific architectural aspects are captured, through domain engineering, in a reference architecture. Application-specific aspects are supported through the use of parameterization or other specialization methods and the instantiation process is outlined for particular applications.

### 5.3 System Generation and Refinement

System generation has been suggested for further raising the level of abstraction for programming computers. Possibly the most fully developed example is the collection of system generation tools supporting the development of compilers from high-level specifications of language syntax, language semantics, and hardware architecture. Recently, system generation has been applied to software architecture descriptions as a means to create rapid prototypes and implementations of system families whose architectures are well understood. Not only does system generation improve productivity when building systems, it also supports the displacement of engineering effort from the details of implementation to higher-payoff tasks such as component design and assignment of inter-component communication mechanisms.

The success of domain-specific system generators inspired the creation of GenVoca [4], a domain independent generator for the creation of domain-specific system generators. The primary goals for GenVoca are to allow automatic customization of components in order to improve productivity in system building, and to produce systems that perform at least as well as hand-built and optimized versions. Systems are composed by combining subsystems that are built out of parameterized reusable component libraries. GenVoca is based on the realizations that complex systems require reuse at a coarser granularity than traditionally available in function-level libraries, that reuse of components depends on standardization of component interfaces, and that parameterization can be used to automatically customize system components.

Architecture refinement, as described by Moriconi et al. [35], is similar in some respects to system generation, but does not depend on pre-implemented components. Rather it is based on the "faithful interpretation" of components as the architecture is gradually refined into an implementation. Faithful interpretation means that all facts defined in an architectural description exist in all other levels of a system's description and that implied negative facts do not. Thus, when useful, the highest-level system abstraction can be relied upon during system analysis at all other levels. Faithful interpretation is achieved through the application of logic-based correct refinement patterns to local refinements that are then combined into the next, lower level of system abstraction. Each level of refinement is guaranteed to be correct based on the guaranteed correctness of the local refinements.

## 5.4 Architectural Views

No matter what form of description is used to describe a software system's architecture, no one description can or should contain all the information that is important to all stakeholders in the development process. An individual stakeholder should have access to a description that contains only the information necessary for understanding aspects of the system relevant to their work. Perry and Wolf [41] introduced the notion of architectural views for this purpose. Architectural views can be thought of in the same sense as views are used in civil architecture, where a building is described in several ways: scale-model view, floor-plan view, builder's detailed view, and the like. The scale-model view provides the general sense of what the structure will appear like in its environment. This view is of interest to landscapers and zoning boards. The builder's detailed view, in contrast, provides information of interest to electricians and plumbers.

In software architecture, the need for different views results from the varied needs of different stakeholders, as well as the varied types of analyses one might want to perform as development progresses. Perry and Wolf associate the implementation view with the builder's detailed view. Other views suggested by Kruchten [29] include logical, process, deployment, and use-case views. The first three views are architectural views and the use-case view is for validation. While a system is being designed, it is good to consider how it can be represented for each of these views. Kruchten recognizes that specific systems require different numbers and kinds of views. Simple systems may not require all of those described in his model while others may require more [30]. He suggests additional views, such as safety views and security views, that would be important to specific classes of systems such as nuclear reactor controllers. Soni et al. [52] discuss categories of architecture: conceptual, module, execution, and code. These categories were defined after studying the structure of a variety of industrial software systems. They feel that viewing software structure from different perspectives is crucial as it will provide a base for formal reasoning about software development.

## 6 Summary

In summary, software architecting is an important phase of the software development process. Equally important are the associated activities of architectural description and analysis. Software architecture can be thought of in analogy to building architecture. In both cases the architecture serves a variety of purposes over the life of the system or the building. The initial blueprint or architectural description serves as an unambiguous base for communication among the stakeholders of the project. The blueprint is available very early in the development process and as such can be used to reason about the fitness of the structure to fulfill its purpose before effort is expended on creating the actual structure. As the system ages, the architecture can be used as a reference to detect undocumented changes to the system or as an aid to support implementation-based maintenance activities such as regression testing and impact analysis. Recognizing the important role of software architecture in software development has led many computer scientists to focus their research efforts on developing languages tailored to software architecture description and on developing associated analysis techniques. The formalization of architectural description and analysis techniques provides a tractable means of reasoning about the construction and maintenance of large and complex software systems.

## References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, May 1994.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] R. Allen, D. Garlan, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, pages 70–79. ACM SIGSOFT, November 1998.

[4] D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, and J. Thomas. The Genvoca Model of Software System Generation. *IEEE Software*, 11(5):89–94, September 1994.

[5] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[6] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An Approach to Integration Testing Based on Architectural Descriptions. In *Proceedings of the 1997 International Conference on Engineering of Complex Computer Systems*, pages 77–84. IEEE Computer Society, September 1997.

[7] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming*, 33(2):101–131, February 1999.

[8] L. Druffel and W.E. Riddle. The STARS Program: Overview and Rationale. *Computer*, 16(11):21–29, November 1983.

[9] M.S. Feather. Rapid Application of Lightweight Formal Methods for Consistency Analyses. *IEEE Transactions on Software Engineering*, 24(11):949–959, November 1998.

[10] Formal Systems (Europe), Ltd. *Failures Divergence Refinement: FDR2 User Manual.* Formal Systems (Europe), Ltd., Oxford, England, October 1997.

[11] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[12] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–188. ACM SIGSOFT, December 1994.

[13] D. Garlan, D. Kindred, and J.M. Wing. Interoperability: Sample Problems and Solutions. Available from the authors, 1995.

[14] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*, pages 169–183. IBM Center for Advanced Studies, November 1997.

[15] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific, New Jersey, 1993.

[16] D. Garlan and Z. Wang. Acme-Based Software Architecture Interchange. In *Proceedings of the Third International Conference on Coordination Models and Languages*, number 1594 in Lecture Notes in Computer Science, pages 340–354. Springer-Verlag, April 1999.

[17] D. Giannakopoulou. The TRACTA Approach for Behaviour Analysis of Concurrent Systems. Technical Report DoC 95/16, Department of Computing, Imperial College of Science, Technology and Medicine, September 1995.

[18] R.M. Gonzales and A.L. Wolf. A Facilitator Method for Upstream Design Activities with Diverse Stakeholders. In *Proceedings of the 1996 International Conference on Requirements Engineering*, pages 190–197. IEEE Computer Society, April 1996.

[19] D.E. Harms. *The Influence of Software Reuse on Programming Language Design.* PhD thesis, Ohio State University, Columbus, May 1990.

[20] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[21] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[22] P. Inverardi, A.L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages*, number 1282 in Lecture Notes in Computer Science, pages 46–63. Springer-Verlag, September 1997.

[23] D. Jackson and C. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[24] D. Jackson and J.M. Wing. Lightweight Formal Methods. *Computer*, 29(4):21–22, April 1996.

[25] F. Jahanian and A. Mok. Modechart: A Specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[26] T.M. Khoshgoftaar, E.B. Allen, R. Halstead, G.P. Trio, and R.M. Flass. Using Process History to Predict Software Quality. *Computer*, 31(4):66–72, April 1998.

[27] J. Kramer and J. Magee. Exposing the Skeleton in the Coordination Closet. In *Proceedings of the Second International Conference on Coordination Models and Languages*, number 1282 in Lecture Notes in Computer Science, pages 18–31. Springer-Verlag, September 1997.

[28] J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A Case Study. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 91–100. IEEE Computer Society, May 1998.

[29] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[30] P. Kruchten. Software Architectur—A Rational Metamodel. In *Proceedings of the Second International Software Architecture Workshop*, pages 5–7, October 1996.

[31] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153. Springer-Verlag, September 1995.

[32] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the Behaviour of Distributed Software Architectures: A Case Study. In *Fifth IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 240–247, October 1997.

[33] N. Medvidovic. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, number 1301 in Lecture Notes in Computer Science, pages 60–76. Springer-Verlag, 1997.

[34] D. Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, July 1998.

[35] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[36] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, October 1995.

[37] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying Static Analysis to Software Architectures. In *Proceedings of the Sixth European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, number 1301 in Lecture Notes in Computer Science, pages 77–93. Springer-Verlag, 1997.

[38] K. Ng, J. Kramer, and J. Magee. A CASE Tool for Software Architecture Design. *Journal of Automated Software Engineering*, pages 261–284, 1996.

[39] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 177–186. Association for Computer Machinery, April 1998.

[40] O. Oskarsson and R.L. Glass. *An ISO 9000 Approach to Building Quality Software.* Prentise Hall PTR, Prentice Hall, Inc., Upper Saddle River, New Jersey, 1996.

[41] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[42] A. Podgurski and L.A. Clarke. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[43] R.S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, 4 edition, 1997.

[44] D.J. Richardson and A.L. Wolf. Software Testing at the Architectural Level. In *Proceedings of the Second International Software Architecture Workshop*, pages 68–71, October 1996.

[45] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1998.

[46] A. Rosetti. Generazione di Test Cases da Specifiche Formali della Architettura Software. Technical Report Tesi di Laurea, Dipartimento di Matematica Pura ed Applicata, L'Aquila, Italy, March 1997.

[47] R.W. Schwanke, V.A. Strack, and T. Werthmann-Auzinger. Industrial Software Architecture with Gestalt. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 176–180. IEEE Computer Society, March 1996.

[48] M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 1997 International Computer Software and Applications Conference*, pages 6–13, August 1997.

[49] 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*, 21(4):314–335, April 1995.

[50] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. *Computer Science Today: Recent Trends and Developments*, pages 307–323, 1997.

[51] United States Naval Postgraduate School Software Engineering. CAPS User's Manuals. 1997.

[52] D. Soni, R.L. Nord, and C. Hofmeister. Software Architecture In Industrial Applications. In *Proceedings of the 17th International Conference on Software Engineering*, pages 196–207. Association for Computer Machinery, April 1995.

[53] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Chaining: A Software Architecture Dependence Analysis Technique. Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado, Boulder, Colorado, September 1997.

[54] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems. Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[55] RAPIDE Design Team. Draft: Guide to the Rapide 1.0 Language Reference Manuals. July 1997.

[56] RAPIDE Design Team. Draft: Rapide 1.0 Architecture Language Reference Manual. July 1997.

[57] RAPIDE Design Team. Draft: Rapide 1.0 Pattern Language Reference Manual. July 1997.

[58] R. Terry and M. Devito. *The ArTek Architecture Description Language, version 4.* ARDEC/Teknowledge, July 1995.

[59] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Versioned Software Architecture. In *Proceedings of the Third International Software Architecture Workshop*, pages 73–76, November 1998.

[60] I. Vessey and R. Glass. STRONG vs. Weak Approaches to Systems Development. *Communications of the ACM*, 41(4):99–102, April 1998.

[61] S. Vestal. *MetaH Programmer's Manual Version 1.27.* Honeywell, Inc., Minneapolis, MN, 1998.

[62] L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors. *Joint Proceedings of the SIGSOFT '96 Workshops.* ACM Press, New York, New York, 1996.

[63] B.W. Weide, W.D. Heym, and J.E. Hollingsworth. Reverse Engineering of Legacy Code is Intractable. OSU-CISRC-10/94-TR55 October, Ohio State University, 1994.

[64] W.-J. Yeh and M. Young. Compositional Reachability Analysis using Process Algebra. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 49–59. ACM SIGSOFT, October 1991.

[65] M. Young and R.N. Taylor. Rethinking the Taxonomy of Fault Detection Techniques. SERC TR-62-P September, Department of Computer Science, Purdue University, 1991.

[66] G. Zelesnik. UniCon Reference Manual. Technical Report CMU-CS-97-TBD, Carnagie Mellon Univeristy, 1997.

[67] J. Zhao. Using Dependence Analysis to Support Software Architecture Understanding. *New Technologies on Computer Software*, pages 135–142, September 1997.