Analysis of a Software Product Line Architecture: An Experience Report

Running Title: Analysis of a Product Line Architecture

Robyn R. Lutz [1]
Jet Propulsion Laboratory
California Institute of Techology and
Dept. of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, IA 50011-1041
(515) 294-3654
rlutz@cs.iastate.edu

Gerald C. Gannod [2] [3]
Dept. of Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
(480) 727-4475
gannod@asu.edu

[1]Contact Author.
[2]This research was performed while this author was a visiting researcher at the Jet Propulsion Laboratory.
[3]This author supported in part by NSF CAREER Grant CCR-0133956.

Abstract

This paper describes experiences with the architectural specification and tool-assisted architectural analysis of a mission-critical, high-performance software product line. The approach used defines a "good" product line architecture in terms of those quality attributes required by the particular product line under development. Architectures are analyzed against several criteria by both manual and tool-supported methods. The approach described in this paper provides a structured analysis of an existing product line architecture using (1) architecture recovery and specification, (2) architecture evaluation, and (3) model checking of behavior to determine the level of robustness and fault-tolerance at the architectural level that are required for all systems in the product line. Results of an application to a software product line of spaceborne telescopes are used to explain the approach and describe lessons learned.

Biographies

Robyn R. Lutz is a Senior Engineer at Jet Propulsion Laboratory, California Institute of Technology, and an Associate Professor in the Department of Computer Science at Iowa State University, Ames, Iowa. Dr. Lutz has worked on spacecraft projects in fault protection, real-time commanding, and software requirements and design verification. Her research interests include software safety, safe reuse of product families, formal methods for requirements analysis, intrusion detection, and fault monitoring and recovery strategies for spacecraft.

Gerald C. Gannod is an assistant professor in the Department of Computer Science and Engineering at Arizona State University. He received the MS ('94) and PhD ('98) degrees in Computer Science from Michigan State University. His research interests include software product lines, software reverse engineering, formal methods for software development, software architecture, and software for embedded systems. Dr. Gannod is a recipient of a 2002 NSF CAREER grant.

1 Introduction

The analysis of a software product line architecture investigates the extent to which a proposed architecture supports both the shared requirements for all systems in the product line and the distinct requirements pertaining to individual systems in the product line. When the product line implements mission-critical or high-performance requirements, the analysis of the software product line architecture must account for these additional constraints in the architectural design. This paper describes experiences with the architectural specification and tool-assisted architectural analysis of one such mission-critical, high-performance software product line. Topics include the Architecture Description Language representation of the product line, architecture evaluation techniques, model checking of key behaviors, and lessons learned from the application.

The paper is divided into six sections. Section 1 provides an overview of the process used in analyzing the architecture. Section 2 presents the background, namely an overview of the application domain (a product line of spaceborne telescopes) and a discussion of related work. Sections 3, 4, and 5 describe the three main steps in the architectural analysis process: Architecture Recovery and Specification (Section 3), Architecture Evaluation (Section 4), and Tool-Assisted Architecture Analysis (Section 5). Each of these three sections is subdivided into (1) a process description for that step, (2) an extended example of that step drawn from the application domain, and (3) a discussion of the general lessons learned, intended for developers of other critical, high-performance product lines. Section 6, the concluding section, summarizes the take-away results for the architectural analysis of high-performance product lines.

The following paragraphs provide a brief overview of the architectural analysis process. Figure 1 depicts the process graphically. Since this application built on an existing product, rather than initiating a new product line, the analysis began with an effort to recover, i.e., to understand and specify in a useful way, the existing software architecture. This effort demonstrated the analytical value of specifying an existing architecture with an Architecture Description Language (ADL) both in terms of identifying mismatches between the supposed and the actual architecture, and in terms of providing a baseline for subsequent automated analyses. The ADL model and its usefulness as a baseline are described in Section 3.

Once an accurate ADL model of the software product line architecture was established, it was used to evaluate the adequacy and level of support provided by the architecture for the slightly different requirements of the individual systems in the product line. In particular, a set of scenarios

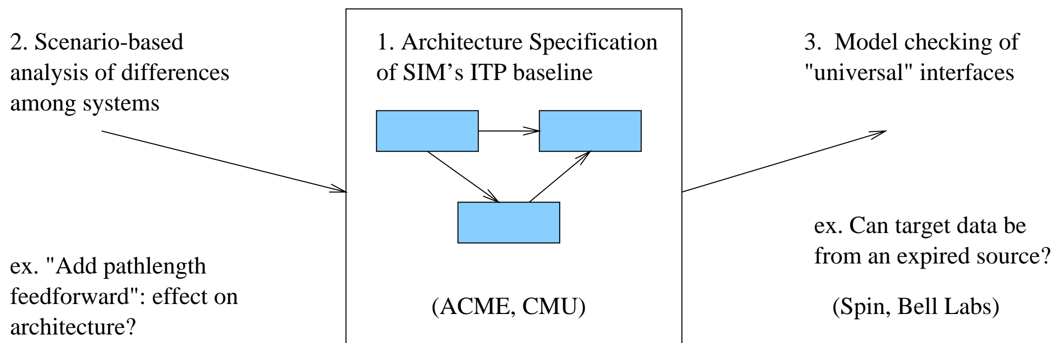

Figure 1: Architectural Analysis Process

representing typical or anticipated changes from the requirements of existing systems was developed to exercise the architecture's modifiability. Section 4 details this step and describes the special problems that requirements for high performance place on the architectural evaluation of a software product line.

The behavior of some key interfaces common to all the systems in the product line were further verified using tool-supported analysis. By extending the ADL model, formal verification using model checking of these interfaces was practical and cost-effective. Of particular concern in the analysis was the fault-tolerance of a critical data interface shared by the entire product line. Tool-supported model checking allowed a demonstration of some undesirable consequences of an architectural decision. Section 5 describes the process by which the ADL model was translated into a format readable by the model checker, the formal verification performed on the critical interface, and the benefits of this verification approach.

2 Background

This section discusses background material in the areas of space interferometry and software architecture analysis.

2.1 Interferometers

The product line described in this paper is a set of interferometer systems under development by the Jet Propulsion Laboratory. An interferometer, in this context, is a collection of telescopes that

act together as a single, very powerful instrument. An interferometer combines the starlight it collects from telescopes in such a way that the light "interferes" or interacts to increase the light's intensity and the precision of the observation. Over the next several decades these interferometers will be used to explore the origins of stars and galaxies and to search for Earth-like planets around distant stars.

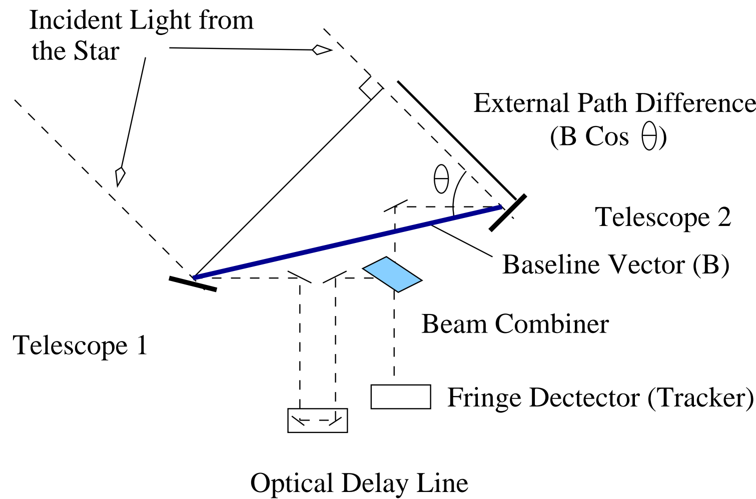

Figure 2: Interferometer

In particular, three spaceborne interferometers are either under development or planned for launch in the next eleven years, with additional formation-flying interferometers envisioned for subsequent years (Danner and Unwin 1999, Origins Science Committee 2000). Two ground-based, prototype interferometers in the product line are currently operational, with at least two more planned.

The software in these interferometers has a high degree of commonality with a managed set of shared features built from core software components (Gannod and Lutz 2000, Lutz 2000, Gannod et al. 2001). A group of developers at JPL with a strong background in interferometer software provides reusable, generic software components to the interferometer projects.

2.2 Related Work

The work in this paper builds on product family techniques such as Commonality Analysis (Ardis and Weiss 1997, Lutz 2000) and the FAST process (Weiss and Lai 1999), which systematically model

the required similarities and differences among family members. The architectural implications of product line models have been analyzed by Perry (Perry 1998), by Gomaa and Farrukh (Gomaa and Farrukh 1999), by Bosch (Bosch 2000), and by researchers at SEI, among others (Clements and Northrup 2002, Schwanke and Lutz 2001). To date, the emphasis has been on developing architectures for new product lines rather than on evaluating the architecture of an existing product line, as is done here.

The Software Architecture Analysis Method (SAAM) is a scenario-based method for architectural assessment. A scenario, in this context, is a description of an expected use of a specific product line. SAAM also tests modifiability, e.g., by proposing specific changes to be made to the system. A related architectural analysis method is the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al. 1998). This iterative method is based on identifying a set of quality attributes and associated analysis techniques that measure an architecture along the dimensions of the attributes. Sensitive points in an architecture are determined by assessing the degree to which an attribute analysis varies with variations in the architecture. In our approach, we focus on quality attributes that are specific to product line architectures. As such, the approach can be applied in either the SAAM or the ATAM context.

ACME is an architecture description language for high-level architectural specification and interchange (Garlan et al. 1997). For the work described in this paper, ACME was selected as a specification language in part because it contains constructs for embedding specifications written in a wide variety of existing ADLs, making it extensible to both existing and future specification languages. Another advantage of ACME is that it is supported by an architectural specification tool, ACMEStudio, for graphical construction and manipulation of software architectures. ACME provided an easy-to-read, easy-to-update, graphical view of the architecture that facilitated review by the engineers. The product line architecture described in this chapter was specified using the ACME ADL (Garlan et al. 1997) and ACMEStudiosupport tool.

Rapide (Luckham and Vera 1995) is a suite of techniques and tools that support the use of *executable architectural design languages* (EADLs). The toolset supports analysis of time-sensitive systems from the early construction phase (e.g., architecture definition) to analysis of correctness and performance. Wright (Allen and Garlan 1997) is an ADL based on the CSP specification language (Hoare 1985). The primary focus of the Wright ADL is to facilitate the specification of connector, role, and port semantics. In addition to being based on the well-established CSP

semantics and allowing partial specification, existing Wright tools support the ACME ADL, thus providing a clean interface with the existing ACME specification.

In this work, the motivation for choosing a particular technique was based on a desire to eventually transfer the technology to the project engineers. In addition, there was an interest in achieving interoperability with other tools. As such, it was found that the ACME ADL and associated ACMEStudio tool presented the least amount of educational overhead. ACME also had the advantage of being able to embed other ADLs, including Wright, in its specification. However, it was recognized that several alternatives such as Rapide exist and are investigating the possibility of performing similar analyses with those tools.

3 Architecture Recovery and Specification

3.1 Process

The goals of architecture recovery and specification are to familiarize the analysts with the problem domain and implemented solution, and to support construction of a software architectural representation that is, above all, correct and consistent. In the experience described here, the main inputs to the process were project documentation, source code, and communication with developers.

Recovery of the software architecture involved a fairly intensive study of the product line requirements and the available systems (testbeds and prototypes). Extensive documentation of the requirements and design for the reusable software components, as well as C code for the prototypes of the components, were available. In addition, several hundred pages of project-specific documentation, maintained in a web-based library, were used. Predictably, more documentation existed for projects farther along in their development. System descriptions were available for all the interferometers in the product line; software requirements and design documents were still high-level and informal for later missions; and code was not yet available for any of the spaceborne interferometers.

A draft architecture was recovered from the available information and compared with an existing description, somewhat out-of-date, of the architecture then planned. As shown in Figure 2, the original documentation for the interferometry software depicts the software using a layered style. However, during the analysis and subsequent specification of the architecture, it was discovered that the documented architecture exhibited "layer bridging" properties, whereby non-adjacent layers in the architecture communicated, thus "bridging" or by-passing intermediate layers. In addition,

sibling components located in a layer were found to communicate, contrary to the layered style.

Gizmo Prototypes				
Instrument CDS	Delay Line	Fringe Tracker	Wide Angle Pointer	Star Tracker
Gizmo Design Pattern				
Gizmo Framework	Servo & Controller Framework	I/O & Modulation Framework	Command Engine Framework	Command & Telemetry Framework
Core Services				
Inter–processor Communication	Periodic Task Scheduler	Hardware Framework		
Configuration				

Figure 3: Original Core Architecture

Differences between the actual architecture and the architecture as previously documented were investigated and resolved. The high-level interferometer architecture was re-specified in a style that was consistent with the services and behaviors described in lower-level documentation and currently planned for implementation. The resulting architecture more accurately specified the architectural style as heterogeneous with a collection of communicating processes as well as a constrained pipe and filter (e.g., data flow) interaction between the Instrument CDS (Command and Data Subsystem) and all the remaining components.

Project engineers were consulted to validate the accuracy of the newly drafted architectural model against the actual architecture. They also reviewed the accuracy of the analysts' understanding of architectural components of future systems in the product line. The expert feedback was instrumental in constructing a more accurate representation of the interferometer architecture.

Components	Core	D1	D2	D3
Baselines	1	1	3-4	1
Arms	2	2	6-8	2
Wide Angle Pointer	2	2	6-8	2
Star Tracker	2	2	6-8	2
Delay Line	2	2	6-8	4
Fringe Tracker	2	1	3-4	2
Instrument CDS	1	1	1	1
User Interface	1	1	1	1

Table 1: Comparison Matrix

3.2 Example

A diagram depicting the interferometer product line architecture is shown in Figure 4. The ACME representation of this architecture formed the basis for subsequent analyses, both manual and automated. In the diagram, hardware components are shown as rounded rectangles while the software components are shown as sharp rectangles. The connectors, represented by lines between components, depict the relationships between components in the architecture. This particular diagram represents the software that exists within an "arm" of an interferometer, where a standard interferometer has two arms.

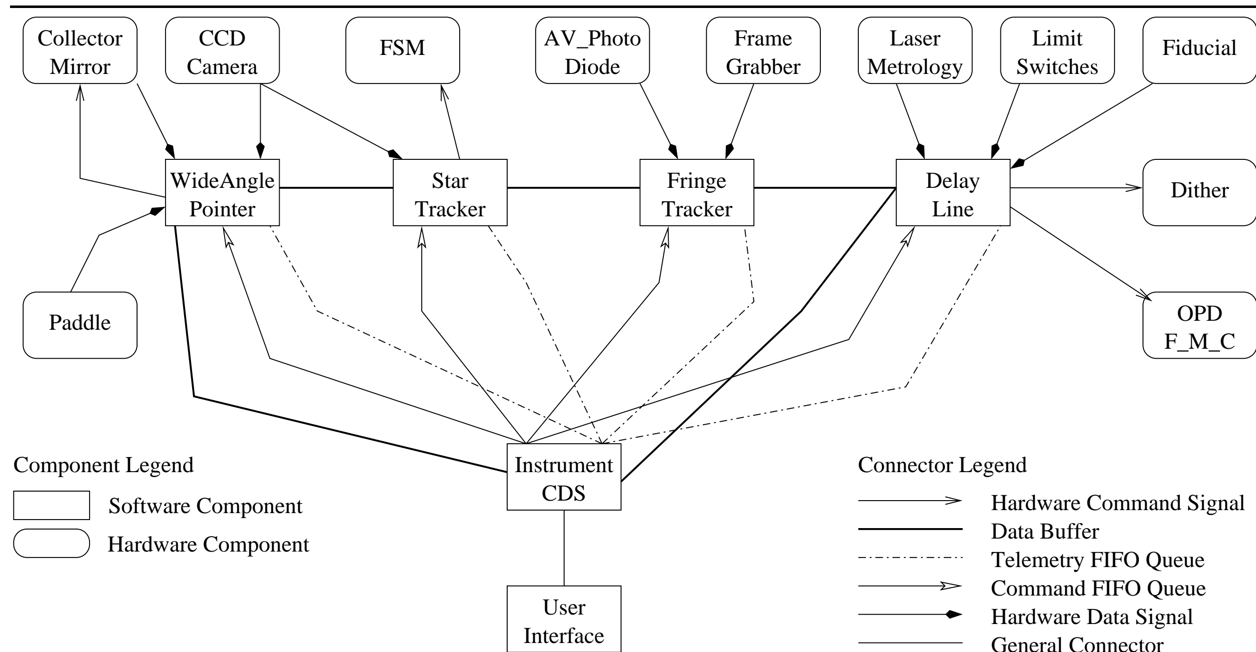

Figure 4: Interferometer Software Architecture

The architecture of the interferometer software uses a heterogeneous architectural style that includes the use of both the independent component style and a sequential data flow style. Each of the connectors from the Instrument CDS component to the Wide Angle Pointer, Star Tracker, Fringer Tracker, and Delay Line components are either input or output queues. The semantics between the components from the Instrument CDS perspective adheres to an independent components style. The interaction between the Wide Angle Pointer and the Star Tracker, the Star Tracker and the Fringe Tracker, and the Fringe Tracker and the Delay Line is in the form of a data flow style, where transformations of data occur in sequence starting from the Wide Angle Pointer and ending at the Delay Line.

Among the components shared by the interferometer systems and discussed in this paper are the Delay Line, the Fringe Tracker, and the Internal Metrology. Briefly, the Delay Line component compensates for the difference in time between the arrival of starlight at the separate mirrors. The Fringe Tracker component provides constant feedback to the Delay Line regarding needed adjustments to maintain peak intensity of the fringe (patterns of light and dark bands produced by interference of the light). The Internal Metrology component provides input to the Delay Line regarding small changes in distances among parts of the interferometer that must be included in its calculations.

With regard to the interferometer product line, the most difficult part of architectural recovery and specification was understanding the required variations among the systems in the product line, especially the variation in the number of copies of core components in the specific systems. It was found that the interferometers have a single baseline (labeled the "core" in Table 1) as their basic building block. The baseline has as components two arms, two delay lines, two fringe trackers, etc. Most interferometers have a single baseline. Those systems with more than one baseline have multiple copies of the baseline building block and, thus, a number of each component that is a multiple of two (e.g., a two-baseline system with four delay lines, four fringe trackers, etc.). There are a few deviations from this architecture (one interferometer operates with a variable number of baselines; an interferometer testbed may use a single component where two would be used on a flight interferometer), but in general, the core architecture, shown in Fig. 4, captures the baseline.

To further validate the accuracy of the architecture in terms of its scalability to future, planned products in the product line, the architecture was checked against the individual product line derivatives. A simple table was constructed and maintained to describe the components of future systems in the product line in a format that the developers could quickly review for accuracy.

In the table (excerpted in Table 1), each row represents a different component that could be potentially present in an interferometer system. The columns represent the different derivatives that are currently either being developed or are planned for deployment over the next several years. This table represents features of the architecture that are *common* in behavior across all systems in the product line, but that can *vary* in multiplicity based on the number of potential starlight collectors or "arms".

3.3 Lessons Learned for Product Lines

The architectural recovery and specification of the interferometer product line provide some insight into issues involved in modeling existing product lines: achieving consistency between the specified and the actual architecture, using abstraction to identify a core architectural building block, and facilitating review by domain experts via readable ADL specifications.

- *Resolving discrepancies between actual and documented architectures.* Some product lines, such as the interferometer, are not originally designed as product lines but evolve into them as new products are created. When a product line approach is applied to a set of systems in which some have already been built, the actual architecture may give evidence of evolution away from the earlier planned architectures. In this case, the documented architecture may not match the architecture that has been, and will be, used to build systems in the product line. Architectural recovery can contribute to the re-specification of the architecture that will serve as a baseline for future implementations.

- *Abstraction.* It proved quite useful in the architectural specification phase to abstract out a core architectural building block (the baseline shown in Fig. 4) that was reused intact throughout the product line. Some systems in the product line (e.g., testbeds) use a single copy of this building block; others have multiple copies that operate either serially or in parallel. The use of parallel building blocks and of multiple copies of the building block to achieve performance requirements (here, high accuracy and image resolution) appears to be typical of high-performance product lines. Additional work is needed to investigate how abstraction in architectural specification can best handle such critical performance requirements.

- *Advantages of ADL specifications.* Use of the architectural interchange language, ACME, to specify the architecture encouraged communication and review by experts. The graphical view provided a front-end that represented the abstract architecture clearly and accurately. The readability of the ADL specification made it easier to verify that the architectural style was common to each of the systems in the product line.

4 Architecture Evaluation Using Scenarios

4.1 Process

The next phase of the process was to perform a number of analyses in order to help determine to what extent the architecture was amenable to a product line development approach. The primary goal was to determine if certain desirable quality attributes present in most product line architectures were also present in the interferometer architecture. The goal of this analysis was to exercise the product-line architecture by considering the effect on it of representative change scenarios. To do this, the effect of the variabilities required by the individual missions on the architecture was evaluated.

To study the modifiability of the interferometry product line architecture, representative examples of required modifications were extracted from the requirements specification of four systems currently planned or under development. The advantage of using these scenarios was that it moves the discussion from a rather amorphous, high level of generality ("modifiability") to a concrete, context-based level of detail particular to the product line ("adds pathlength feedforward capability").

Singling out the modifiability requirements from the documentation for the various systems was straightforward. Required changes to previous systems were usually called out explicitly. Rationales were often described, based on the unique scientific aspects of each system's mission. Analysis of architectural consequences of required changes was more challenging, since this required design knowledge. Occasional access to domain experts was essential, since they could elaborate on the design consequences of the planned evolvability and answer analysts' questions. This scenario-based architectural analysis provided a rapid, low-cost way to evaluate the architectural consequences of some key product-line variabilities for the interferometer applications.

4.2 Example

Of the properties that cannot be observed at runtime, modifiability is the key property required by the interferometer product line. Modifiability, according to Bass, Clements, and Kazman, "may be the quality attribute most closely aligned to the architecture of a system," and, as such, is a good way to evaluate the architecture (Bass et al. 1998). Bass, Clements, and Kazman identify four categories of modifiability: Extensibility (changing capabilities, adding new functionality, repairing bugs); Deleting capabilities (streamlining, perhaps to deliver a less-capable and less-expensive

version of a product); Portability (adapting to new operating environments, such as processor hardware, input/output devices, and logical devices); and Restructuring (modularizing, optimizing, or creating reusable components).

All four of these categories of modifiability are represented by significant requirements within the interferometer product line systems:

Extensibility. Potential extensibility variations include new algorithms (e.g., a different fringe-search algorithm) and added features (e.g., pathlength feedforward, internal metrology).

Deletions. Deletions involve removal of previously required capabilities. Architectural support for deletions in product lines is essential since experience has shown that systems are often scaled down unexpectedly during development to meet resource or schedule constraints (Lutz 2000). In the interferometer application, deletions usually involved testbeds or prototypes that both added capabilities to support analysis of new technologies and also deleted, or stubbed in, some previous capabilities. For example, testbeds use pseudostar (simulated) input rather than actual starlight, whereas the science interferometers use direct starlight as input.

Portability. Portability changes are widespread, since different interferometers in the product line will use different starlight detector hardware and different operator interfaces (e.g., a handheld paddle for the testbeds, remote commandability for the flight units). The interferometer software will run on multiple processors, with the number of processors being a variability among the systems. The software is required to run on these different platforms with only minor modifications.

Restructuring. Restructuring changes that are not included in the other categories are limited. A proposed change to optimize for reuse is the only scenario used in the architectural evaluation. It is worth noting that all the changes are evolutionary rather than revolutionary in nature. Even the most far-reaching change, a proposed move to CORBA, would initially implement only some of the standard's features. While more dramatic change scenarios could be imagined, they were judged unlikely in this set of systems. Whether scenario-based evaluation of modifiability is useful where change scenarios are broader and less concrete is an open question.

Table 2 shows nine of the representative changes selected to evaluate the modifiability of the architecture: three extensibility changes, one deletion, four portability changes, and one restructuring. The scenarios were selected on the basis of coverage of the classes of modifiability, likelihood of architectural impact, and importance to the future system(s) in the product line. The changes were all variabilities in the product line specification, i.e., not common to all the interferometers. The approach was to use these representative scenarios to exercise and evaluate the baseline architecture.

Attribute	*Scenario Type*	*Example Scenario*	*Effect on Architecture*
Extensibility	Change algorithm	Algorithm for fringe search changed	No change required
Extensibility	Add feature	Pathlength feedforward capability	No style change; additional connectors
Extensibility	Add feature	Internal metrology added	No style change; additional components and connectors
Deletion	Delete input	Use pseudostar rather than actual	No change required
Portability	Change HCI device	Shift handheld paddle to remote device	Connector unchanged
Portability	Change sensor	Starlight detector hardware changed	Interface intact; component implementation changes
Portability	Add input units	More starlight collectors	No style change; "duplicate" existing pieces; see discussion
Portability	Add processors	Distribute targeting computation	No style change; change within components
Restructuring	Optimize for reuse	Proposed switch to CORBA	Might change style and connectors

Table 2: Analyzing the Architecture's Modifiability via Scenarios

A summary of the the results of the evaluation of the architecture's modifiability appears in Table 2 . Column 1 indicates to which of the four categories of modifiability each scenario belongs (Extensibility, Deletion, Portability, or Restructuring). Column 2 is a high-level description of the scenario (e.g., "Change algorithm", "Add feature", "Change sensor", etc.). Column 3 briefly describes the particular scenario. Column 4 indicates the effect of that modifiability scenario on the baseline architecture.

Of the nine scenarios in Table 2, four involved no change to the baseline architecture. These scenarios were: change of algorithm, deletion of input, change of human-computer interface device, and change of sensor device. Two other scenarios, related to extensibility, require additional connectors and, in one case, an additional component not in the original architecture. However, these extensions are relatively straight-forward and their scope is easy to anticipate.

The other three scenarios require significant changes to the product line architecture, but the changes are not visible at the level of the specified architecture. In one case (add input units), implementation of the scenario can involve adding "arms" (i.e., additional axes) to the interferometer. This has no effect on the more detailed core architecture (which represents a single axis), but requires duplication/replication of connectors and components on the baseline architecture, a significant architectural consequence. The scenario that distributes the targeting computation over more processors can be accommodated without change to the baseline architecture. At the level of the model, there was no commitment to implementation details such as number of processors. The sole restructuring scenario, a possible switch to CORBA, might change both the style and the

implementation of the connectors, and would require further investigation.

In summary, the following results were noted in the study of the architecture's support for the required modifiabilility scenarios:

- **Locality of change.** Most modifiability scenarios demonstrated good locality of change for the specified architecture (i.e., involved changes that could be readily scoped). The existence of an architectural specification assisted in this effort. Most scenarios do not affect the services required of other components.

- **Units of reuse.** The units of reuse in the architecture tended to be small. For example, a Delay Line is a unit, but a Delay Line-Fringe Tracker-Star Tracker is not. All Delay Lines have a high degree of commonality, and the interfaces between a single Delay Line and a single Fringe Tracker are similar for all members (the "portability layer"), but the number of Delay Line-Fringe Tracker interfaces varies greatly among the product line members. The architectural style was not changed by the scenarios, but the number of connections and, to a lesser degree, components, was changed. There are many different cross-strappings possible and a large amount of reconfiguration involved in meeting the real-time constraints on the various missions. Having small units of reuse may complicate verification and integration of individual members (e.g., with regard to contention, race conditions, starvation, etc.).

- **Role of multiplicity in a high-performance product line.** Several of the scenarios involved adding multiple, identical components or connectors. However, these copies are not redundant, in the sense of adding fault tolerance, since they are all needed to achieve the required performance. For example, if starlight collectors are added, it is to increase the amount of starlight that the interferometer can process in order to meet requirements for detecting dim targets. Likewise, if processors are added, it is to meet requirements for increasing the resolution capability of an interferometer. In this architecture, multiplicity does not add redundancy or robustness for the most part; there are not spare units or alternate data paths.

- **Architectural style.** Despite the range of variations that affect the architecture (e.g., varying the number of ports on a component, varying the number of instances of a component), the interfaces themselves are relatively stable. Recognizing the long timeline over which the product line will extend (proposed launches from 2003 to 2020) and the primacy of performance (with continuous improvement of hardware and algorithms), the project designed

well for evolvability.

4.3 Lessons Learned for Product Lines

A baseline architecture for a product line shows the commonality that exists among the members of that product line. Each member of the product line uses this architecture or an adaptation of it. Thus, nothing in the architecture can constrain the anticipated variabilities among the members. The modifiability analysis of the interferometer product line architecture was based on well-documented techniques that have worked well in a variety of application domains. The process is summarized below, followed by a discussion of its applicability to high-performance product lines and its use in challenging architectural support for planned variations in requirements.

- *Verifying architectural support for modifiability.* The process of verifying that the architectural style accomodates the required modifications for future systems is:

  1. Identify anticipated changes from available documentation and project information. These anticipated changes form product line variabilities that the baseline architecture must accommodate.
  2. Categorize the anticipated changes into modifiability categories (extensibility, deletion, portability, restructuring).
  3. Select and develop scenarios for each category. The choice of scenarios is made to broadly challenge the goodness of the architecture with regard to the four modifiability categories.
  4. Evaluate the effect of each modifiability scenario on the baseline architecture. This gives a measure of the goodness of the architecture with respect to the anticipated variabilities for this product line.

- *Availability of scenarios.* Appropriate scenarios for exercising the architectural support for required system variabilities were readily available in the existing project documentation. Many of the requirements variabilities were explicit ("unlike system x and y, system z shall . . . "). Additional scenarios were provided by use cases, descriptions of operational modes, and requirements for exceptional (off-nominal) cases or fault handling. Scenarios were selected to exercise all four categories of modifiability, with representation of both narrowly scoped

changes (e.g., deleting an input source in a testbed) and broader, more loosely defined changes (e.g., adding complex capabilities such as internal metrology).

- *Performance issues.* Most studies of product lines do not involve high-performance product lines. However, high-performance product lines are being built and put into operation. In high-performance product lines, the range and scope of the variabilities tend to be less negotiable than in other product lines. This is due to the very tight performance and accuracy requirements. For example, an upcoming interferometer, the Space Interferometry Mission (SIM), requires precision at the level of picometer metrology and microarcsecond astrometry (Colavita et al. 1999) (see Table 3). To achieve a high level of precision, significant real-time constraints exist, often with limited flexibility to accommodate reuse concerns. In such product lines, performance requirements on new systems may drive the choice of hardware, algorithms, and added capabilities. The consequence for reuse is that in trade-offs of modifiability vs. performance, performance wins.

Component	Requirement
Baselines	10 m (SIM) - 10 km (ST-3)
Imaging Resolution	10 microarcsecond (SIM)
Path length knowledge	50 picometers (SIM)
Path length control	10 nanometers (SIM)
Formation flying	1 cm precision (ST-3)

Table 3: High-performance system requirements driving software

- *Differences from single-system architectural analysis.* The scenario-based analysis of the product line architecture differs from a scenario-based analysis of a single-system architecture in that the scenarios do not represent use cases of a system but variabilities, i.e., the *differences* among systems. The modifiability scenarios serve a role somewhat like boundary testcases, probing the limits of the architecture. The scenario-based analysis for the product line also differs from a scenario-based analysis of a single system in that the architectural style of the product line, rather than the architecture of a specific system, is challenged. For example, a requirement to add starlight collectors to a system would change that system's architecture but, as seen above, does not change the product line's architectural style.

5 Tool Assisted Architecture Analysis

5.1 Process

The process of automated analysis involved: (1) *Architecture specification in an ADL*, (2) *Formal specification of behavior*, and (3) *Analysis of behavior to determine fault-tolerance and robustness.* The approach used in the selection of notations and tools is described here. One of the goals of this project was to determine the extent to which automated support tools could be used to aid in the analysis of a product line software architecture. Specifically, this involved identifying tools that could be adopted with little overhead, while still satisfying the objective of formally analyzing the architectural behavior to determine fault-tolerance and robustness. This meant that the selected tools should have a reasonable level of support and documentation.

In addition to recovering and specifying the high-level view of the interferometer architecture, behaviors of component interactions were derived from existing design documentation. Specifically, we used information found in design documents to help construct a formal specification of component interactions in the interferometer software.

While ACME provides an infrastructure for high-level architecture specification and ADL interchange, it lacks a significant behavioral specification component. Consequently, another ADL, Wright, was used for the formal specification of behavior. The resulting formal specifications were used to analyze the behavior of various aspects of certain interactions between components in the architecture. To validate the formal analysis, source code from the interferometer components planned for reuse was informally reverse engineered to determine whether properties observed in the formal specification were present in the implementation. The Spin Model Checker was used to further analyze behaviors of interest. Spin (Holzmann 1997) is a model checker that has been used for verifying the behavior of a wide variety of hardware and software applications. Promela, the input specification language for Spin, is based on Dijkstra's guarded command language as well as CSP.

The primary reason for choosing each of the notations and tools listed above was a pragmatic one. The notations are related either via direct tool interchange support (as is the case between ACME and Wright) or by some semantic foundation (e.g., CSP foundation for Wright and Promela). As such, the ACME framework (including Wright specifications) could be used for specifying the interferometer architecture, and verification using Spin could follow naturally with a small amount of translation of the embedded Wright into Promela.

5.2 Example

A key element of the interferometer architecture was the use of the "Target Buffer" connector. This connector, both in the design and in the implementation, is a non-locking buffer used to communicate target trajectories to the Delay Line component by several other components. A target is a specified position for the Delay Line controller to achieve. The target trajectories from multiple sources are combined by the Delay Line's target generation software to calculate a target. The Target Buffer connector was viewed as a possible concern, especially in light of the non-locking feature. Behavior involving this connector was formally specified in order to study its impact on the system.

There are several components that are either directly or indirectly impacted by the non-locking nature of the Target Buffer connector: Target Sources, a Command Controller, and a Target Generator component. The Target Generator uses the values written to the Target Buffer by various Target Sources to compute a target position for the interferometer. The Command Controller provides control for the computation by enabling or disabling the Target Sources. Target Sources write a timestamped value to the Target Buffer, with the timestamp determining a time that the target value becomes valid.

The Target Generator uses the following four-step sequence for calculating the target position:

1. Promote waiting targets to active status if the current time is greater than or equal to the timestamp
2. Read new targets from enabled target sources
3. Pend (assign to *wait status*) or activate new targets based on timestamps
4. Compute the total target

The Wright specification of the interaction between the Target Generator and the potential sources of data that are written to the Target Buffer is shown in Figure 5. The Source specification models the fact that a source internally decides whether or not to write a new value to the Target Buffer. Finally, the Target Generator specification models the target-position algorithm described above.

From the Wright specification, we constructed a Promela specification, portions of which are found in Figures 7 and 8. A sample of a message sequence chart for this model is shown in Figure 6. In the diagram, the vertical bars represent processes, messages received or sent by a process are shown as arrows between the bars, and the relative time between sending and receiving the various

```
Style TargetComputation
Connector TargetBuffer
   Role Writer = writetarget!x -> Writer |~| Tick
   Role Reader = readtarget?x -> Reader |~| Tick
   Glue = Writer.writetarget!x -> Glue []
        Reader.readtarget!x -> Glue [] Tick
Component Source
   Port CDSCommand = enable -> CDSCommand |~|
                   disable -> CDSCommand |~| Tick
   Port DLTarget = write!x -> DLTarget |~| Tick
   Computation = (CDSCommand.enable -> Generate) []
      (CDSCommand.disable -> Computation) [] Tick
   where { Generate = DLTarget.write!y -> Generate []
           Generate [] Tick }
Component TargetGenerator
   Port Input = readtarget?x -> TargetBuffer |~| Tick
   Computation = (_promote ->
               Input.read_target?x ->
               _pend_or_activate ->
               _compute -> Computation [] Tick )
end Style

Configuration TargetComputationInstance
Instances
   tb1 : TargetBuffer
   src1 : Source
   dl : TargetGenerator
Attachments
   src1.DLTarget as tb1.Writer
   dl.Input as tb1.Reader
End Configuration
```

Figure 5: Subset of the Wright Specification

messages is shown by the length and angle of the arrows. Here a controlling process (icds) sends message to a pair of sources, corresponding to enabling or disabling commands. Since the delay_line process (e.g., the target generator) receives data only through shared target buffers, only messages indicating which sources are enabled are sent. The message and timestamp processes are merely function calls that are used to randomly generate data (message) and timestamps.

The model of the target generation process was used to determine whether or not the following situations could occur.

Data From Disabled Sources. Is there a potential for calculating the target position by using data from sources that are currently disabled?

Best Data from Enabled Sources. Is there a potential for calculating a target position by using data that is less current than data currently in the target buffer?

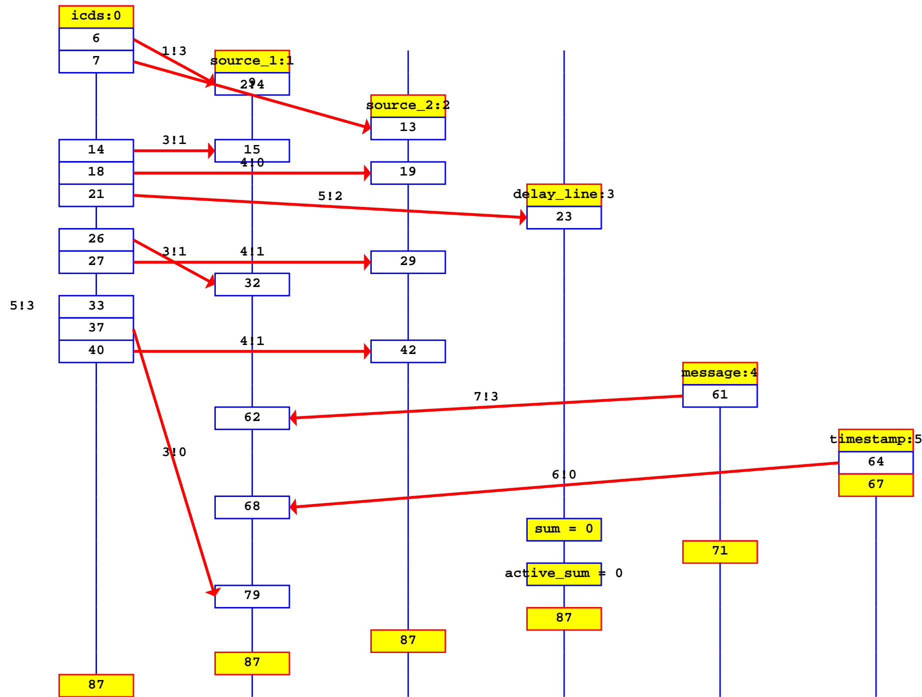

Figure 6: Message Sequence Chart for Target Generation

In the first case the interest was in determining whether or not it was possible to generate a target position by using data from inactive sources. In essence, a target position input can be read by the Target Generator, pended due to the timestamp (e.g., the timestamp indicates that the target value is not to be used until some time in the future), and subsequently promoted into use when the timestamp matches (or precedes) the current time. The potential inconsistency occurs during the time that the target is pended and is caused by the fact that a source can be disabled during this waiting period. To verify the existence of this behavior within the model, an assertion was used to compare the value of the target as computed from the currently active sources and the sum computed from the values stored in the target buffer. If, at the time that values from multiple target buffers are being used to compute a target, the value computed from the target buffers is not identical to the sum that can be computed from the currently active sources, then the data is inconsistent.

Within the specification in Figure 8, the statement `assert( active_sum == sum )`, is used to verify this scenario. Figure 9 shows the output of the Spin model checker for the corresponding assertion. Specifically, the figure shows that the assertion failed as well as statistics on the verification

```
proctype source_1 (chan cds){
   chan cmd;
   chan ts = [1] of { int };
   chan msg = [1] of { int };
   int active_or_inactive;
   cds?cmd;
   cmd?active_or_inactive;
   do :: (msgs_generated < max_msgs) &&
         (active_or_inactive == true) ->
         if :: run message(msg);
               msg?o_tb1;
               run timestamp(ts);
               ts?s1_ap;
               msgs_generated = msgs_generated + 1;
            :: skip;
         fi;
      :: (done != true) -> cmd?active_or_inactive;
      :: (done == true) -> break;
   od
}
```

Figure 7: Promela Specification of Target Source

run.

The second case involves the following situation. As before, a target from a source is read, potentially pended, and eventually promoted. Because of the sequencing of events, a new target value from the source can overwrite the recently promoted target and, based on the timestamp, be valid for immediate use. Again we used an assertion to verify the existence of this behavior, this time checking to see whether recently promoted data has been overwritten with older data.

In order to determine whether these cases were also present in the code, we examined source files and were able to verify that the situations, as documented and as specified with Wright, did in fact exist in an early, pre-flight version of the source code.

In each of these cases, the use of a non-locking buffer coupled with the target-generator algorithm provided the potential for intermittent values that are inconsistent with the desired and current target. The interferometry project engineers confirmed that the Spin model checker accurately modeled the software behavior in both situations. In the first case, a target from a currently disabled target source may still be activated. In the second case, a newly received target with a less-current timestamp can overwrite an active target. However, in neither case is the software behavior contrary to intent, given the underlying assumptions about the operational use of the software.

```
proctype target_generator (chan valid)
{
   int v, active_sum;
   int prev1 = s1;
   int prev2 = s2;
end_tg:
   do
   :: (msgs_generated < max_msgs) ->
      /* "activation" of pended targets achieved by
         maintaining previous value of s1 or s2 */
      prev1 = s1; prev2 = s2;
      /* read new targets from active target sources */
      valid?v;
      if
      :: (v == NONE) -> active_sum = 0;
      :: (v == S1_ONLY) -> s1 = o_tb1; active_sum = s1;
      :: (v == S2_ONLY) -> s2 = o_tb2; active_sum = s2;
      :: (v == BOTH) -> s1 = o_tb1; s2 = o_tb2; active_sum = s1 + s2;
      fi;
      /* if either of the following doesn't hold, then
         less current data is being used for the target */
      if
      :: (s1_ap < now) -> assert(prev1 == s1);
      :: (s2_ap < now) -> assert(prev2 == s2);
      fi;
      /* check if pended or not */
      if
      :: (v > NONE) ->
         if
         :: ((s1_ap <= now) && (s2_ap <= now)) -> sum = s1 + s2;
            assert( active_sum == sum );  /* if not, then data is taken from
                                             inactive sources */
         :: ((s1_ap <= now) && (s2_ap > now)) -> sum = s1;
         :: ((s1_ap > now) && (s2_ap <= now)) -> sum = s2;
         :: ((s1_ap > now) && (s2_ap > now)) -> skip;
         fi;
      :: (v == NONE) -> skip;
      fi;
      /* compute target */
      printf("v = %d, sum = %d, s1 = %d, s2 = %d\n",v, sum, s1, s2);
      printf("MSC: sum = %d\n", sum);
      printf("MSC: active_sum = %d\n", active_sum);
      /* reset sum */
      s1_ap = now; s2_ap = now;
      sum = 0; active_sum = 0;
   :: (msgs_generated >= max_msgs) -> break;
   od;
   done = 1;
}
```

Figure 8: Promela Specification of Target Generator

```
pan: assertion violated (active_sum==sum) (at depth 411)
pan: wrote pan_in.trail
(Spin Version 3.2.3 -- 1 August 1998)
Warning: Search not completed
+ Partial Order Reduction
+ Compression

Full statespace search for:
never-claim          - (not selected)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid endstates - (disabled by -E flag)

State-vector 164 byte, depth reached 434, errors: 1
  891047 states, stored
1.2657e+06 states, matched
2.15674e+06 transitions (= stored+matched)
       3 atomic steps
hash conflicts: 1.13363e+06 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
156.824 equivalent memory usage for states (stored*(State-vector + overhead))
19.440 actual memory usage for states (compression: 12.40%)
State-vector as stored = 10 byte + 12 byte overhead
2.097  memory used for hash-table (-w19)
0.240  memory used for DFS stack (-m10000)
21.896 total actual memory usage

nr of templates: [ globals procs chans ]
collapse counts: [ 18038 4 4 13 13 156 14 ]
0.83user 33.51system 0:34.33elapsed 100%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (108major+5316minor)pagefaults 0swaps
```

Figure 9: Spin Output of Verification for the Disabled Source scenario

5.3 Lessons Learned

The construction and analysis of a formal model of the interferometer product line provides some insight into the behavior shared across products and is a mechanism for supporting automated analysis. In addition, the formal model provides an infrastructure for analyzing future changes.

- *Model Checking Common Behavior to Determine Robustness and Fault-Tolerance.* The automated analysis of the interferometer architecture using the Spin model checker was greatly facilitated by the availability and use of the Wright and ACME ADLs. In effect, by using this combination of tools, we were able to use model checking in a manner that was directed by the structure and behavior of a software architecture. That is, the software architecture specification was used to direct the model checking activity by facilitating identification of

potentially interesting points of interaction in the interferometer architecture. Given the fact that any behavior observed in the architecture is potentially replicated among all product line members, we found that the approach was a good complement to the manual analysis activities.

- *Model Checking Modifications.* One area of high risk during software modification is the interfaces between various components and their connectors. In particular, changes in interfaces can often lead to potential mismatches between provided and expected behavior. By constructing a formal model of the interaction between components through the corresponding connectors, some assurance can be attained regarding communication via shared buffers. In addition, the formal model provides an infrastructure for analyzing future changes to the communication properties between components.

- *Lightweight Formal Methods.* From a formal methods perspective, the investigations described here provide yet another example of the value of lightweight formal analysis. That is, we were able to obtain a reasonably high payoff in analysis with a small amount of specification effort.

6 Conclusion

The experience described here regarding the recovery, specification, and analysis of a high-performance, scientific product line may be useful to practitioners in several ways. First, the approach used was adapted to the realities of an existing product line architecture. This meant that the architecture had evolved away from the initially documented architecture and that recovery of the actual architecture formed an initial step.

Secondly, the need for accurate representation of a set of highly complicated systems with tight performance constraints required iterative review by domain experts. The results of the architectural recovery were thus captured in an ADL model that supported review as well as subsequent inquiries.

Thirdly, a key concern with a product line intended to span several decades was the adequacy of the architecture's modifiability in implementing required variabilities in the future systems. To address this issue, the architecture was analyzed against a set of representative scenarios exemplifying the required modifiability attributes. It was found that high-performance requirements are often inflexible, and that this can limit the modifiability of the architecture.

Fourthly, the criticality of the application domain motiviated the use of additional, formal analysis of key critical quality attributes such as robustness and fault-tolerance at the architectural level. Automated tools and model checking were used to evaluate the consequences of architectural decisions for the product line. The model checking provided additional assurance that some specific architectural features, flagged by the analysts as possibly vulnerable, were correct. While the earlier scenario analysis addressed issues related to the modifiability of the interferometer architecture for a product line, the automated analysis was primarily of use for analyzing quality attributes such as robustness and fault-tolerance, which were viewed as common across product line members.

The application of this combined approach to the interferometer product line architecture resulted in some measurements of both the flexibility and limits of its architectural style that assisted the project in planning for future missions. Similar combined approaches may be useful for developers of other product lines where an architecture is to be recovered from existing systems or where the architecture must support high-performance requirements.

Acknowledgments

We thank Dr. John C. Kelly for his continued support of this work. We thank Dr. Braden E. Hines, Dr. Charles E. Bell, and Thomas G. Lockhart for helpful discussions and explanations regarding the reuse of interferometry software. Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Funding was provided under NASA's Code Q Software Program Center Initiative, UPN #323-08 and a NASA/ASEE Summer Faculty Fellowship.

References

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

Ardis, M. A. and Weiss, D. M. (1997), Tutorial: Defining families: The commonality analysis, *in* 'Proceedings of ICSE '97'.

Bass, L., Clements, P. and Kazman, R. (1998), *Software Architecture in Practice*, Addison Wesley.

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

Clements, P. and Northrup, L. (2002), *Software Product Lines, Practices and Patterns*, Addison-Wesley.

Colavita, M. M., Wallace, J. K., Hines, B. E., Gursel, U., Malbet, F., Palmer, D. L., Pan, X. P., Shao, M., Yu, J. W., Boden, A. F., Dumont, P., Gubler, J., Koresko, D. C., Kulkarni, S. R., Lance, B. F.,

Mobley, D. W. and van Belle, G. T. (1999), 'The Palomar Testbed Interferometer', *Astrophysical Journal* **510**, 505–521.

Danner, R. and Unwin, S., eds (1999), *Space Interferometry Mission: Taking Measure of the Universe*, Jet Propulsion Laboratory.

Gannod, G. C. and Lutz, R. R. (2000), An Approach to Architectural Analysis of Product Lines, *in* 'Proceedings of the International Conference on Software Engineering (ICSE) 2000', pp. 548–557.

Gannod, G. C., Lutz, R. R. and Cantu, M. (2001), Embedded software for a space interferometry system: Automated analysis of a software product line architecture (invited), *in* 'Proceedings of the International Conf. on Performance, Computing and Communications', pp. 145–150.

Garlan, D., Monroe, R. T. and Wile, D. (1997), ACME: An Architecture Description Interchange Language, *in* 'Proceedings of CASCON'97', Toronto, Ontario, pp. 169–183.

Gomaa, H. and Farrukh, G. A. (1999), A reusable architecture for federated client/server systems, *in* 'Proc Fifth Symposium on Software Reusability', pp. 113–121.

Hoare, C. (1985), *Communicating Sequential Processes*, Prentice Hall.

Holzmann, G. J. (1997), 'The Model Checker Spin', *IEEE Transactions on Software Engineering* **23**(5), 279–295.

Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T. and Carriere, S. (1998), The Architecture Tradeoff Analysis Method, *in* 'Proceedings of ICECCS', pp. 68–78.

Luckham, D. and Vera, J. (1995), 'An Event-Based Architecture Definition Language', *IEEE Transactions on Software Engineering* **21**(9), 717–734.

Lutz, R. (2000), 'Extending the product family approach to support safe reuse', *The Journal of Systems and Software* **53**(3), 207–217.

Origins Science Committee (2000), 'Origins Science Roadmap 2000', NASA.

Perry, D. E. (1998), Generic architecture descriptions for product lines, *in* 'Proc. of ARES II: Software Architectures for Product Families (LNCS 1429)', Springer-Verlag, pp. 51–56.

Schwanke, R. and Lutz, R. (2001), 'Experience with Architectural Design of a Modest Product Family', (submitted).

Weiss, D. M. and Lai, C. T. R. (1999), *Software Product-Line Engineering*, Addison-Wesley, Reading, MA.