An Approach to Architectural Analysis of Product Lines

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

Robyn R. Lutz[§]
Jet Propulsion Laboratory
4800 Oak Grove Drive
M/S 125-233
Pasadena, CA 91109-8099
(515) 294-3654
rlutz@cs.iastate.edu

ABSTRACT
This paper addresses the issue of how to perform architectural analysis on an existing product line architecture. The contribution of the paper is to identify and demonstrate a repeatable process for the analysis of an existing product line architecture. The approach defines a "good" product line architecture in terms of those quality attributes required by the particular product line under development. It then analyzes the architecture against these criteria by both manual and tool-supported methods. The phased approach described in this paper provides a structured analysis of an existing product line architecture using (1) formal specification of the high-level architecture, (2) manual analysis of scenarios to exercise the architecture's support for required variabilities, and (3) model checking of critical behaviors 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 and evaluate the approach.

Keywords
Software Architecture, Software Architecture Analysis, Product Lines, Interferometry Software

1 INTRODUCTION

A *software product line* is a collection of systems that share a managed set of properties that are derived from a common set of software assets [4]. A product line approach to software development is attractive to most organizations due to the focus on reuse of both intellectual effort and existing tangible artifacts. The systems in a software product line usually share a common architecture. For a new product line, many alternative architectures are derived from requirements and one is selected as the "baseline" or "core" for future systems. For a product line that leverages existing systems, an architecture may already be in place with organizational commitment to its continued use.

This paper addresses the issue of how to perform architectural analysis on an existing product line architecture. The contribution of the paper is to identify and demonstrate a repeatable process for the analysis of an existing product line architecture. Throughout the paper, application to a software product line of spaceborne telescopes is used to explain and evaluate the approach. The approach defines a "good" product line architecture in terms of those quality attributes required by the particular product line under development. It then analyzes the architecture against these criteria by both manual and tool-supported methods.

This paper demonstrates the analytical value of specifying an existing architecture with an Architectural Description Language (ADL), both in terms of identifying architectural mismatches with the product line and in terms of providing a baseline for subsequent automated analyses. Once an ADL model exists, the architecture can be exercised manually by measuring the effect on the architecture of each of a set of scenarios selected to capture the required attributes (e.g., modifiability, fault tolerance). We found that this technique was particularly effective at verifying whether or not the architecture supported planned variabilities within the product line.

Further verification of the architecture involves automated tool support to analyze key, common behaviors. We were particularly interested in the adequacy of the fault-tolerant behavior of a critical data interface common to all systems. Model checking of the targeted behaviors allows demonstration of the consequences of some architectural decisions for the product line.

The phased approach described in this paper provides a structured analysis of an existing product line architecture using (1) architectural recovery and specification, (2) manual analysis of scenarios to exercise the architecture's support for required variabilities, and (3) model checking of critical behaviors at the architectural level

[*]This research was performed while this author was a visiting researcher at the Jet Propulsion Laboratory.
[†]This author supported in part by a NASA Summer Faculty Fellowship.
[‡]Contact Author.
[§]Mailing address: Dept. of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA 50011-1041.

Draft of paper submitted to ICSE 2000.

that are required for all systems in the product line.

The rest of the paper is organized as follows. Section 2 provides background relating to software architecture, product lines, and the interferometer application. Section 3 describes the three-step approach outlined above in greater detail. Section 4 presents and discusses the results from the manual and tool-supported analyses. Section 5 briefly describes related work. Section 6 offers concluding remarks and indicates some directions for future research.

2 BACKGROUND

This section describes background material in the areas of software architectures, software product lines, and interferometry.

2.1 Software Architectures

A *software architecture* describes the overall organization of a software system in terms of its constituent elements, including computational units and their interrelationships [15]. In general, an architecture is specified as a configuration of components and connectors. A component is an encapsulation of a computational unit that has an interface that specifies the capabilities that the component can provide and the ways that a component delivers its capabilities. The interface of a component is specified by the the *type* of the component, by one or more *ports* supported by the component, and by the *constraints* imposed on the ports of the component, where component types are intended to capture architectural properties. Ports are the interaction points through which a component exchanges resources with its environment. Port specifications specify the signatures, and optionally, the behaviors of the resource. Logic-based formal specifications may be attached to a port to precisely capture behavioral properties. Formal specifications of this sort enable a semantic-based approach to analyzing architectural behavior.

Connectors encapsulate the ways that components interact. A connector is specified by the *type* of the connector, the *roles* defined by the connector type, and the *constraints* imposed on the roles of the connector. A connector defines a set of roles for the participants of the interaction specified by the connector. Connector types are intended to capture recurring component interaction styles.

Components are connected by configuring their ports to the roles of connectors. Each role has a domain that defines a set of port types and only the ports whose types are in the domain can be configured to the role.

Another important concept in the area of software architectures is the concept of an *architectural style*. An architectural style defines patterns and semantic constraints on a configuration of components and connectors. As such, a style can define a set or family of systems that share common architectural semantics [13]. For instance, a *pipe and filter* style refers to a pipelined set of components whereas a *layered* style refers to a set of components that communicate via hierarchies of interfaces. The distinction between architectural style and architecture is an important concept throughout the work described here. As one would expect, all the systems in our example product line share a base architectural style and a set of shared software components that are organized and communicate in certain prescribed manners. However, there are architectural variations among the systems regarding the number of components and connectors, with some systems replicating portions of the baseline reference architecture in their individual architectures.

2.2 Product Lines

Bass, Clements, and Kazman define a *software product line* as "a collection of systems sharing a managed set of features constructed from a common set of core software assets" [4]. These assets typically include a base architecture and a set of shared software components. The software architecture for the product line displays the commonality that the systems share and provides the mechanisms for variability among the products. The systems in the product line are referred to as *members* or *derivatives* of the baseline architecture or architectural style.

2.3 Interferometers

The product line of interest in this work is a set of interferometer projects under development by NASA's Jet Propulsion Laboratory. An interferometer, in this context, is a collection of telescopes that act together as a single, very powerful instrument. Interferometers will be used to explore the origins of stars and galaxies and to search for Earth-like planets around distant stars. An interferometer combines the starlight it collects from telescopes in such a way that the light "interferes" or interacts to increase the intensity of the observation. This allows precise measurements to be made.

Among the NASA interferometers either proposed or under development for launch in the next twenty years are the Space Interferometry Mission (SIM), the New Millenium Program's Space Technology-3 (ST-3), and the Terrestrial Planet Finder (TPF), as well as the ground-based Keck Interferometry Project and the Mount Palomar Interferometer [8, 14]. ST-3 and TPF use telescopes that fly in formation on separated spacecraft but work together as a single instrument, while the other projects involve multiple telescopes working together on one or more fixed axes.

Among the components shared by the interferometer systems and discussed in this paper are the Delay Line, the Fringe Tracker, and the Internal Metrology. The Delay Line software compensates for the difference in time between when starlight arrives at the separate mirrors. The Fringe Tracker software 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 software provides input to the Delay Line regarding small changes in distances among parts of the interferometer that must be included in its calculations.

In previous work, we analyzed commonalities and variabilities of the JPL interferometry software project [12]. The software in these interferometers has a high degree of commonality with a managed set of shared features built from core software components [3]. A group of developers with a strong background in interferometer software is in place at JPL to develop and provide to the interferometer projects a set of reusable, generic software components.

3 APPROACH

In this section we describe the approach that was used to analyze an interferometer software product line. Section 3.1 summarizes the overall process used during the project and introduces the architectural recovery, discovery, and specification of the existing product line; Section 3.2 describes the manual analysis process used to measure quality attributes related to product lines; and Section 3.3 describes the behavioral analysis performed using automated tool support.

3.1 Process

A software architecture is one key required element that should be present in order to analyze software for product line "fitness" since it is the architecture, above any other artifact, that is being reused. One of the properties of this particular product line is that although an architecturally-based product line approach was not used in the construction of the software, the artifacts (both conceptual and physical) were being used in a manner indicative of a product line approach. As such, several software products had been developed or were in the process of being developed based on the core architecture.

For the interferometer software, we performed the following architecture-centered steps:

1. Architecture recovery, discovery, and specification
2. Manual architectural analysis
3. Tool assisted architectural analysis

The first step, architecture recovery, discovery, and specification, was used in order to facilitate two goals: 1) to familiarize the analysts with the problem domain and implemented solution, and 2) to support construction of a software architectural representation that was consistent with current standards and vocabulary. For this step, documentation, source code, and developer communication was used to assist in the construction of a reasonable specification of the software architecture. The resulting specifications formed the basis for all subsequent analyses, manual and automated.

The software architecture recovered in the first step formed the baseline or core architecture for the interferometer product line. The assumption in this step (later confirmed by the analysis described below) was that, although changes in software code are frequent, significant modifications to the software architecture are infrequent. As such, a reasonable, initial view of the software architecture can be derived from existing design documents and later modified as new information is recovered.

To aid in the validation of the models constructed in the first step, we consulted with the project engineers to determine the accuracy of the architecture as documented in comparison with how the project engineers viewed the architecture. This information was instrumental in constructing a more accurate view of the interferometer architecture.

To further validate the accuracy of the core architecture and its scalability to the existing and planned products in the product line, we then compared the core to the individual product line derivatives. To facilitate the comparison, we used Table 1 as a medium for communication with several developers. In the table, 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 served as a simple way to represent features of the architecture that are *common* in behavior to each potential derivative, but can potentially *vary* in multiplicity based on the number of potential starlight collectors or "arms". For each derivative, we consulted with developers to verify that the number of components listed in the table was consistent with individual mission plans.

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

The next phase of the approach was to perform a number of analyses in order to help determine whether the architecture was amenable to a product line develop-

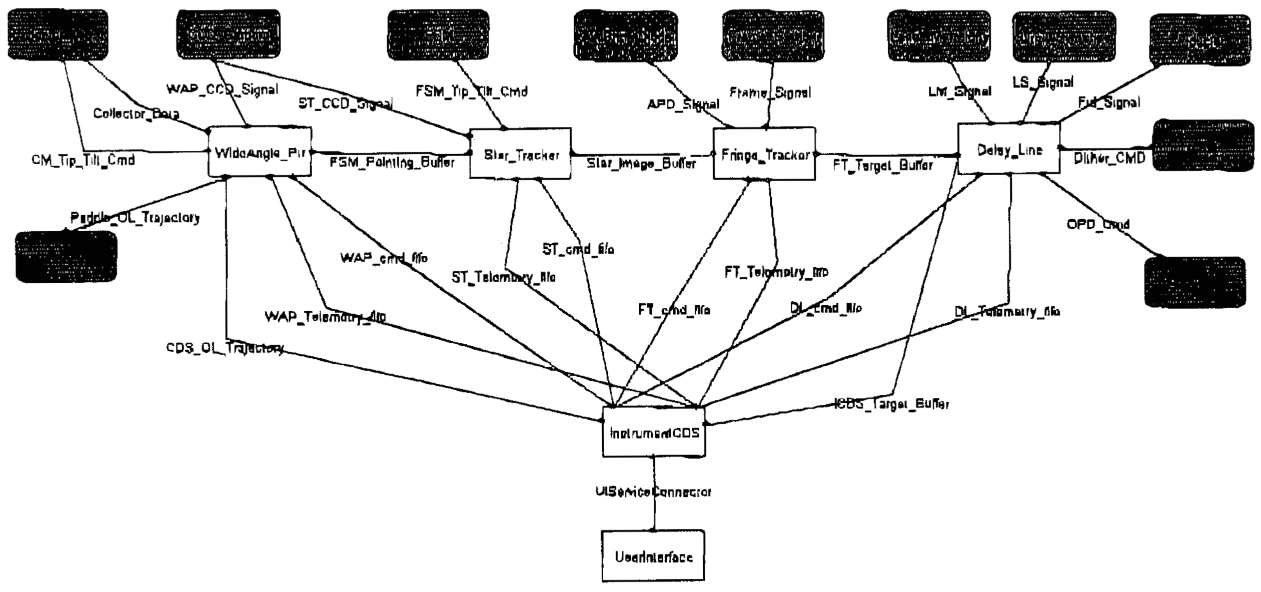

Figure 1: Interferometer Software Architecture

ment 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. In addition, we were interested in performing behavioral analysis in order to study how behavioral interactions in the core architecture might potentially impact derivatives.

The remainder of this section is divided into *Manual Architectural Analysis* and *Analysis Using Automated Support Tools*. One of the interesting aspects of this bifurcation of the analysis along manual and automated analysis lines is that the quality attributes that fall into the class of *variabilities* seem to be supported only by manual analysis techniques whereas the *commonalities* seem to be supported in some manner by automated tools. As the work described here is only a single point of data, we do not attempt to explain the observation, although we do find it interesting and recognize the need for further investigation along these lines.

3.2 Manual Architectural Analysis

Bass, Clements, and Kazman divide quality attributes into those that can be discerned by observing the system at runtime and those that cannot [4]. Of the ones that cannot be observed at runtime, modifiability is the key property required by the interferometer product line. Modifiability, according to Bass et al., "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., identify four categories of modifiability: Extensibility or changing capabilities, Deleting capabilities, Portability (adapting to new operating environments), and Restructuring.

To evaluate the modifiability of the interferometry product line architecture, we found examples of each of the four categories of modifiability in the requirements specification of a system currently being developed in the product line . We then manually analyzed the effect of each change on the specified architecture. This interferometer system was chosen because its requirements were well documented and its requirements presented a good challenge to the modifiability of the baseline architecture.

The approach used is very similar to SAAM [9], a scenario-based method for analyzing architectures. A scenario 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. The advantage of the scenario-based approach is 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").

The interferometer product line has significant requirements that fall under each of the four categories of modifiability.

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

- Deletions involve changes required to support the

Modifiability 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

incremental capabilities of the various testbeds and prototypes. For example, one testbed uses pseudostar (simulated) input rather than actual starlight, whereas the science interferometers will use direct starlight as input.

- Portability changes are widespread, since different interferometers in the product line will have different numbers of starlight collectors, mirrors, telescopes, etc. In addition, different systems 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 a variability among the systems.

- 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.

As shown in Table 2, nine representative changes were selected to evaluate the modifiability of the architecture: three extensibility changes, one deletion, four portability changes, and one restructuring. All these changes are 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. A discussion of the results of the application to the baseline interferometer architecture and, more generally, of the advantages and disadvantages of this approach can be found in Section 4.

3.3 Analysis using Automated Support Tools

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, it was our intent to identify tools that could be adopted with little overhead, while still satisfying the objective of formally analyzing the architectural behavior. This meant that the selected tools should have a reasonable level of support and documentation.

Analysis Steps

The following tasks were identified as the critical path for achieving our automated analysis objectives: (1) Architecture specification in an ADL, (2) Formal specification of behavior, and (3) Analysis of behavior. The approach used in the selection of notations and tools is described here. The results of the tool-supported analysis are described and discussed in Section 4.

ACME [5] ADL and ACMEStudio [1] were chosen for the specification of the architecture. ACME is an architecture description language that has been used for high-level architectural specifications [5]. ACME contains constructs for embedding specifications written in a wide variety of existing ADLs, making it extensible to both existing and future specification languages. ACME is supported by an architectural specification tool, ACMEStudio, that supports graphical construction and manipulation of software architectures. Analysis of the design documents yielded the software architecture depicted in Figure 1.

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. The Wright ADL was used for the formal specification of behavior. Wright [2] is an ADL based on the CSP specification language [6]. 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, existing Wright tools support the ACME ADL, so provided a clean interface with the existing specification.

The final step involved using the formal specifications to analyze behavior of various aspects of certain interactions between components in the architecture. To increase confidence in the validity of the formal analysis, source code 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 [7] is a symbolic 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.

4 RESULTS

4.1 Architecture Specification

As shown in Figure 2, the original documentation for the interferometry software depicts the architecture using a layered style. However, during the analysis and subsequent specification of the architecture, it was discovered that the architecture, as documented, 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. Consequently, the high-level interferometer architecture was re-specified in a style that was consistent with the services and behaviors described in lower-level documentation. The resulting architecture, shown in Figure 1, more accurately specified the architecture as a heterogeneous architecture with a collection of communicating processes as well as a constrained pipe and filter interaction between the Instrument CDS and all of the other remaining components.

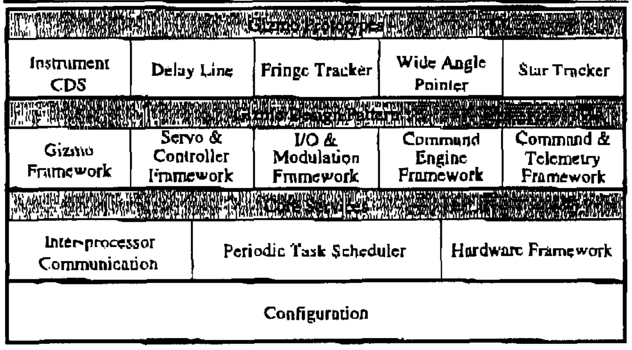

Figure 2: Original Core Architecture

4.2 Manual Analysis Results

The baseline architecture shows the commonality that exists among the members of the 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.

For the interferometer product line, a key aspect of the "goodness" of the baseline architecture was how modifiable it was. It was with the goal of exercising the product line architecture that we considered the effect on the architecture of each of nine representative modifiability scenarios, drawn from the documentation.

Effect on architecture of scenarios

Table 2 summarizes the results of our manual analysis of the product line architecture for modifiability via the nine scenarios described in Section 3.2. Column 1 lists 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, 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 accomodated 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.

Discussion

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 redundancy. Several of the scenarios involved adding multiple, identical components or connectors. However, these copies are not redundant, in the sense of adding robustness, 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, redundancy does not add robustness for the most part; there are not spare units or alternate data paths.

Performance. One of the unusual aspects of this application is that the range and scope of the variabilities tend to be non-negotiable. This is due to the very tight performance and accuracy requirements on the interferometry missions. For example, an upcoming interferometer, the Space Interferometry Mission (SIM), requires precision at the level of picometer metrology and microarcsecond astrometry. To achieve this level of precision, significant real-time constraints exist with limited flexibility to accomodate reuse concerns. Performance requirements on each mission also drive the choice of hardware, algorithms, and added capabilities. The consequence for reuse is that in trade-offs of modifiability vs. performance, performance wins.

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 interferometry project is committed to keeping the architectural style stable. Most importantly, this demonstrates itself in their maintaining the commonality of the interfaces. The number of interfaces is not constant among product line members, but 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 has done a good job of designing for evolvability.

Repeatable process. The manual analysis of the architecture is a repeatable process that can be applied to product lines. The process is as follows:

1. Identify anticipated changes from available documentation and project information. These anticipated changes form product line variabilities that the baseline architecture must accomodate.

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.

4.3 Analysis Using Automated Support Tools

While the manual analysis addressed issues related directly to the use of the interferometer architecture as a product line, the automated analysis was primarily of use for analyzing behavior viewed as common across product line members. As such, any behavioral properties (both positive and negative) discovered at the architectural level were likely to be common to all members of the product line.

Spin Verification

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 star targets to the Delay Line component by several other components. The Target Buffer connector was viewed as a possible concern, especially in light of the non-locking feature. It was determined that behavior involving this connector should be 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 3. 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 the Promela specification found in Figures 4 and 5 with the intention of determining whether or not the following situations could occur.

```
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 3: Subset of the Wright Specification

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 to calculate a target position by using data that is less current than data currently in the target buffer?

In the first case, we were interested 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.

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.

```
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_tbl;
         run timestamp(ts);
         ts?s1_ap;
         msgs_generated = msgs_generated + 1;
      :: skip;
      fi;
   :: (done != true) -> cmd?active_or_inactive;
   :: (done == true) -> break;
   od
}
```

Figure 4: Promela Specification of Target Source

Using the Spin model checker, it was verified that these situations do in fact exist. 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 anomalous 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.

Discussion
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.

```
proctype delay_line (chan valid)
{
   int sum;
   int v;
   do
   :: (msgs_generated < max_msgs) ->
      /* "activation/promotion" of
          pended targets achieved
          by maintaining previous
          value of s1 or s2 */
      /* read new targets from active target sources */
      valid?v;
      if :: (v == 0) -> skip;
         :: (v == 1) ->
            s1 = o_tb1;
            o_tb1 = clear;
         :: (v == 2) ->
            s2 = o_tb2;
            o_tb2 = clear;
         :: (v == 3) ->
            s1 = o_tb1;
            s2 = o_tb2;
            o_tb1 = clear;
            o_tb2 = clear;
      fi;

      /* check if pended or not */
      if :: (v > 0) ->
         if
         :: ((s1_ap <= now) && (s2_ap <= now)) ->
               sum = s1 + s2;
         :: ((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 == 0) -> skip;
      fi;

      /* compute target */
      printf("v = %d, sum = %d, s1 = %d,
            s2 = %d\n",v, sum, s1, s2);
      /* reset sum */
      s1_ap = now;
      s2_ap = now;
      sum = 0;
   :: (msgs_generated >= max_msgs) -> break;
   od;
   done = 1;
}
```

Figure 5: Promela Specification of Delay Line

5 RELATED WORK

As described in Section 3.2, the Software Architecture Analysis Method (SAAM) is a scenario-based method for architectural assessment. A related architectural analysis method is the Architecture Tradeoff Analysis Method (ATAM) [10]. 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.

Rapide [11] 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. In our work, the motivation for choosing a particular technique was based on a desire to eventually transfer the technology to the project engineers. In addition, we were interested in interoperability with other tools. As such, we 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 in its specification. However, we recognize that several alternatives such as Rapide exist and are investigating the possibility of performing similar analyses with those tools.

6 Conclusion

The work described here identifies and demonstrates a process for analysis of an existing product-line architecture. The results of the architectural recovery and discovery are captured in an ADL model to support subsequent inquiries. The architecture is manually analyzed against a set of representative scenarios that capture the required quality attributes. Further analysis of critical behaviors at the architectural level uses automated tools and model checking to evaluate the consequences of architectural decisions for the product line. The application of this combined approach to the interferometer's product line architecture resulted in some measurements of both the flexibility and limits of its architectural style that could assist the project.

Further work is planned in several areas. In previous work we have used formal techniques for reverse engineering of code [?]. We plan to investigate how reverse engineering can also be used to assist in the recovery of product line assets. This may involve consideration of different analysis frameworks (e.g., Rapide) that offer fully integrated environments and investigation of Wright/Spin translations. We also plan to pursue the relationship between product-line commonalities/variabilities and analysis techniques. The observation here that quality attributes relating to variabilities (e.g., modifiability) seem best supported by manual analysis techniques whereas commonality attributes are best analysed with automated tool support (e.g., model checking) merits further study. Finally, we would like to make more precise the role of architectural issues in product line decision models.

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.

REFERENCES

[1] Acmestudio: A graphical design environment for acme. http://www.cs.cmu.edu/~acme/ AcmeStudio/ AcmeStudio.html.

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

[3] M. A. Ardis and D. M. Weiss. Tutorial: Defining families: The commonality analysis. In *Proceedings of ICSE '97*, May 1997.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[5] D. Garlan, R. T. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

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

[7] G. J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[8] Jpl interferometry projects. http://huey.jpl.nasa.gov/ice/ice_projects.html.

[9] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, 1996.

[10] R. Kazman, M. Klein, M. Barbacci, H. Lipson, T. Longstaff, and S. Carriere. The Architecture Tradeoff Analysis Method. In *Proceedings of ICECCS*, 1998.

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

[12] R. Lutz. Extending the product family approach to support safe reuse. *The Journal of Systems and Software*, 2000.

[13] N. Medvidovic and R. N. Taylor. Exploiting architectural style to develop a family of applications. *IEE Proc. in Software Engineering*, 144(5-6):237–248, Oct-Dec 1997.

[14] Origins program. http://origins.jpl.nasa.gov.

[15] M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.