Submitted for publication.

Understanding Tradeoffs among Different Architectural Modeling Approaches

Roshanak Roshandel[*], Bradley Schmerl[†], Nenad Medvidovic[*], David Garlan[†], Dehua Zhang[†]

[*]Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
{roshande,neno}@usc.edu

[†]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{schmerl,garlan,zhangd}@cs.cmu.edu

Abstract

Over the past decade a number of architecture description languages (ADLs) have been proposed to facilitate modeling and analysis of software architecture. While each claims to have various benefits, to date there have been few studies to assess the relative merits of these approaches. In this paper we describe our experience using two ADLs to model a system initially described in UML, and compare their effectiveness in identifying system design flaws. We also use this experience to shed light on techniques for extracting architectural models from a UML system description.

1. Introduction

A critical component of an engineering basis for software architecture is the availability of notations for formal architectural representation and analysis. Indeed, over the past decade there has been considerable research into this issue, leading to a large number of proposals for architectural description languages (ADLs). Each ADL typically provides some unique capabilities for modeling, together with tools to carry out analyses of properties. For example, one ADL may be suitable for code generation, while another may be better suited for formal analysis with respect to topology, interfaces, or interaction protocols.

Unfortunately, in the pantheon of architectural modeling approaches, it is often unclear what aspects of architectural design the different approaches focus on, how they relate, and what benefits they can provide in increasing the quality of a software system. Thus, when making decisions about how to model a system, it is often hard to determine the best approach to take, and whether the effort required in producing multiple models would provide significant benefits.

A case in point is the role of modeling in the design of SCRover, a mobile robot based on the Mission Data System (MDS) architectural style [6][7] from NASA's Jet Propulsion Laboratory (JPL), and built using the MDS implementation framework. MDS is an architectural approach created by JPL to streamline the development of software for space missions. SCRover was designed and developed in collaboration with the MDS team and is sufficiently complex to be indicative of typical autonomous robot software used in NASA's space missions. Additionally, it is being used as a testbed for research by various institutions into increasing the dependability of NASA's space software, through NASA's High Dependability Computing Project [16].

The SCRover project represents a typical approach to developing MDS-based systems. The modeling lessons learned can thus be applied to other MDS systems. Furthermore, errors detected in the design of SCRover may be indicative of the types of errors encountered in other MDS-based systems.

The broad applicability of SCRover means that it is also an attractive testbed for applying modeling technologies. SCRover was designed in the context of the MBASE software process [21], which extensively employs UML. The architectural nature of MDS makes SCRover an ideal candidate for *architectural modeling*. Architectural aspects of SCRover are difficult to extract from the UML documentation, and automated analysis of the documentation is not practical. Instead, peer-reviews are used to analyze SCRover's design documentation.

In this paper we document our experience in using two representative ADLs, Acme [9] and Mae [18], to model SCRover. Both models were derived from the initial MBASE UML design, but were developed independently of each other, and focus on different aspects of the architecture. We describe how each approach used the SCRover documentation to develop the respective architectural models and discuss the differences that resulted from focusing on different aspects of the original documentation. We show how these differences led to the automatic detection of distinct, but complementary, classes of errors, and how automatic analysis afforded by either ADL yields better results than peer-review of the SCRover documentation for architectural defects.

The rest of this paper is organized as follows: Section 2 provides some background to software architectures and architectural modeling, and introduces MDS and SCRover. Section 3 describes the two approaches that we used to model SCRover, discusses details of each ap-

proach, and the process of developing each model. In Section 4, we evaluate the results of each approach using a detailed classification of architectural defects. Sections 5 and 6 discuss lessons learned and future work.

2. Background

2.1. Software Architecture

While there are numerous definitions of software architecture [1][17][19], the basis of all of them is the notion that an architecture describes a system's gross structure using one or more views. These views shed light on concerns such as the system's composition, its main pathways of interaction, and the key properties of its parts. Furthermore, an architectural description ideally includes sufficient information to allow analysis and critical appraisal.

At its core an ADL[14] typically represents an architectural model as a graph of interacting *components* (e.g., [5][9][15]). Nodes in the graph (the components) represent the principal computational elements and data stores of the system: clients, servers, databases, etc. Arcs are termed *connectors*, and represent the pathways of interaction between the components, which can be realized in a system by a complex base of middleware and distributed systems support. To account for various behavioral properties of a system, elements in the graph can typically be annotated with property lists, although the mechanisms for this differ across ADLs. For example, properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth).

There are a number of benefits to constraining the design space for architectures by associating a *style* with the architecture. An architectural style typically defines a set of types for components, connectors, interfaces, and properties, and may include rules that govern how instances of those types are composed. Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [5][15][17][22]. Typically, analysis of an architectural model cannot be defined rigorously without appeal to a particular style, even if that style merely states that every component must have certain properties.

2.2. Mission Data System (MDS)

The Mission Data System (MDS) [6][7] is a methodology, an architectural style, and an implementation framework designed and built by NASA's JPL. It attempts to capture decades of experience in developing space software, and represents a product family approach to space mission software. The goal of MDS is to bridge the conceptual gap between scientists, system engineers, and software developers. It also strives to provide a set of tools and methodologies that enable development of reliable systems and that reduce development costs by promoting reuse and preventing erroneous behaviors early in the development life cycle.

The MDS design methodology offers a technique for capturing data in terms of *States*, *Commands*, and *Measurements*. Moreover, MDS component types include *Controllers*, *Estimators*, *Sensors*, *State-Variables,* and *Actuators.* Instances of these interact by manipulating or communicating data. A set of constraints that govern manipulation and communication of the data among architectural elements forms the *MDS architectural style*. The MDS implementation framework offers implementation-level abstractions that adhere to the MDS style. The framework is a C++ library of approximately 250,000 lines of code that provides over 35 reusable packages for common functionality such as state-oriented control, event logging, time services, data management, visualization, and units of measurement.

2.3. SCRover

The SCRover project [2] is a collaborative effort by the University of Southern California (USC) and JPL to develop a campus public safety robot performing mission scenarios representative of JPL's planetary rover missions. It was designed and built using the MDS methodology and implementation framework. In the version of SCRover used in this report, its functionality includes basic robot navigation and control capabilities such as moving along a wall, turning as needed, and avoiding obstacles. In addition, the robot reports images obtained by its camera, range information, and sensor and battery health data. The implementation includes over 3000 lines of application-specific code in addition to the MDS framework code.

The SCRover project is intended to serve as a testbed for research and academic organizations to accelerate software engineering technology maturity and transition in the context of NASA's High Dependability Computing Project [16]. With this potential scrutiny in mind, SCRover was designed carefully using the MBASE process [3][21], with extensive documentation for requirements and design, and extensive use of design reviews.

3. Architectural Modeling for SCRover

The high-level architecture of SCRover is shown in Figure 1. In accordance with the MDS architectural style, each high level component shown in the figure will be further refined into corresponding *Controller*, *Estimator*, *Adaptor*, and *State variable* components, with connections between them. These components and connectors interact according to the stylistic constraints of MDS. The rest of this section describes each of the three models of SCRover and how the two architectural models were derived from the UML models.

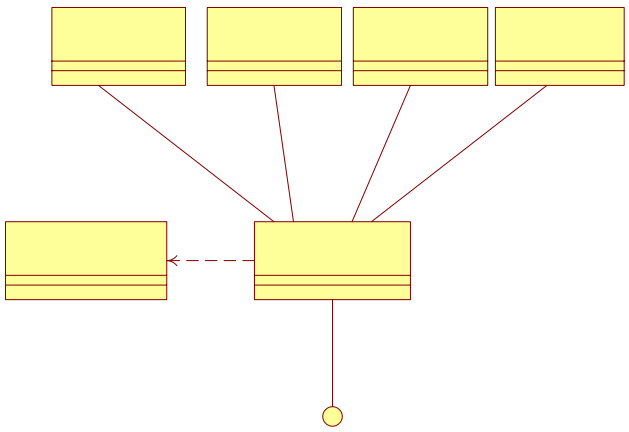

Figure 1. SCRover high-level architecture.

3.1. The MBASE-UML Modeling Approach

Initially, the architecture of the SCRover system was designed by a research group at USC using a subset of UML in the context of USC's MBASE process [21]. MBASE offers a set of integrated models that capture *product*, *success, process,* and *property models* of the system under development. The MBASE process defines an approach for negotiating requirements, capturing operational concepts, building initial design models, assessing project risks, and planning the life cycle. These aspects are captured in separate documents that are constantly maintained to ensure their consistency.

MBASE's Software and System Architecture Description (SSAD) is a 140-page document that uses a subset of UML diagrams (use-case, class, and sequence diagrams) to model the topology and interactions among different SCRover's subsystems. The SSAD also models the process for achieving the specified system goals. English prose, tables, and other conventions are used to further elaborate the functionality of the system. Specifically, the SSAD focuses on the following aspects of system design:

- *System Analysis* elaborates the goals of the system, the processes by which the goals may be achieved, and the scenarios that describe these processes.
- *Architectural Design* describes the topology of the system in terms of components and their interfaces (i.e., method signatures). It describes the components' behaviors in English, and the system's overall behaviors using sequence diagrams. Use-cases model the manner in which components are used in the system.
- *Implementation Design* further refines the components in terms of class diagrams that specify the components' attributes and their provided interfaces. Operations are associated with interfaces, and are modeled in terms of pre- and post-conditions captured in English.

The SSAD provides a starting point for understanding a system's structure and functionality. However, the lack of formal semantics associated with UML manifests itself in this process. Moreover, the use of English prose, although descriptive, hampers effective modeling of the system since automated support cannot be provided to ensure the consistency and correctness of the specification. Instead, peer-reviews of the documentation are performed to ensure both its internal consistency, and its compliance with the MDS architectural conventions.

In the context of our experience, the complexity of SCRover and the underlying MDS architectural style, together with the informality of the SSAD, motivated us to consider other modeling approaches that would enable automatic analysis and help to reveal a broader class of errors early in the development process.

3.2. The Acme Modeling Approach

To model systems written using the MDS framework, the Acme team first encoded the MDS architectural conventions as an Acme style. We based the style definition on a set of documents from JPL that prescribed in English prose the types of elements in an MDS design and the constraints on how instances of those types could be combined. The resulting style formally captures the high-level vocabulary and rules governing the architectures of all MDS systems.

The architectural style consists of six component types (e.g., *Sensor*, *Actuator*, *Controller*), eight connector types (e.g., *Command Submit*, *Measurement Request*, *State Update*), seventeen port types and eighteen role types, in addition to 39 formal rules that specify what it means to have a correct MDS system topology. Figure 2 illustrates a small segment of an architecture written in this style. This segment depicts interaction between a *Controller*, an *Actuator*, and an *Estimator*. In this interaction the *Controller* submits a command to an *Actuator* via its *Command Submit* connector. The *Actuator* then notifies the *Estimator* that it received a command. Subsequently, the *Estimator* queries the *Actuator* to find out what the command was. Examples of the MDS rules:

1. If an *Estimator* can be notified of a command by an *Actuator*, then that *Estimator* must be able to query the *Actuator* for the command.
2. An *Actuator* must have exactly one *Controller* connected to it.
3. An *Actuator* must have the same number of *Command Submit*, *Command Notification* and *Command Query* ports (one for each type of command that it receives).

The first MDS rule above can be captured in Acme with the following predicate:[1]

```
invariant (forall e :! EstimatorT in self.components |
  (forall cnp :! CmdNotProvT in e.ports |
    (forall a :! ActuatorT in self.components |
      (forall cnr :! CmdNotReqrT in a.ports |
        (connected (cnp, cnr) ->
          (exists cqr :! CmdQryReqrT in e.ports |
            exists cqp :! CmdQryProvT in a.ports|
              connected (cqr, cqp)))))));
```

These rules are automatically checked by Acme tools.

3.2.1 The Acme Process

The Acme team primarily used the SSAD to develop an architectural model of SCRover. The process consisted of three phases that built up evidence for the types of components and connectors in the architecture.

Phase 1: *Identify components and possible connections.* Primarily using the class diagrams, the components are identified, and then the connectors are recorded based on the associations between classes in the diagrams. Because a given UML association between components may be one of several possible architectural connectors, some of these are marked as *possible connectors,* pending further evidence from the other phases.

Figure 3 gives an example of a diagram from the SSAD, showing an interaction between a *Position & Heading Estimator* and a *Hardware Adaptor*.[2] From this diagram, it is straightforward to derive the existence of an *Estimator* component. However, there are several choices for what the association in the diagram could mean:

- The *Hardware Adaptor* is being used as a *Sensor*, and either the *Estimator* is periodically requesting measurements, or it is notified of measurements from the *Sensor*. The direction of the arrow argues against the former; if it is the latter, however, there is no corresponding measurement requested by the *Estimator*, as required by MDS rules.
- The *Hardware Adaptor* is being used as an *Actuator*, and the *Estimator* is either notified of a command, or is querying a command. For reasons similar to the above, we cannot tell which connector it actually is.

To resolve the above ambiguity, we need the information from Phase 2.

Phase 2: *Refine connections based on sequence diagrams*. Evidence from messages in sequence diagrams is used to substantiate and disambiguate possible connectors obtained from Phase 1. If a message cannot be mapped to any possible connectors from Phase 1, a connector is added to the architecture based on knowledge of the MDS style. The lack of a connector is then noted as an omission from the SSAD class and interaction diagrams.

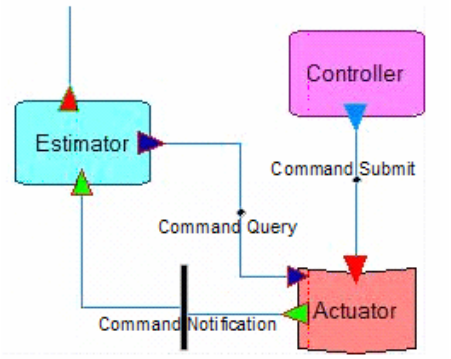

Figure 2. A *Controller/Actuator/Estimator* pattern in MDS

Figure 4 shows the portion of a sequence diagram from the SSAD that we used to disambiguate whether the *Hardware Adaptor* is being used as a *Sensor* or A*ctuator* by the *Position & Heading Estimator*, and to determine the type of connection between the two components. The method call and parameter name suggest that the *Estimator* polls the *Adaptor* for a measurement, leading to the conclusion that the hardware adaptor is being used as a sensor in this case, and that the connection is a measurement request.

Phase 3: *Review and resolve inconsistencies.* To complete the MDS architecture, the final phase reviews the results of the previous phases against all other available documentation. Any inconsistencies that arise from decisions made in Phase 1 or 2 are also reviewed. If an inconsistency is found, "reasonable" decisions based on knowledge of the MDS architectural style are made. If necessary, Phase 1 and/or 2 processes are repeated.

3.2.2 Acme View of SCRover

At the end of this process we had an architectural view of SCRover that shows the components involved in each control loop, their connections, and the relationship between control loops. For example, from the model it is evident that the *Position and Heading Controller* uses information from both SCRover's *Position* and *Heading*, as well as the nature of known *Obstacles*. It is also immediately apparent from the model that components query *Sensors* and *State Variables* directly, and are not notified of changes (which the MDS style allows).

3.3. The Mae Modeling Approach

Acme models were primarily used to capture stylistic constraints. In contrast, Mae [18][22] was used to model

[1] In this rule, self refers to the system, *italicized* words refer to predefined Acme functions, and the clause <name> :! <type> means that <name> declares the type <type>

[2] In the Acme MDS architectural style, *Actuators* and *Sensors* are distinguished as separate entities whereas in the SSAD they are bundled together in *Hardware Adaptors*. The "mode of use" attribute of an *Adaptor* dictated whether it was a *Sensor* or an *Actuator* in this process.

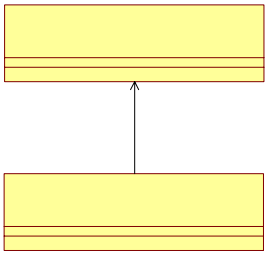

Figure 3. Portion of SSAD's component classifier diagram

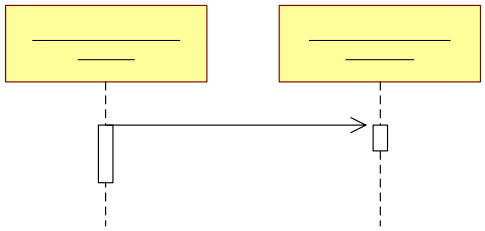

Figure 4. Portion of SSAD's sequence diagram showing interaction between *Adaptor* and *Estimator*

the refinement of the SCRover components' functionality and enable analysis of architectural mismatch. Mae is an extensible architectural evolution environment. It enables modeling, analysis, and management of different versions of architectural artifacts, and supports domain-specific extensions to capture additional system properties.

At its core Mae leverages xADL 2.0 [5], an XML-based ADL. xADL 2.0 is a collection of modularly organized XML schemas that represent components, connectors, and interfaces. Extensions to xADL may be built to represent additional architectural properties via new XML schemas. The extensions used for modeling SCRover are depicted in Figure 5(b).

Mae employs the *Static Behavior* extension to xADL's core to capture static behavioral properties of the system. Figure 5(a) depicts how a component type is specified in the context of this schema: pre- and post-conditions and invariants are used to statically describe the state of the component using a set of variables (*StateDecl*); *invariants* may constrain the values for these variables; signatures are instances of an *interface type* (not shown in the figure), and in addition to a pointer to their parent type, have a name, a direction (provided or required) and a set of *interface elements;* an interface element in turn is specified in terms of a method signature, and is mapped to one or more *operations* where associated pre- and post-conditions are captured; finally, the *subtype* field is used to capture relationships among components.

The *MDS Types* extension shown in Figure 5(b) was built by extending the static behavioral model to capture namespaces and complex inheritance information for components. Finally, the *MDS Implementation* schema links architectural artifacts to their implementation-level counterparts. This extension is primarily used for code generation from the architectural model.[3]

3.3.1 The Mae Process

These extensions in concert define a specific ADL that can capture all the functional requirements of architectural elements in the context of MDS. To build a Mae architectural model of SCRover, we needed to refine the models in the MBASE-SSAD document. This process was centered on the following two activities:

Phase 1: *Identify Components and Connectors.* Similarly to the experience in [6], we extensively used the existing class diagrams to identify components and connectors in the system. Sometimes components were formed by merging two or more classes that together would correspond to an MDS component type (*Controller*, *Estimator*, etc.). Once the components were identified, their interconnections were established by leveraging associations in the class diagrams. Additional connectors were also identified in the next phase, when the components' interfaces were modeled.

Phase 2: *Refine component services in terms of interfaces and their associated operations.* Once the components were identified, we needed to specify the services they provide to the system, as well as those they require from the system. The former were directly obtained from the UML class diagrams, while the latter were derived using a combination of class diagrams and the detailed information provided in the Implementation Design section of the SSAD, along with extensive help from a SCRover system architect.

Using this information, we were able to partially model the system in Mae. Additional work was required in converting the pre- and post-conditions (as specified in the Implementation Design section of the SSAD) into first-order logic expressions required by Mae. The extra effort was due to two obstacles: (1) These conditions were specified in English prose and often lacked proper connection to a component's state. Consequently a great portion of the Mae team's effort focused on determining and formalizing these conditions, such that they could be analyzed by Mae; (2) The Operations specified in the SSAD failed to identify the signatures (interface types) to which a given interface element belongs. Identifying these signatures required intimate knowledge of the MDS stylistic rules that dictate specific interactions among components and connectors. This also was performed in close collaboration with a SCRover system architect.

[3] The details of MDS Types and MDS Implementation schemas may not be described in this paper as they may be subject to the U.S. International Traffic in Arms Regulation (ITAR).

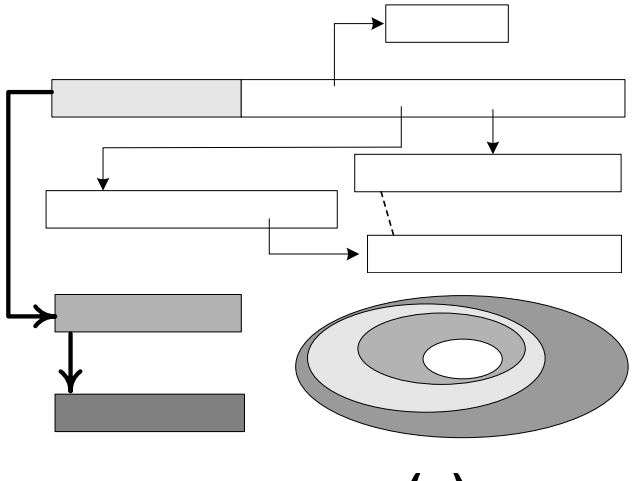

Figure 5. Sample schema structure (a) and Layers of schematic extensions (b)

3.3.2 Mae View of SCRover

Following the above process, we constructed a complete Mae model of SCRover. For illustration, a portion of the specification of SCRover's *PositionHeadingStateVar* component type obtained using the above process and built based on the Mae's xADL schemas is shown below.[4]

```
ComponentType:
MDS.SCRover.PositionHeadingStateVar
StateDecl       PHSV: PositionHeadingSVType
Invariant
Subtype         MDS.StateVar
Signature       (Prov) StateUpdate
   InterfaceElement    setPHStateVar
      InputParam       Var: PositionHeadingSVType
      OutputParam
   Operation
      PreCond          Var <> NULL
      PostCond         ~PHSV <> Var
Signature       (Prov) StateQuery
   InterfaceElement      getPHStateVar
      InputParam         Time: TimeTag
      OutputParam        PositionHeadingSVType
   Operation
      PreCond            (Time > 0) AND #PHSV > 0
      PostCond           result = PHSV
```

3.4. Summary of Approaches

Both Acme and Mae used the MBASE UML documentation as the basis for developing the two architectural models. The first phase of each approach was essentially the same, using class diagrams and associations to obtain a first cut of the SCRover architecture's topology. However, the subsequent phases differed. While the Acme team primarily used sequence diagrams to clarify the connections between components, the Mae team used the SSAD's Implementation Design to determine component services and behaviors.

The differences in these phases highlight a key difference in the two modeling approaches. The Acme team was mostly interested in checking whether the topology of the SCRover architecture conformed to the MDS rules. On the other hand, the Mae team assumed that the topology of the architecture conformed to the MDS style and was interested in checking whether component services were used correctly.

The differences in approach produced complementary models that were able to detect different kinds of errors. These errors are discussed in the next section.

4. Comparison

While UML provided a broad view of the SCRover's design and functionality, Acme and Mae specifically focused on architectural aspects of the system. Consequently, the classes of defects discovered by each approach differed. In analyzing these defect classes, we realized that there is a pattern to the kinds of defects each approach helps to reveal, resulting in a taxonomy depicted in Figure 6. The taxonomy focuses on classes of defects that are architectural in nature. It helps to clarify the respective strengths and weaknesses of each approach, and could serve as a basis of identifying other approaches that may help detect other types of defects.

At its top level, the taxonomy classifies architectural defects as *Topological errors* or *Behavioral inconsistencies*. Topological errors tend to be *global* to the architecture and concern aspects related to the configuration of components and connectors in the system. They are often a result of the violation of constraints imposed by architectural styles. Some topological errors are *directional* in nature: the specific direction of communication required by the style is violated. An example of this error is when in a Client-Server architecture the server requests a service from a client. Other topological errors are *structural* in nature and are further divided into *usage* violations and *incompleteness* of the specification. An example of a usage violation is when a communication link between two components is missing, or alternatively, when a communication link between components exists where it should not be present. Incompleteness manifests itself when there is insufficient information for specifying the properties of the architecture's components and connectors.

Behavioral inconsistencies relate to architectural information *local* to a component, or a connector servicing a set of interacting components. They concern mismatches between *dynamic behaviors*, or *protocol* of interacting components. An interface defect occurs when the signatures of the corresponding provided and

[4] In this notation, ~ denotes a new value for a variable, while # denotes the cardinality of a set.

required services of two components are mismatched. The static behavioral inconsistency reveals mismatches between the pre- and post-conditions of corresponding provided and required services in two components. The foundation for identifying these mismatches may be found in [13]. Finally, a protocol inconsistency reveals mismatched interaction protocols among components.

In the rest of this section, we use the above taxonomy to discuss the defects identified as a result of a peer-review of SCRover's MBASE-SSAD document, as well as the results of the automated analyses in Acme and Mae.

4.1. MBASE-UML Defect Detection

UML diagrams produced in the context of the MBASE process were effective in providing a high-level graphical view of the SCRover application that is reasonably easy to understand, although not always easy to *unambiguously* interpret. In addition to understanding the properties of the system under development, another goal of software modeling is ensuring correctness of the models, as well as consistency among them. The latter would help to detect errors early in the development process, thus reducing the development costs.

Unfortunately, the UML diagrams, along with the tables and English language prose that detailed the architecture and design of the system, cannot be automatically analyzed. Instead, a peer-review process of the artifacts proved helpful by identifying 38 defects in the documentation. The peer-review was carefully conducted by the system architects, and had to be repeated after each major change to the design. The nature of 38 discovered defects varied from English language problems and typographical errors, to sophisticated errors that could potentially cause harmful behaviors; some of them were architectural in nature, while others were conceptual. These errors were further classified and documented [2][20].[5] Even though peer-reviews are an effective way to identify defects, our initial hypothesis was that automated support can help to make this process even more effective. In the next two sections, we describe how Acme and Mae detected architectural defects beyond those identified by the peer-review process.

4.2. Acme Defect Detection

The developers of SCRover used the MDS implementation framework, in which type checking can be used to check for conformance to some of the MDS rules. They operated with incomplete knowledge of the informal English rules that were given to the Acme team by JPL to define the Acme MDS style. Instead, they had to rely on JPL personnel with knowledge of the MDS style rules to uncover architectural errors in design reviews. On the other hand, once the Acme team codified the rules, the Acme toolset was able to automatically check architectures for violations of these rules.

Following the process outlined in Section 4.2, the Acme team developed a full Acme architectural model of SCRover. This architectural model was created using AcmeStudio, an architecture development environment that allows an architect to draw an architectural model, and utilizes architectural styles to provide the architect "templates" for element types. Additionally, AcmeStudio provides incremental checking of the adherence of an architectural model to style rules via a constraint analysis engine. AcmeStudio also type checks the design to ensure that the system correctly uses the component and connector types in the MDS style. Moreover, AcmeStudio prevents certain errors occurring by construction. For example, if it is specified that component type A can only have ports of types B and C, then AcmeStudio will not allow the architect to add ports of other types to components of type A.

Using AcmeStudio we discovered 11 defects in the SCRover model derived from the SSAD documentation, of which 6 were new defects not previously detected by the UML peer-review. An example defect detected by AcmeStudio is the *PositionHeadingSensor* component querying the *PositionHeadingEstimator* component for a measurement; the communication direction as specified in the SSAD violated MDS stylistic rules.

It is worth mentioning that the total effort expended on adapting Acme to MDS and extracting the SCRover model was relatively small: roughly 120 person-hours, 80 hours of which was used to develop the architectural style, 30 hours to transform the SCRover UML documentation to an architectural model in that style, and 10 hours to tailor the environment, model the system, and conduct the analysis.

4.3. Mae Defect Detection

The specialized MDS schemas discussed in Section 3.3 were used to create a complete model of SCRover. To do that, xADL 2.0's accompanying toolset [5] was used to automatically reconfigure the API needed to manipulate architectures adhering to these schemas. In turn, this resulted in the automatic adaptation of Mae's design subsystem. Furthermore, Mae's analysis subsystem was manually adapted to correspond to the updated schemas. Consequently, Mae supports analysis of static behavioral properties of a system in the MDS style.

To verify the consistency of an architectural configuration, Mae checks whether the interfaces of the corresponding provided and required services of communicating components match, and then it does so for the com-

[5] Out of the 38 original UML defects, only 24 were architectural in nature and thus relevant to the remainder of our discussion.

ponent's static behaviors (specified in first-order logic) [12][22]. Finally, Mae verifies whether the subtyping relations specified between the components within the architectural configuration hold.

Using the Mae environment, we were able to identify 21 inconsistencies in the SCRover architecture, of which 6 were new defects not previously detected by the UML peer-review. The inconsistencies were the result of mismatched signatures and pre- and post-conditions of components' services. An example mismatch is an error in the specifications of the *PositionHeadingController* and *PositionHeadingStateVar* component types. The *PositionHeadingController* requires the *getPHStateVar* service that is provided by *PositionHeadingStateVar*. The provided service's pre-condition was more restricted than the pre-condition of the required service, and thus the required service may not have been satisfied under certain circumstances.

As in the case of Acme, the effort spent on adapting Mae to MDS was light. The total effort was roughly 160 person-hours, of which about 50 hours was spent on adapting the tool to model MDS architectures, 80 hours on extracting the Mae models out of the UML specification, and the remaining 30 hours was spent on building the model, using the tool, and performing the analyses.

4.4. Summary

We used the taxonomy depicted in Figure 6 to classify the defects detected by each of the three approaches. The results reinforced the hypothesis of the benefits of multi-view modeling. Figure 7 depicts the number and type of defects in each category. We also note the following:

- Peer-reviews of the UML design diagrams revealed both structural and behavioral inconsistencies. In particular, they revealed directional errors as well as mismatches in interfaces and pre- and post-conditions (shadowed boxes in Figure 6).
- AcmeStudio primarily detected structural errors in all three categories of directional, usage, and incomplete specification (light shading in Figure 6).
- Mae detected behavioral inconsistencies at the level of interfaces and static behaviors (dark shading in Figure 6).
- Of particular interest is the observation that both Acme and Mae revealed errors previously undetected using UML modeling and peer-reviews. Additionally, Acme and Mae detected different classes of errors, emphasizing the complementary nature of the approaches and associated analyses.

The three approaches, which were applied independently by three different research groups, not only confirmed each other's analysis results, but also demonstrated the value of viewing SCRover (and MDS) from different perspectives. Mae and Acme in tandem detected all architectural defects identified by the peer-review of UML models, and additionally identified previously undiscovered defects. UML peer-reviews, on the other hand, identified additional classes of defects that were not architectural in nature [2]. Finally, taking the effort data into consideration, once the initial style characterization and MDS adaptation of Mae and Acme were complete, the effort and expertise required to detect errors using the automated tools became significantly lower than what is needed for a peer-review of design documents. This benefit would particularly manifest itself in modeling future MDS-based systems.

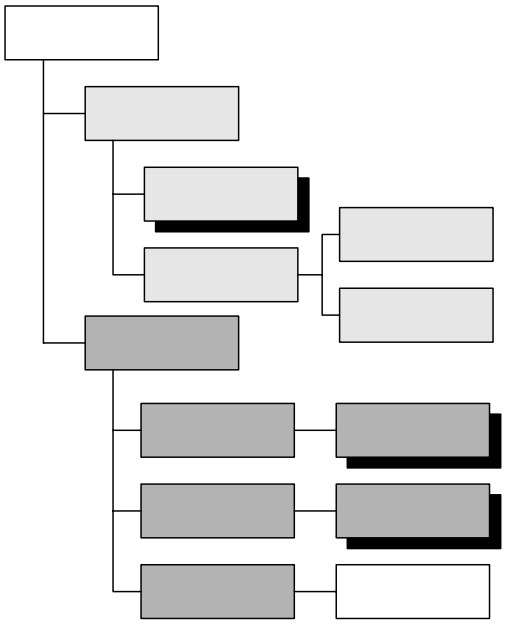

Figure 6. Classification of architectural defects

5. Lessons Learned

The work described in this paper was valuable in that it enabled us to apply our respective technologies to a "real world" problem. There are, however, some general lessons that make this experience more broadly applicable, beyond our two research groups. These lessons can be classified into two categories.

5.1. Multi-View Design

It has been accepted that it is both beneficial and necessary to design a software system from *multiple perspectives*. For example, this belief is reflected in the central role multiple views play in UML and the recent IEEE standard for architectural description (IEEE Std 1471-2000). Despite this, there is still a lack of specific details, or experience, regarding the choice and value of particular types of view, and their inter-relationships. Our experience provides two general lessons in this regard.

The first lesson deals with the complementary nature of UML and ADLs. While previous work, including our own, has looked at this issue from a more analytical perspective [10][12] we have now demonstrated in an actual project that it is possible, and indeed useful, to map from UML to architectures in a principled way. A related observation is that, while our previous work strived for automatable refinement solutions from UML to ADLs, UML's semi-formal nature (and often informal use) required a much more human-intensive refinement process in the SCRover project. UML's lack of formality and its typical use, which involves only a subset of the available diagrams (e.g., extensive use of class and sequence diagrams in SCRover), also make it harder to uncover architectural errors directly as compared to Acme and Mae. On the other hand, UML models system aspects that are closer to the implementation than is the case with the two ADL models. UML is, therefore, more beneficial to system implementors than are ADLs.

The second lesson deals with the complementary nature of different ADLs. Again, our previous work [14] has argued that different ADLs provide complementary capabilities, but this project has given us *practical* evidence to support this claim. Different ADLs can detect different kinds of architectural errors depending on the system aspects and properties they model. While one could choose an arbitrarily large number of ADLs to maximize system analysis, the pragmatics of a software project are likely to mandate that this be limited to a small number of notations that cover the largest cross-section of pertinent issues. Our experience with SCRover indicates that two natural candidate analyses enabled by ADLs include conformance to style rules and consistency of component interactions.

5.2. Architectural Design

The benefits of *formal architectural design* have been widely touted in research literature. At the same time, practitioners have often opted for less formal modeling solutions such as UML. Our experience provides evidence that a degree of formality is a "necessary evil," at least with respect to two types of architectural analysis: high-level behaviors and style invariants.

In both cases, formality of models enables automated analysis, which is essential for scalable error detection. For example, SCRover is a medium-sized project, consisting of around 30 components. But even for this size, basic architectural errors such as interface mismatches and improper connector usage were undetected in SCRover's UML models. Furthermore, the complexity and interplay of the set of architectural style rules (39 in the case of MDS) make manual checking infeasible.

On the other hand, an ADL's explicit focus on component signatures and interface semantics is a direct enabler

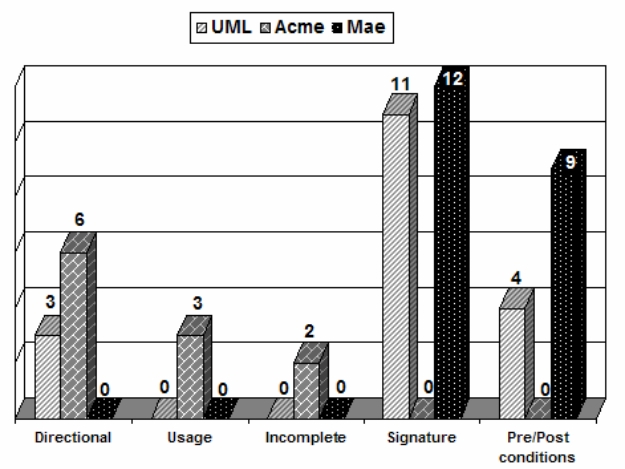

Figure 7. Defects detected by UML, Acme and Mae approach (by type and number)

of problem detection in cases where components do not agree (perhaps in subtle ways) on assumptions of interaction. Similarly, architectural style invariants permit one to check for satisfaction of structural (topological) constraints, missing or extra elements (such as missing connectors), and missing or inadequately specified system property values. In both cases, the formal models are much more amenable than informal models to early detection of errors and to tracing the sources of those errors.

6. Conclusion and Future Work

In this paper we have presented our experience in applying two architectural modeling techniques to software in the space domain. The experience showed that Acme and Mae identify different classes of errors than those found in a typical UML-based approach with frequent design reviews. Moreover, the experience confirmed that different ADLs find different classes of defects, and function in a complementary fashion. We also observed that the process of translating existing UML documentation into architectural models, while human-intensive, is similar in both cases. Although our experience is limited to a single software system, we believe that the approach taken and the lessons learned should apply to other software systems that use UML for documenting their design, and that have a rich architectural style.

This experience also indicates avenues for future work. First, and most obvious, is verification and generalization of the lessons learned. We plan to do this by conducting similar analyses of other JPL space software, and then later other domains in which the architectural style is also rich (for example, automotive software). We anticipate that the result of different case studies will be a more formal defect classification scheme, and a more rigorous mapping process between UML and architectural models.

The second area of future work is to develop a better understanding of the relationship between architectural modeling approaches, particularly between Mae and Acme. Not every feature of each approach was used in this experience, and so there may be more areas of overlap between these tools that need to be understood and reported. To facilitate this, we plan to explore integration opportunities between the two toolsets. This integration will also provide the opportunity to address consistency maintenance issues between the models.

Finally, our experience was restricted to two types of architectural modeling approaches, and two types of analyses. We would like to understand where other modeling approaches and analyses fit into this context (for example, some of those described in [14]).

Acknowledgements

This work was supported by NASA-HDCP contracts to CMU, JPL, and USC. It also benefited from significant support by JPL's Dan Dvorak, Kenny Meyer, Kirk Reinholtz, Nicolas Rouquette; and by USC's SCRover development team, headed by Barry Boehm. We also wish to acknowledge the cooperation and continuous help with xADL tools provided by the developers of xADL: Eric Dashofy, Andre van der Hoek, and Richard Taylor. This material is also based upon work supported by the National Science Foundation under Grant Number CCR-9985441.

References

[1] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Pra*ctice. Addison-Wesley, 1999.

[2] Boehm B., et. al., Using Testbeds to Accelerate Technology Maturity and Transition: The SCRover Experience, *USC Technical Report Number USC-CSE-2003-507*, 2003.

[3] Boehm B., and Port D., "Balancing Discipline and Flexibility with The Spiral Model and MBASE", Crosstalk, December 2001, pp. 23-28 (http://www.stsc.hill.at.mil/crosstalk).

[4] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Document Software Archtiectures: Views and Beyond.* Addison-Wesley, 2002.

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

[6] Dvorak, D. Challenging Encapsulation in the Design of High-Risk Control Systems. In *Proc. 2002 Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'92),* Seattle, WA, November 2002.

[7] Dvorak D, Rasmussen R., Reeves G., Sacks A., Software Architecture Themes in JPL's Mission Data System, *AIAA Space Technology Conference and Expo*, Albuquerque, NM, 1999.

[8] Garlan, D., Allen, R.J., and Ockerbloom, J. Exploiting Style in Architectural Design. *Proc. Symposium on the Foundations of Software Engineering*, New Orleans, LA, 1994.

[9] Garlan, D., Monroe, R.T., and Wile, D. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*. Leavens, G.T., and Sitaraman, M. (eds). Cambridge University Press, 2000 pp. 47-68.

[10] Garlan, D., Kompanek, A., and Cheng, S.-W.,Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computer Programming* Volume 44, Elsevier Press, pp. 23-49.

[11] Kruchten, P.B. The 4+1 View Model of Architecture. IEEE Software, 2(6):42-50, 1995.

[12] Medvidovic N., Rosenblum D.S., Robbins J.E., and Redmiles D.F., Modeling Software Architectures in the Unified Modeling Language, *ACM Transactions on Software Engineering and Methodology*, January 2002.

[13] Medvidovic N., Rosenblum D.S., and Taylor R.N., A Language and Environment for Architecture-Based Software Development and Evolution, In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99),* Los Angeles, CA, May 1999

[14] Medvidovic N., and Taylor R.N., A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), pp. 70–93, 2000.

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

[16] NASA High Dependability Computing Project (HDCP) http://www.hdcp.org/.

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

[18] Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N., Mae - A System Model and Environment for Managing Architectural Evolution, Submitted to ACM Transactions on Software Engineering and Methodology (In review), 2002.

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

[20] USC Center for Software Engineering, SCRover Increment-I documentation package, http://matador.usc.edu:8888/export_package/wall-following_scenario, 2003.

[21] USC Center for Software Engineering, *Guidelines for Model-Based (System) Architecting and Software Engineering*, http://sunset.usc.edu/research/MBASE, 2003.

[22] van de Hoek A., Rakic M., Roshandel R. , Medvidovic N., Taming Architecture Evolution, in Proceedings of the 6th European Software Engineering Conference (ESEC) and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, 2001.