IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4526

Aspect-Oriented Architectural Analysis using Multi-level Modeling of Complex Systems

Phillip Schmidt, Robert Duvall, Greg Mulert, Jaime Milstein, and Jesus Rivera
The Aerospace Corporation[1], 2350 E. El Segundo Blvd
El Segundo CA 90245-4691
Phillip.P.Schmidt@aero.org

1. INTRODUCTION

The Real-time Architecture-Centric Testbed (REACT), that we developed, adopts a new architecture-centric, early-discovery approach to analyzing and modeling architecture designs prior to code development. Contractor-provided architecture artifacts are typically Unified Modeling Language (UML) class, sequence, and state diagrams, but other non-UML data (e.g. document-oriented usecase descriptions, spreadsheet models, design studies, task configurations etc.) may also be provided. Lessons learned and an example simulation using REACT's early design is discussed in [1]. The early design of REACT used specially developed tags and primitives to augment additional architecture information via a commercial UML tool or a REACT data entry Graphical User Interface (GUI) (e.g. platform specific information). We found during spiral development cycles that the practice of extensive manual external augmentation of UML diagrams did not scale when large UML models consisting of thousands of classes and methods would change. Lessons learned in using REACT's aspect-oriented approach are discussed in [2]. REACT's architecture-centric approach differs from related research that has focused on developing automated techniques to support architecture synthesis and object-oriented software development (e.g. code generation). Automated techniques to synthesize state machines from OMT scenario diagrams are discussed in [3],[4],[5]. In [6], a UML statechart synthesis technique from collaboration diagrams is applied to produce more complete behavioral specifications. The DYNAMO environment [7] promotes deriving static architectural information from dynamic scenario models. Some commercial UML tools provide proprietary development methodologies to auto-generate state-driven simulation and development support for real-time systems [8],[9]. Although REACT supports automated synthesis of state/activity diagrams from sequence diagrams and animation during simulation, REACT's focuses on using these techniques to perform automated aspect-oriented static and dynamic analysis of proposed architectural descriptions to discover and remedy architectural shortfalls early. This paper presents some architectural challenges we frequently encounter when analyzing complex systems. We briefly highlight REACT's new aspect-oriented design approach and how we exploited a multi-level modeling approach to address UML modeling shortfalls that we encountered when performing architectural analysis of large satellite communication systems.

2. ARCHITECTURAL CHALLENGES FOR COMPLEX SYSTEMS

The long-term modeling goal for REACT is to build an architecture assessment capability that accurately reflects the embedded system software architecture of the satellite systems under study. REACT is used to exercise various workload scenarios to perform early discovery of possible flaws in design and understand better the bounds of correct operation. Scenarios of interest may be suggested by the contractor or by REACT. When code is available, REACT will reverse engineer that and compare it to the UML design, and use sizing parameterizations to refine earlier assessment models. There are several problems of interest to REACT. These include:

Model key input/output queues etc. and monitor their size

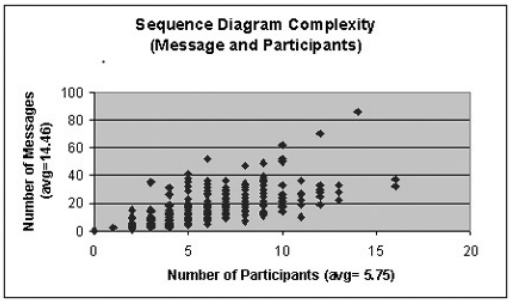
- Investigate the profile of method executions (given that granularity) for different time-critical scenarios
- Investigate possible effects of task starvation from tasking/priority structure
- Investigate bandwidth utilizations of various busses due to software messaging
- Backward compatibility issues with legacy communication software
- Multi-satellite communication and synchronization/control/timeout issues
- Error events and recovery scenarios
- Estimates of memory and CPU utilization over time

Upon receipt of UML, we often discover a disconnect between the level of architectural detail provided by the contractor and what is necessary to fully address the interest areas stated above. For large, highly complex systems, there is a trade-off between what insight REACT hopes to get out of a UML model, and what it takes for a contractor to provide the detail to get that insight. For example, in one project we found 6,071 methods used in all the UML class diagrams. 56% of those methods did not have any behavioral detail described within either a state/activity diagram or a sequence diagram. To uniquely characterize all of a method's behavior via a sequence diagram, a method's behavior could be documented via a principal participant in one sequence diagram.[1] In the project under study, this would require 6071 sequence diagrams. However, in the project, there were only 410 sequence diagrams. Consequently a principal participant sequence diagram description would document only 7% of all methods. Even if the 6071 sequence diagrams were provided, the program complexity is actually much greater than what a human can understand by manually inspecting the architecture. To see this, we created a REACT aspect to generate the number of messages versus the number of sequence diagram participants for the 410 sequence diagrams as shown in Figure 1.

The average (documented) sequence diagram has 14.46 messages and involves 5.75 participants. This means that on the average there are between 2 and 3 messages per participant on each sequence diagram, indicating that the number of action-groups per methods would be between 2 and 3. For 6,000 methods, this means between 12,000 to 18,000 action-groups would be needed by a human to "understand" the design behavior at the method level. Understanding the implications of change to the methods of interaction for such a complex system cannot be managed manually. Techniques to automate the modeling of complex systems are essential to not only identify architectural shortfalls, but also to close the discrepancy between what behavior is provided and what behavior is needed to model the system.

There are at least three approaches to support modeling of the contractor architectural artifacts. These are method-level, participant

Figure 1: Sequence Diagram Complexity



level, and use-case level modeling. The next section briefly describes REACT's new aspect-oriented design and section 5 discusses REACT's approach to mitigate the shortfalls of the UML architectural descriptions.

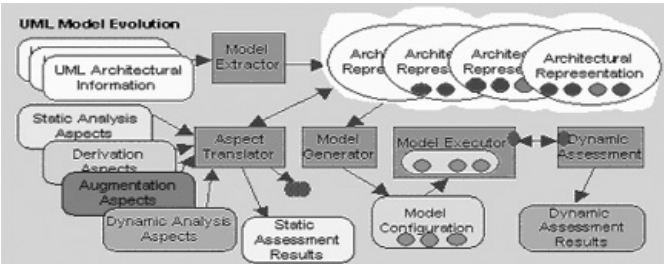
3. REACT'S ASPECT-ORIENTED DESIGN

REACT's architectural analysis design is patterned after an aspect-oriented approach [10]. An aspect defines an area of concern and usually provides some action to take regarding that concern. Aspect-orientation is usually described in terms of defining aspects over the programming language space. In REACT however, the definition of aspects is performed over the architectural design space. In this way, REACT supports the ability to separate concerns but is non-architecturally intrusive since its role is to passively evaluate a contractor-given architecture, not redesign it. REACT's aspects are principally ones of discovery, derivation, augmentation, and dynamic assessment, not necessarily redesign (though these, too, could be developed). For example, a discovery aspect (usually expressed in XML) might look for method names in a sequence diagram that are undefined within a class. A derivation aspect might collect all events. An augmentation might provide additional meta-information such as parameterizing task priority information. REACT's aspects can be very sophisticated as they effectively allow query and augmentation over the architectural design space. A partial list of REACT's aspects include:

- All classes with duplicated class names but different attributes or method definitions
- Special tags used within sequence diagrams to denote requirement traceability information
- All sequence diagram participants
- Documentation information
- Class methods that do not appear in any sequence diagram
- All guard conditions used in backward self-message
- Sequence diagrams for which no class is defined
- Sequence diagrams that have a participant/object of a particular class

We found that REACT's aspect-oriented architectural assessment is an effective approach for static and dynamic analysis. Figure 2 illustrates REACT's new aspect-oriented design. REACT accepts architectural information principally in the form of UML and extracts this information into an architectural representation. If this representation contains behavioral information (such as state/activity diagrams, or sequence diagram information), a Model Generator can generate model configuration files that capture the logical behavior of the UML design.

Figure 2: REACT's Aspect-Oriented Architectural Assessment



These model configuration files can be interpreted during simulation that can support dynamic assessment. The activity diagrams are directly derived from the corresponding UML models prior to any code development. REACT supports extensive static analysis of UML designs. Although many UML tools provide built-in static checks, and provide scripting and programming capabilities to enable the tool user to program particular static analyses, this approach is tool-specific. Since our analysis requires that we accept UML from possibly any vendor, REACT adopts a UML tool-independent approach to static analysis. This has proven to be quite useful in identifying architectural inconsistencies and incompleteness especially in early architectural development when complete behavior may not be known. Manual inspection of the UML often leads us to design custom "aspects" for further investigation. For example, one contractor used a particular UML icon (the destructor) in a non-standard way. Using aspects, we were quickly able to identify the exact location of all 239 instances after accessing over 400 sequence diagrams. In another example, REACT was able to identify discrepancies between UML and interface control document interfaces. In the following sections, we discuss how REACT's aspect-oriented approach has been applied to support dynamic assessment when UML models contain missing and incomplete information.

4. MODELING LEVELS

Method-level Modeling

Method-level modeling captures the behavioral detail of all methods used within the software so that timing and behavioral studies can be performed. It is the lowest level of granularity and offers the greatest modeling fidelity. Modeling at this level enables behavioral replacement with reverse-engineered code, and can study other integration issues such as legacy interface documents, and potential multi-satellite cross-link behavior. This level supports fine-granularity studies involving ground-system interfaces. As discussed above, to be successful in the long-term, method-level modeling often requires more behavioral detail than a contractor usually provides. As stated earlier, for the satellite project under study, only 44% of the methods were described anywhere, and at most only 7% as a principal participant.

Participant-level Modeling

Participant-level modeling deals with the interactions at the sequence diagram participant level. Normally each participant within a sequence diagram would have a state/activity diagram to characterize its behavior, and the participant is usually an object instance of some class. When standard sequence diagrams are used, participant level modeling at this level provides only logical flow with no control flow information. When sequence diagrams support control flow, then translated state/activity diagrams can be generated. Modeling at this level generally is class-oriented and of greater granularity. For large complex systems, UML sequence diagram participants can be incomplete. In one satellite project, all but 8 participants on the sequence diagrams represented class instances (objects). Of the 2,300 participants, only 866 were unique, which means that only 75% of the classes were captured at this level.

Use Case-level Modeling

Use case-level modeling attempts to capture behavior described by collections of use case steps (called courses of action (COAs)). Use case level modeling permits the testing of pre/post conditions and will provide evaluation of the processing behavior of those "threads of activity" defined by use cases. Within the projects we have studied, the software use cases are usually described outside of UML as text documents. The COAs represent processing actions and decision point actions. The use cases include pre and post conditions as well as alternative actions and triggers that may drive a particular use case. Use case

modeling maps closely to requirements-level processing but may not directly trace to actions that are within sequence diagrams. One lesson we have learned is that COAs traced to sequence diagram level granularity instead of method-level granularity make it difficult to map the many COA actions to method level behavior. We have found that COA traceability varies widely between the different software components being developed. Some software developers are more thorough than others.

5. APPLYING MULTI-LEVEL MODELING

Given the above limitations in method behavioral descriptions, participant modeling, and use case coverage, what is REACT's approach to meet it's modeling goals? In this section, we describe how REACT's aspect-oriented approach is being used to support multi-level modeling.

Method-level Modeling

REACT has developed a capability to automatically translate UML sequence diagrams (that contain vendor-specific control information) into state/activity diagram representations within REACT's internal representation. The result of this automated translation is a large number of condition and action-list items that need to be parameterized in order that they can be simulated. Techniques to do the parameterization automatically are being developed. What this entails is a separation of model-specific information (e.g. how should a particular decision be modeled) from architectural representation (e.g. there is a decision id 45 on activity diagram id 34), and techniques to provide a mapping assignment (e.g. model decision id 45 on activity id 34 using the provided probability distribution.) Method-level modeling will be performed by searching over the sequence diagrams looking for a principal participant that is invoked by the method. The above analysis indicates that the coverage of method behavior via this technique will be fewer than 10% of all methods. For methods not discovered this way, REACT auto generates a default, black box, generic-processing behavior that will likely lose processing detail because it will be a crude representation of actual behavior, but will provide 100% completion of behavioral detail. REACT is developing aspects to automatically parameterize behavioral information based on default primitive behavior or by importing prior augmented architectural information. For example, although initially missing architectural detail may be a simple processing block, later UML models may provide greater detail, or even behavior extracted from reverse-engineered prototype code could be applied.

Techniques to support automatic augmentation via aspects are being developed. Since REACT's internal representation is XML-based as well as the separate model-specific information files, it is not difficult to define aspects that navigate over REACT's internal representation looking for similar architectural matches. In contract, in REACT's old design, locations for architectural matches would be described as embedded tags that were inserted via manual methods. As different UML models were frequently released (biweekly) such a manual-intensive augmentation approach quickly became infeasible. REACT's new aspect-oriented approach enables modeling-specific information to be designed, kept and to evolve separately from REACT's architectural representation. Aspects processing is then provided that take pre-defined parameterization data in a model-specific information file and map it to an action location within REACT's representation using XML queries and navigation over REACT's architectural space.

Since not all sequence diagrams follow a principal participant representation approach, it is possible that any given method invoked in different contexts, will display different behavior. To support this, REACT will develop techniques to extract and manage the multiple behaviors for methods invoked within sequence diagrams. For example, aspects to locate the different representations will be developed. Merging these multiple behaviors into a common activity diagram is an area for further research.

Method behavior may be replaced when code is reverse-engineered, but behavior detail will be limited by the detail of reverse-engineering techniques. Techniques to automatically replace reverse-engineered method behavior will be explored. Tailoring commercial reverse-engineering tools is another area of further research. It is always recommended that the contractor improve the behavioral detail of their class methods directly, but if this is unlikely or not possible, early code development should be made available for reverse-engineering and then REACT's aspect-oriented augmentation can be applied. Finally aspects to search for design refactoring possibilities are also being studied.

Participant-level Modeling

To support participant-level modeling, REACT's auto-translation of sequence diagrams will be retargeted to generate state/activity descriptions for every participant within each sequence diagram. This is a different type of auto-generation than method-level modeling and results in a state/activity diagram associated with a participant instead of a method. A sequence diagram thread can be executed via an invocation stimulus to a participant at the beginning of the sequence diagram. In practice, making the auto-generated sequence diagrams executable usually takes some manual adjustment because not all the decision points, guard conditions, iteration criteria, etc. with contractor sequence diagrams are unambiguous. This is also true of method-level behavior.

REACT also supports the ability to connect the processing of different sequence diagrams together. REACT was able to identify a logical flaw in a contractor design by simulating the execution of the activity diagrams extracted from the sequence diagrams. The flaw was discovered because a necessary pre-condition before a method invocation was not satisfied. Although UML tools generally provide support object instance identification, UML descriptions frequently do not contain object instance identifiers to facilitate the chaining together of different sequence diagrams. It is important to know that an object instance on one sequence diagram represents the same instance on another sequence diagram. REACT's model generation process lives with this incompleteness (by issuing complaint warnings), but it is always better to insist the UML developer directly provide these distinctions.

Use case-level Modeling

Modifications to REACT's internal representation are being defined to incorporate use case information that is not directly available from UML. We implemented routines to parse the use case description documents and generated several types of XML files that capture pre/post conditions, flow diagram information, and triggering events. Some manual interpretation of this data will be needed because the semantics of natural wording is not often clear. In such instances recommendations for clarifying/correcting the use cases are made to the contractor to minimize manual intervention. The ability to manage the evaluation of pre and post conditions and support trigger and conditionals has been developed. Our early prototype of this information created a lightweight flow diagram that was significantly different than our more comprehensive representation of state/activity diagrams. In particular, one problem we are working is the ability to auto-generate geometry information for the auto-generated flow diagrams. Our provided use cases, of course, do not have such information, but it turns out that such information is quite valuable during simulation to support animation. Each processing COA step will be identifiable via this translation. The result of this automated translation will be a large number of condition and action-list items needing parameterization similar to method-level modeling. Techniques to automate this are being implemented, though some manual intervention to identify natural text expressions for equivalent semantics will be needed. Once this intervention is complete however, it is possible to develop techniques to automatically reapply these equivalences in later versions of the use cases. Later extensions to this work could explore techniques to automatically suggest/perform mappings of use case behaviors to method level behaviors, but this is currently quite difficult because this mapping information is not available from the contractor. This problem can be avoided by mapping COAs to method level behavior.

Other REACT Modeling

Direct state/activity-level modeling is of course possible. When state/activity diagrams are present, REACT's model generation can pro-

cess them. REACT is continuing to develop its internal representation to support monitors and is developing techniques to support dynamic reception/interpretation of model information. REACT will also pursue advanced static analysis activities (deadlock detection, reachability, etc.) and collect performance-related information derived from contractor artifacts (e.g. traffic model simulation).

6. SUMMARY

We have shown how REACT's aspect-oriented architectural assessment approach is being applied to support architectural informational shortfalls frequently encountered in modeling large complex systems. The magnitude of the number of classes and methods and their complex interactions requires automated techniques to understand their behavior and identify architectural flaws early. With spiral development cycles requiring frequent revisions to UML architectures, it is necessary to provide flexible techniques that can perform analysis over evolving architectural designs. We have shown how missing behavioral information can be identified and provided with default behavior that can later be refined automatically. With REACT's aspect-oriented approach, aspects can be written to identify architectural sites of interest and either modify, augment or map modeling information so that static and dynamic assessments can be made. REACT's automated translation and simulation of sequence diagrams permits different levels of modeling granularity. REACT's internal management of this model information minimizes the amount of manual intervention required and permits the reuse of prior data over those portions of the architecture that have not changed.

ENDNOTES

[1] In this approach, usually a generic actor is used to invoke the method. The method invocation of course appears as a message from the actor to an object instance of the class containing the method. This object instance is called the principal participant because all of its interactions from the point of invocation describe the behavior of the method. Depending on the UML tool used, different levels of behavior can be captured. Some UML tool vendors permit embedding control logic within a sequence diagram.

[2] © 2002 The Aerospace Corporation

REFERENCES

[1] Schmidt, Phillip, Milstein, Jaime, Duvall, Robert, Lankford, Jeffrey, Rivera, Jesus, "Lessons Learned Using REACT: An Architectural Testbed for Real-time Embedded Systems," in *Proceedings of the 2002 Information Resources Management Association International Conference*, Seattle WA, May 19-22, 2002.

[2] Schmidt, Phillip, Duvall, Robert, J. Lankford, G. Mulert, "Evaluation of Aspects in UML Models," in *Proceedings of the Ground System Architectures Workshop*, March 13-15, 2002, The Aerospace Corporation, El Segundo, CA 90245. See http://sunset.usc.edu/gsaw

[3] Koskimies, Kai, Mannisto, Tatu, Systa, Tarja, Tuomi, Jyrki, "SCED: A tool for Dynamic Modelling of Object Systems," University of Tampere, Department of Computer Science, Report A-1996-4, 1996.

[4] Koskimies, Kai, Systa, Tarja, Tuomi, Jyrki, Mannisto, Tatu, "Automatic support for modeling OO software," *IEEE Software*, Vol 15, Number 1 (1998) 42-50

[5] Koskimies, Kai, Makinen, Erkki, "Automatic synthesis of state machines from trace diagrams." *Software – Practice and Experience*, 24(7):643-658, July 1994.

[6] Khriss, Ismail, Elkoutbi, Mohammed, Keller, Rudolf, "Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams," in *Lecture Notes in Computer Science 1618*, edited by Bezivin, Jean and Muller, Pierre-Alain, The Unified Modeling Language, UML 98: Beyond the Notation, p132-147, Selected Papers, First International Workshop, Mulhouse, France, June 1998, Springer.

[7] Normark, Kurt, "Deriving Classes from Scenarios in Object-oriented Design, May 1997, See http://www.cs.auc.dk/~normark/dynamo.html

[8] Telelogic Tau UML/SDL Suite, www.telelogic.com

[9] I-Logix Rhapsody UML, www.ilogix.com

[10] See *Communication of the ACM*, October 2001, Vol. 44, No. 10.