IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITP4300

Lessons Learned Using REACT: An Architectural Evaluation Testbed for Real-Time Embedded Systems

Phillip Schmidt, Jaime Mistein, Robert Duvall, Jeffrey Lankford and Jesus Rivera
The Aerospace Corporation[1], California

INTRODUCTION

Recent costly space launch failures and difficulties with on-orbit operation have highlighted the fact that space system architecture designs are becoming increasingly complex to analyze. This complexity stems from the desire to increase program functionality, improve current performance, and seek greater program success within tighter cost/schedule constraints. Additionally, embedded spacecraft software architectures often require support for complex single and multi-satellite protocols, may utilize object-oriented designs and technologies whose performance and maintenance costs may be uncertain, and require the use of custom hardware, often only available after the software has been architected. This increased complexity creates significant program risk. The current procedure of manual inspection of hardcopy Unified Modeling Language (UML) software architecture designs is ineffective and inefficient in finding subtle design flaws. This is true for large ground systems as well as embedded system architectures where it is increasingly difficult to make technical tradeoff decisions based solely on qualitative judgments by integrated product teams. The current post-design, code-centric testing approach to problem resolution is also costly. Little or no coordination between architectural analysts and software evaluators involved in independent readiness reviews is practiced today. The result has been an increased risk in flawed/incomplete architectural designs leading to flawed/incompatible implementations, and possible "sleeper" design flaws that result in costly on-board failures. The Real-time Embedded Architecture-Centric Testbed (REACT) facility was created to reduce program risk by early identification and resolution of software architectural shortfalls.

This paper discusses the design goals and some lessons learned from using our REACT facility. Section 2 describes some of the challenges that we faced in developing a REACT environment. Section 3 describes REACT's architectural framework, and the technical approaches to address those challenges. Section 4 provides an example of how REACT has been used to perform architectural analysis. An actual satellite communication protocol model developed in UML is used with a multi-satellite, multi-terminal communication model. To illustrate REACT's capabilities, a subtle design flaw in the UML was introduced. This error would have been extremely difficult to find from manual inspection of UML output. The paper describes REACT's capability to automatically extract architectural information, generate an executable model configuration file, execute the model, analyze model results, and trace the error back exactly to the particular part of the architecture in error. It should be noted that REACT analyzes UML architecture design information—not code, illustrating REACT's goal of providing early discovery insight prior to code development. Section 5 summarizes some of the lessons learned and future directions for REACT.

REACT CHALLENGES

We faced many challenges when designing REACT:

Contractor-driven Architecture Analysis. REACT's purpose is to perform architectural system engineering analysis for complex, real-time, embedded satellite control systems. Our problem is unique from UML's mainstream usage, because our role is independent analysis of contractor-provided architectural designs, not software development. Many of the commercial UML tools sold today focus on developing UML diagrams for use by software developers, not systems engineers involved in analyzing proposed architectures. Some real-time commercial UML tools such as Artisan, and Rhapsody support auto-code generation and simulation support if vendor-specific development methodologies are followed, but our experience has been that contractors select their own UML tools and generally have been unwilling to relinquish code/design control or commit to proprietary architecture practices. REACT's initial design challenge is to design an open, analysis architecture that would be as UML-vendor independent as possible, yet be able to support contractor-provided architectures that often used UML in different ways.

Early Discovery of Architectural Shortfalls. The REACT environment differs from a classical test-bed approach that debugs a current, fully specified "end" design. REACT adopts an architecture-centric, early discovery approach to analyzing and modeling architectural designs prior to code development. REACT assesses how new, often partially specified, improvements impact a contractor's proposed software architectural design or legacy architectures already in use. Examples of REACT activities include: assessing proposed real-time embedded satellite control architectures, performance impacts of using demand assignment multiple access communication resource assignments with highly constrained embedded processors, investigating software architectural revisions that improve the allocation of configurable resources, investigating ways to improve dynamic reconfigurability through the use of component-based architectural design, and tracking OMG's on-going efforts of evolving UML for real-time. Contractor preliminary design specifications are often incomplete and immature making it difficult to understand the proposed software architecture. The benefits of proposed improvements are also often not well understood. Formal architectural models may not exist in electronic media. Legacy systems may have been developed prior to the architectural representational languages such as UML.

Management of architectural augmentation. Because not all architectural details are provided via UML, we needed to develop techniques to augment UML models with auxiliary information. For example, much of the platform specific information such as CPU processing speed, memory size, bus data rates, operating system context switch times, etc., is needed but does not change frequently. Oftentimes, providing this augmented information is labor intensive. We need to distinguish between contractor-provided information and specially provided augmentations. Additionally, to reduce program risk, large system development often follows a spiral development process model in which design and code are released in phases. To support the spiral development and the fact that architectural information could and would change over several phases, we needed to develop a strategy to recognize and appropriately reuse augmented architectural information.

Static Architectural Analysis. Once the architectural UML information is extracted, REACT can provide customized static architectural analysis to ensure the integrity of the model prior to generating an executable model file. This analysis can include syntactical

analysis based upon standard UML usage as well as contractor-specific adaptations. A collection of such semantic analyses could also be defined. For example, if a method raises an exception, there should be something that receives it. Improper semantic use of UML notation could also be identified.

Multi-grained modeling. A wide spectrum of modeling and analysis tools are needed that can start with incomplete specifications and permit rapid simulation and assessment. Initial assessment may utilize coarse-grained models at the component level, but must eventually support finer grained models that address object-level performance level threads. Since detailed state machines may not be available, the ability to auto-derive functional flow state machines from sequence diagrams is highly desirable.

Model Reparameterization. Initially some of the processing values for various parts of the UML model will be estimated from early contractor-provided data, legacy information, or system-engineering judgment. Early assessments of different workload models will need to be revisited as the architecture gains more fidelity. We recognize the importance of an improved relationship with independent readiness engineers that measure and evaluate delivered code. REACT's early analysis will identify potential hot spots (areas where performance may be in doubt) and soft spots (areas where preliminary analysis may need recalibration).

Configurable infrastructure for highly adaptive architectures. As we rely on real-time embedded systems that exercise more demanding processing functions within hostile space systems environments, there is an evolutionary trend to reduce the risk of mission failure by enabling the software architectures to be more responsive to unexpected circumstances. These architectures introduce new dynamic operational strategies that need to be evaluated and understood. Also contractors frequently adopt proprietary real-time operating systems with support tools that also utilize proprietary communication protocols to capture and analyze collected data. Often these tools operate standalone and do not have effective means to be rapidly configured to exercise multiple scenarios such as those involving distributed multiple satellite configurations. This environment makes it difficult to analyze the effectiveness of highly adaptive complex architectures in which internal state conditions determined at runtime can dynamically influence the exercise. For such an environment to be successful, a highly configurable support infrastructure is needed to capture critical real-time events and properly integrate them into a controlling simulation.

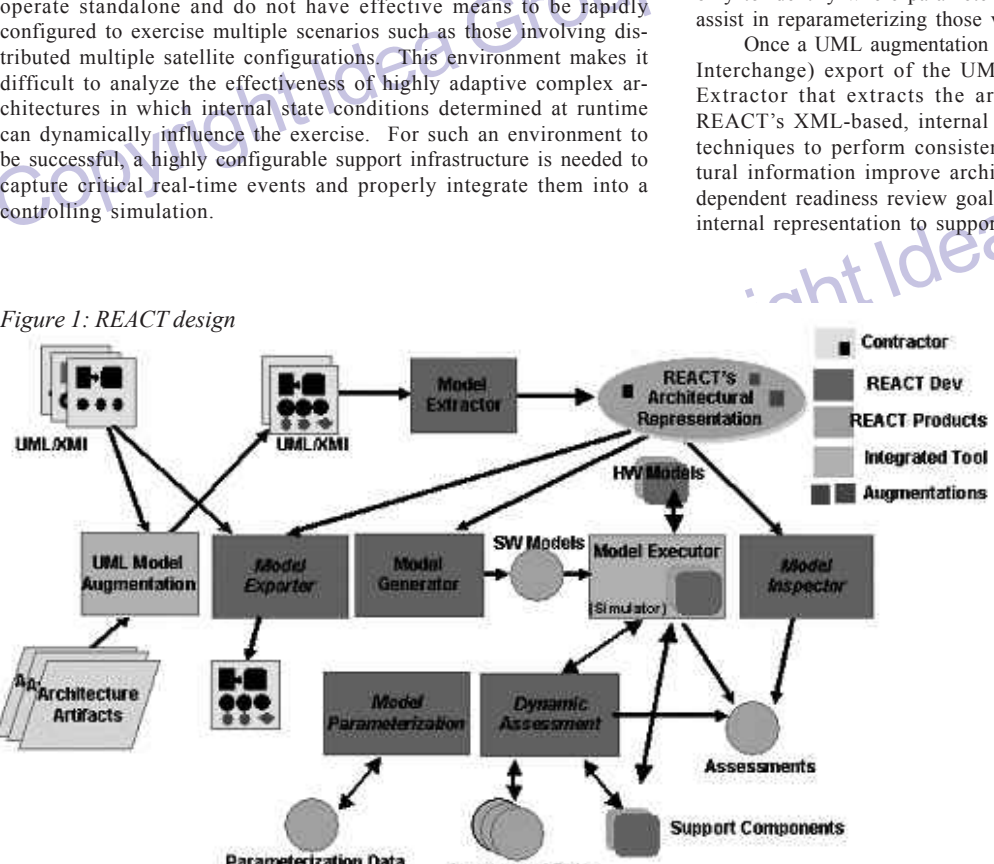
REACT DESIGN

REACT's design structure is illustrated in Figure 1. Components currently under development are shown in italics. REACT's components are entirely written in Java 1.2 and Swing support.

REACT adopts a new architecture-centric, early discovery approach to analyzing and modeling architecture designs prior to code development. Contractor-provided architecture artifacts are typically UML class, sequence, and state diagrams, but other non-UML data (e.g. spreadsheet models, design studies, task configurations etc.) may also be provided. We take the UML and using specially developed tags and primitives [2], augment any additional architecture information via a commercial UML tool or a REACT data entry Graphical User Interface (GUI) (e.g. platform specific information). For example, to characterize a timer event with a default timeout of 55ms and an event priority of 10, a special augmentation tag with value event would be defined that provides additional tags for event.type, event.priority, event.timeoutunits, and event.timeoutval to be defined and associated with the appropriate class diagram representing the event. It is important to know the heritage of the data represented internally within REACT. REACT manages this through the concepts of coloring and safety. REACT identifies (or colors) internal representation data in three ways. The representations are colored black, red, or green to describe original, augmentation, or original containing augmentation data respectively. Augmentations can be safe or unsafe. Unsafe augmentations change the original architecture intent. E.g. new operators added to classes, changing the operator signature, adding new attributes to a class, changing an operator's persistence type are all examples of unsafe changes. Unsafe augmentations may be intentional if it is desired to explore impacts to alternative architectural approaches. REACT's internal representation can always distinguish these differences. In addition, the architectural reference from which augmented data was obtained is also recorded. This is needed not only to identify where parameterized values were obtained, but also to assist in reparameterizing those values when recalibration is necessary.

Once a UML augmentation is completed, an XMI (XML Metadata Interchange) export of the UML model is provided to the REACT Extractor that extracts the architectural information and builds REACT's XML-based, internal representation. REACT's automated techniques to perform consistency and integrity checks of architectural information improve architectural confidence, an important independent readiness review goal. We are currently refining REACT's internal representation to support auto-generation of interaction-based state diagrams from sequence diagrams. The internal representation is used by the Model Generator to generate appropriate configuration files that are input to executable Java components (called actors) in the Model Executor. The Model Executor is Ptolemy, a Java simulation tool, developed at UC Berkeley. [1] Specialized Ptolemy actors were developed to model VxWorks task pre-emption and event management. Currently we use a pre-built Ptolemy model that represents the hardware architecture of our system, but as Ptolemy can also accept a specialized XML-formatted model, called MOML, it is possible to auto-generate a MOML file from augmentation data should it become necessary. Early architectural assessments that focus on critical execution paths are conducted to understand the logical execution behavior of the proposed system. The ability to identify inconsistencies, dead-

Figure 1: REACT design



locks, and architectural performance hot-spots in early design phases can guide software evaluators in testing whether explicit problems have been corrected and can suggest detailed parameter measurements (when code becomes available) to improve architectural models of later phases, or to reinvestigate prior analysis. The Model Inspector will exploit the ability to automatically compare as planned to as built architectures by comparing their different internal representations. The Model Exporter will permit REACT's internal representation (with augmentations) to be exported in an XMI format. Dynamic assessment will take a user-defined specification of what levels of accomplishment are to be evaluated and auto-generate the monitors and evaluations needed. Model Generation will recognize the monitor augmentation within the internal representation and auto-generate the appropriate configuration files. The ability to specify and execute dynamic assessment models is needed to understand how the architecture will function under different (possibly adaptive) scenarios. REACT's novel capabilities provide technical assurances for independent readiness review teams in areas that have not been well addressed.

REACT EXAMPLE

Figure 2 shows an example of how REACT has been used to perform architectural analysis. A multi-satellite, multi-terminal communication Ptolemy model is shown in the upper left corner. The model is hierarchical and contains several levels not shown. The terminal model will generate two specialized types of messages when the model representing the software architecture, communicates with the satellite. The satellite will interpret and route the messages to the appropriate destination. These message types are identified in the right side of Figure 2 by the message spikes of different heights. Graphs for terminal 1, satellite 1, terminal 2, and satellite 2 are shown. This model executes a selected subset of an actual satellite communication protocol. The communication protocol model was based upon UML, which is the principal way in which contractors provide architecture design information. An integrated product team (IPT) studying a proposed enhancement motivated this particular model. The IPT, however, did not perform any quantitative analysis and thought the enhancement would only require the satellite to "clean up some code." The UML model was based on interface requirements, specifications, and other architectural artifacts. To illustrate REACT's capabilities, a subtle design flaw in the UML was introduced simulating an inconsistent ground system contractor's algorithm with the satellite contractor's "clean-up" algorithm. This error would have been extremely difficult to find from manual inspection of UML output. UML models for the ground system and the satellite system were developed in Rose and exported into XMI. REACT's Extractor created the internal architectural representation and the Model Generator produced the configuration files. A sample portion of Model Generator's XML output for the satellite's UML is shown in Figure 3.

Figure 2: REACT analysis example

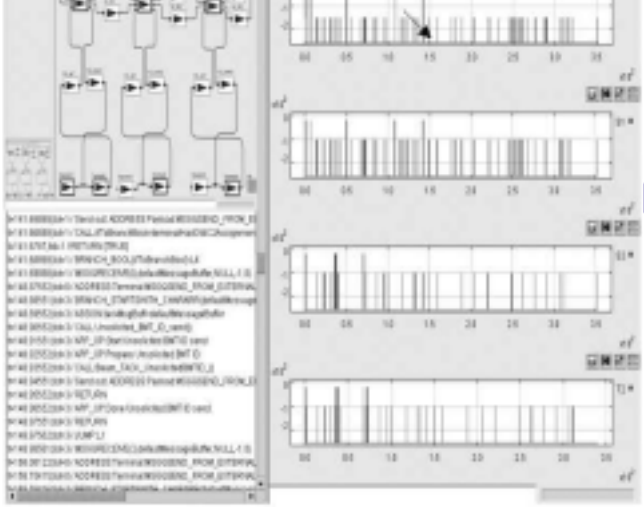


Figure 3: Sample model generator output

```
<?xml version="1.0"?>
<!-- Created by ModelGen, Wed Sep 19 16:46:47 PDT 2001-->
<modelinfo>
    <globalfunction>
        <name>Init_LITE_Payload_Payload</name>
        <localvar>
            <type>LITE_Payload_Payload</type>
            <name>object</name>
        </localvar>
        <activity>
            <runtime>0.01</runtime>
            <action>NEW:LITE_Payload_Payload():object</action>
        </activity>
        <activity>
            <runtime>0.01</runtime>
            <action>CALL:object.run()</action>
        </activity>
        <activity>
            <runtime>0.01</runtime>
            <action>EXIT</action>
        </activity>
    </globalfunction>
    <globalfunction>
        <name>Init_LITE_Payload_Beam</name>
        <localvar>
            <type>LITE_Payload_Beam</type>
            <name>object</name>
        </localvar>
        <activity>
            <runtime>0.01</runtime>
            <action>NEW:LITE_Payload_Beam():object</action>
        </activity>
        <activity>
            <runtime>0.01</runtime>
            <action>CALL:object.run()</action>
        </activity>
        <activity>
            <runtime>0.01</runtime>
            <action>EXIT</action>
        </activity>
    </globalfunction>
    <globalfunction>
```

When the model was executed, a deadlock occurred. This is illustrated in Figure 2 by the arrow. The lower left side of Figure 2 shows a detailed log file that is generated during the model execution. Each log line is directly traceable to the Rose model state machine. By inspecting the log file the actual discrepancies between the satellite and terminal algorithms were found—uncovering the cause of the deadlock. Inspection of the log files of the other terminals and satellites revealed a similar problem.

LESSONS LEARNED AND FUTURE DIRECTIONS

In developing REACT and using it to analyze UML architectures the following lessons were learned:

UML contractor usages and UML vendor implementations can vary significantly. REACT's early architectural assessments found that architectural designs were using UML in unconventional ways. For example, we found that ROSE permitted self-loop constructs on sequence diagrams to indicate local method calls to a sequence diagram object. One developer used these self-loops as a way to provide positional comments. Rose even permitted this. Such a practice is architecturally intrusive and should be avoided. Other UML implementation differences were due to weaknesses in the UML 1.3 specification with respect to associating notes in UML and managing the isActive property. In one case, a UML tool recognized the equivalence between collaboration and sequence diagrams, by providing the capability to auto-translate from one diagram to another. A side effect of this auto-translation is that it is not possible to support positional

notes because positional information within a sequence diagram participant is not tracked since it is "not used" in a collaboration view. In another case, language specific features such as Java's synchronized was noted in the XMI and accessible via special contextual displays but was not diagrammatically viewable. Simple support for tagged values also varied.

Tailored Semantic Architectural Analysis is effective in finding early errors prior to Model Generation. We initially developed some UML toy models for REACT to analyze. In one model we made a semantic error on a state diagram in which a timed event did not have an exit condition if the event were to timeout. We were able to semantically check for this condition in the Model Generator prior to attempting model execution, effectively making the Model Generator into an architectural compiler. The ability to customize the interpretation of specific UML usages is also highly desirable during Model Extraction as well, especially when UML is used in non-conventional ways. In one example, we found early users of UML technology often found that abstract interface representations were not available in their UML tools. To compensate for this, package diagram dependencies were used to represent interface inheritance relationships. This novel interpretation tended to confuse commercial reverse engineering/import tools even though REACT extraction could be made to understand contractor-specific UML idioms.

Legacy Architecture Remodeling in UML. We support customers with a several large, complex architectures that were designed prior to UML and have an expected life of several decades. Much of the architectural information is contained in lengthy specifications. Also many of the original software designers have retired or are nearing retirement. Recently we have suggested that portions of the legacy architecture protocols undergoing evolution be remodeled into UML to not only capture the currently deployed architecture, but also permit architectural analysis of the future proposed changes. The recommendation has worked well for us. Our remodeling activities have not only uncovered architectural inconsistencies, but also identified areas for improvement.

Electronic Architectural Formats. We hope that REACT's architectural analysis success motivates customers to require electronic delivery of all architectural information. Over time XML data schemas will be defined to facilitate its organization and use. Initially we thought requiring XMI as the standard UML exchange format would be a standard way to begin REACT's architectural analysis and we have been modestly successful basing REACT's extractor on XMI. However, although XMI support is improving, today, not all UML tools fully support it, and there are variations in how thoroughly a vendor's UML extensions are captured in specialized tags. Until there is better coordination between UML releases and XMI releases, we recommend that customers request the full UML model be delivered, create their own XMI exports, and create vendor-specific extraction only if needed. Many commercial UML tools provide APIs to extract their proprietary information.

A Knowledgebase of Architectural Information. We found that a large body of contractor provided architecturally detailed information is not directly specified within UML. These include sizing and timing analyses to motivate queues lengths, CPU needs, memory allocations, etc. We are currently working on ways to organize this auxiliary information to bring it into REACT for analysis. We are investigating data repositories using XML data schemas. Our legacy systems are quite complex and have evolved into specialized systems in which few individuals have all thoroughly mastered. The distributed nature of the knowledgebase requires efficient access to various sources of contractor and programmatic information.

Improved collaboration with independent readiness review (IRR) teams. Typically IRR teams are given code about to be deployed and are asked to evaluate its readiness. Because they typically have not been involved during its design, they often begin their analysis from scratch. We found that there is a natural synergy between IRR engineers and REACT's early assessment engineers because the early assessors can identify the troubled areas during design and the IRR engineers can provide more concrete parameterization data to refine REACT models. We hope others find this approach useful.

REACT, an architecture-centric analysis environment that extracts, semantically analyzes, represents, and analyzes contractor-provided architectural information has been presented. REACT developed UML tags to augment architectural information to enable automated model generation, and permits the reuse of augmented model information in subsequent architecture development phases through Model Export. Improved support for developing model parameterization and development of advanced quantitative techniques to support dynamic assessment of modeled architectures are ongoing. This includes techniques to specify, monitor, collect, and interpret dynamically acquired assessment data. A long-term plan for REACT is to develop an environment that can evaluate adaptive execution or plan-based strategies using rule-based logic or probability belief networks. Approaches to assess the impact of redesigned/refactored architectures using REACT's internal representation are also being studied.

ENDNOTE

1 © 2001 The Aerospace Corporation

REFERENCES

[1] Lee, Edward "Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java." Edward Lee, et. al. June 2000. Also see http://ptolemy.eecs.berkeley.edu.

[2] Schmidt, Phillip, "REACT UML Extraction, Augmentation and Modeling Support" (draft July 31, 2001) to be published.

Detection of Shotgun Surgery and Message Chain Code Smells using Machine Learning Techniques