Scenario-Driven Dynamic Analysis of Distributed Architectures

George Edwards, Sam Malek, and Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
{gedwards, malek, neno}@usc.edu

Abstract. Software architecture constitutes a promising approach to the development of large-scale distributed systems, but architecture description languages (ADLs) and their associated architectural analysis techniques suffer from several important shortcomings. This paper presents a novel approach that reconceptualizes ADLs within the model-driven engineering (MDE) paradigm to address their shortcomings. Our approach combines extensible modeling languages based on architectural constructs with a model interpreter framework that enables rapid implementation of customized dynamic analyses at the architectural level. Our approach is demonstrated in XTEAM, a suite of ADL extensions and model transformation engines targeted specifically for highly distributed, resource-constrained, and mobile computing environments. XTEAM model transformations generate system simulations that provide a dynamic, scenario- and risk-driven view of the executing system. This information allows an architect to compare architectural alternatives and weigh trade-offs between multiple design goals, such as system performance, reliability, and resource consumption. XTEAM provides the extensibility to easily accommodate both new modeling language features and new architectural analyses.

1 Introduction

Many modern-day software systems are targeted for highly distributed, resource-constrained, and mobile computing environments. In addition to the difficulties inherent in traditional distributed system development, such as unpredictable network latencies and security concerns, this new environment forces software developers to cope with additional sources of complexity. For example, developers must assume an inherently unstable and unpredictable network topology; they must elevate resource utilization concerns to the forefront of design decisions; and they must take application power consumption profiles into account.

As the complexity associated with software development has increased in this new setting, software engineers have sought novel ways to represent, reason about, and synthesize large-scale distributed systems. The field of *software architecture* has advanced new principles and guidelines for composing the key properties of such systems [1]. In many cases, the concepts and paradigms developed through research in software architecture have drastically altered the way developers conceptualize

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 125–139, 2007.

software systems. For example, in an effort to raise the level of abstraction used for describing large-scale distributed systems above the object-oriented constructs provided by previous software modeling technologies, such as UML 1.x, researchers have attempted to create architecture description languages (ADLs) and associated toolsets that provide the system modeler with higher-level architectural constructs. ADLs endeavor to capture the crucial design decisions that determine the ultimate capabilities, properties, and qualities of a software system [2]. ADL-based representations can be leveraged throughout the software development process for communication and documentation, examination and analysis, validation and testing, and refinement and evolution.

However, the software architecture community has struggled to invent modeling technologies that are semantically powerful as well as flexible and intuitive. ADLs have generally either focused on structural elements (to the detriment of other important system characteristics) or have relied on rigid formalisms that have a narrow vocabulary and cumbersome syntax [3]. The result is that, for many domains (including the mobile systems domain), crucial aspects of a system are not expressible in any of the existing ADLs. As just one example, power usage characteristics, while integral to mobile and embedded systems, are completely ignored by prominent ADLs.

In parallel with (and largely unaffected by) these developments, model-driven engineering (MDE) has emerged as a promising approach to distributed software system development that combines domain-specific modeling languages (DSMLs) with model transformers, analyzers, and generators [4]. DSMLs codify the concepts and relationships relevant in a particular domain as first-class modeling elements [6]. DSMLs also provide multiple model views and specify domain rules that define model validity (*i.e.*, well-formedness). Model transformers, analyzers, and generators examine and manipulate models to create useful artifacts such as component specifications and implementations, supplementary views of the system, or descriptions of emergent behavior.

The work described in this paper leverages the respective strengths of ADLs (*i.e.*, high-level, architectural description) and MDE (*i.e.*, domain-specific extensibility and model transformation) in support of a novel, scenario-driven approach to the modeling and analysis of distributed software architectures.

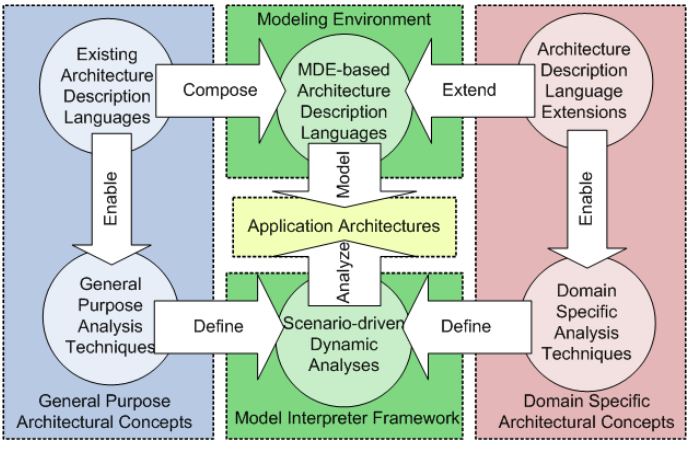

Fig. 1. Software architecture in the model-driven engineering context

Our approach combines extensible modeling languages based on architectural constructs with a model interpreter[1] framework that enables rapid implementation of customized dynamic analyses at the architectural level. The analyses provide statistical data quantifying emergent behaviors and cross-cutting system properties (*e.g.*, end-to-end latencies and system-wide power consumption).

In this manner, an architect can compare architectural alternatives and weigh trade-offs between multiple design goals. In particular, during the early design stages, a software architect can target high-risk events by modeling scenarios that represent unusual or dangerous conditions (*e.g.*, extremely heavy loading). The artifacts produced by our approach can be leveraged by an architect during other stages of the development cycle, as well. For example, during system maintenance and evolution, our approach can be used to assess the impact of modifications to the system (*e.g.*, replacing components with newer versions). A high-level view of the overall approach is shown in Figure 1.

Our initial study of the approach targets software development in distributed, resource-constrained, and mobile computing environments, which is a setting that presents significant challenges for software architects. To demonstrate the approach, we have developed the eXtensible Tool-chain for Evaluation of Architectural Models (XTEAM). XTEAM provides ADL extensions for mobile software systems and implements a corresponding set of dynamic analyses on top of a reusable model interpreter framework. Architectural models that conform to the XTEAM ADL are constructed in an off-the-shelf meta-programmable modeling environment. XTEAM model translators transform these architectural models into executable simulations that furnish measurements and views of the executing system over time.

In order for our approach to be successful, it must fulfill two key requirements:

R1: the language should be extensible to accommodate new domain-specific concepts and concerns as needed.

R2: the provided tool support should be flexible to allow rapid implementation of new architectural analysis techniques that take advantage of domain-specific language extensions.

This paper is organized as follows: Section 2 provides an overview of related work in software architecture and MDE. Section 3 describes how our approach reconceptualizes architectural description and analysis techniques in the MDE paradigm. Section 4 describes our complete tool-chain in detail in order to demonstrate the overall approach. In Section 5, we provide a discussion of the salient aspects of our work. We conclude the paper by summarizing and discussing future directions of this research effort.

2 Related Work

The approach described in this paper builds on previous projects and advancements in software architecture and model-driven engineering. However, our approach and associated tool-chain, XTEAM, exhibit several key differences from previous work.

[1] We use the term “interpreter” to denote a custom-built component that utilizes and manipulates models in order to perform functions such as transformation and analysis.

So that we may better illustrate these differences, this section provides an overview of related projects in software architecture and MDE. In Section 3, we examine more closely how our approach addresses the shortcomings of and represents an improvement over related techniques and technologies.

2.1 Model-Driven Engineering

The flexible nature of MDE has made it a suitable approach for representing different and arbitrarily complex concerns across a wide spectrum of application domains. Although a number of previous works have applied MDE to the analysis and synthesis of distributed and embedded software systems [5, 6, 9], they have either done so at a finer level of granularity than the system's software architectural constructs, or have been tied to a particular implementation platform or analysis engine. While such approaches are very useful in specific contexts, they do not leverage the concepts and paradigms developed by the software architecture community in their modeling languages, and do not provide a framework that allows rapid implementation of customized analyses. Below we provide a brief overview of the most notable MDE projects related to our work.

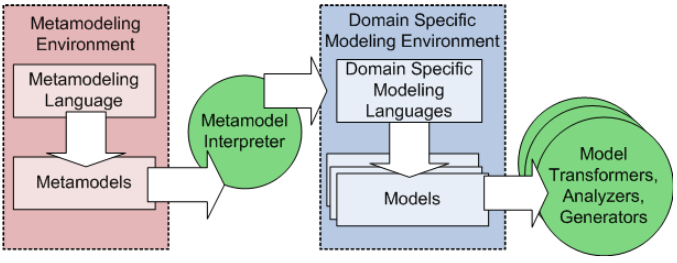

Fig. 2. High-level view of the model-driven engineering process as implemented in GME

Generic Modeling Environment. The MDE paradigm is realized via the appropriate tool support. One of the most widely used MDE tools is the Generic Modeling Environment (GME) [8]. GME is a meta-programmable, graphical modeling environment that enables the creation of domain-specific modeling languages (DSMLs) and models that conform to those DSMLs, as shown in Figure 2. GME also provides interfaces for custom-built components (*i.e.*, model interpreters) to access the information captured in models in order to conduct analysis or synthesize useful artifacts. To demonstrate the approach described in this paper, we have implemented a significant portion of XTEAM in GME.

MILAN. MILAN [6] comprises a DSML for embedded systems based on a dataflow representation, and a set of model translators that generate executable specifications for simulation engines. The dataflow formalism consists of nodes connected by directed edges. Functional, performance, and power simulations can all be generated from a single system model. MILAN also enables automated synthesis of software implementations from system models. While MILAN enables highly accurate simulations, the modeling language requires the developer to build system models using low-level constructs. As noted earlier, the MILAN language is based on a hierarchical dataflow representation. This is an appropriate formalism for signal processing systems, but is not sufficient for large-scale distributed architectures.

Attempting to build and maintain the model of such a system using dataflow can quickly become unmanageable and overwhelming. In contrast, the high-level structural and behavioral abstractions employed by ADLs allow the construction, review, and maintenance of large, complex models with reduced effort and potential for error.

WML and CUTS. The Workload Modeling Language (WML) is another DSML that enables dynamic analysis of component-based architectures [9]. WML allows the modeler to create descriptions of the resource utilization patterns of components for the purpose of evaluating system-wide quality-of-service (QoS) properties. WML models can be automatically transformed into the XML-based inputs required by the Component Workload Emulator Utilization Test Suite (CUTS). WML is tightly coupled to CUTS and requires that models be specified in terms of emulator constructs. The WML model interpreter performs a syntactic translation (from graphical models to XML) rather than a semantic translation (from architectural constructs to simulation constructs). Furthermore, the analysis provided by WML and CUTS is implemented in the emulator engine, rather than in the model interpreter, so the implementation of new analysis techniques would require *changes to* the infrastructure, rather than *utilization of* the infrastructure. Finally, WML does not capture component behavior in a generalized way that permits the representation of complex control flow paths.

2.2 Software Architecture

In this section, we examine two works that are relevant to XTEAM: Finite State Processes (FSP) [16], and xADL [7]. FSP is related to XTEAM because it is a modeling notation used to capture the behavior of software architectures; xADL is related because it is an *extensible* ADL. A number of other well-known ADLs provide some sort of static analysis capability; their relationship to XTEAM is considered in Section 3.

xADL. The eXtensible Architecture Description Language (xADL) was developed as a response to the proliferation of proposed ADLs, each of which had a different focus and addressed different architectural concerns. It was (correctly) observed that no single ADL could anticipate the needs of a wide variety of projects and domains. Consequently, the xADL language is inherently extensible and can be enhanced to support new domain-specific concepts. The language is defined by XML schemas; a "core" schema specifies standard architectural constructs common to all ADLs, while "extension" schemas — written by domain experts and tailored to the needs of specific projects — specify new modeling elements as needed. While xADL represents a promising step towards the flexibility and customizability required by contemporary large-scale distributed systems development, xADL's focus is primarily architectural representation rather than analysis, simulation, or the generation of implementation/configuration/deployment artifacts. xADL's supporting toolset consists of parsers and other syntactic tools that are semantically agnostic. Therefore, the xADL toolset cannot be extended to enforce semantic consistency within architectural models without modifying the toolset's implementation. This is in contrast to MDE, in which DSMLs ensure the construction of models that conform to domain-specific constraints, while model interpreters provide semantically-aware analysis. The result is that xADL, by itself, does not fully capitalize on the potential of architectural modeling.

FSP. FSP is a modeling notation for capturing the behavior of software architectural constructs in terms of guarded choices, local and conditional processes, action prefixes, and so on. FSP also allows for the construction of composite architectural constructs, in which the behavior of a composed element is defined in terms of the behavior of its constituents. While previous works have leveraged FSP models for analysis and simulation of a system's architecture [16], they have not focused on a number of concerns that are important for distributed systems executing in heterogeneous environments, including the structural aspects of an architecture, its deployment onto physical hosts, or extensibility of the notation.

3 Reconceptualization of ADLs

By recasting the concepts and techniques developed by the software architecture community in a model-driven engineering framework, the benefits of an architecture-based approach to large-scale distributed system development are preserved, while the shortcomings of the approach are diminished. The architecture-based approach to software modeling, and the ADLs that support that approach, suffer from two key drawbacks: inflexible notations with a narrow vocabulary, and supporting tools that enable only a limited set of analyses.

Note that these two drawbacks are the corollary of the requirements stated in Section 1. The hypothesis underlying this research is that these shortcomings can be addressed by representing ADLs (and compositions thereof) via domain-specific modeling languages (DSMLs), and performing architectural analysis via model interpreters. However, to achieve this result, two key challenges must be overcome:

- Development of ADLs, even with the benefit of MDE environments, is inherently challenging and requires both software architecture and metamodeling expertise.
- Implementation of custom-built model interpreters that access the information contained in models to perform architectural analyses requires significant effort.

In this section, we provide an overview of our approach for overcoming the above challenges in order to represent and analyze software architectures via MDE techniques and facilities. In Section 4, the conceptual strategies outlined here are made concrete through a detailed discussion of XTEAM and explanatory examples.

3.1 ADLs as Domain-Specific Modeling Languages

The first step in leveraging the MDE approach for software architecture is to codify ADLs as DSMLs within a MDE framework, such as that provided by GME. However, as mentioned above, the creation of semantically powerful, flexible, and intuitive ADLs is non-trivial; in fact, it requires a great deal of expertise in both software architecture and modeling languages. An ADL developer must command a thorough understanding of the central and elemental concepts in software architecture and must be adept in the mechanisms for codification of those concepts.

To overcome this challenge, we advocate an approach that avoids the creation of ADLs from scratch. Instead, we rely on *ADL composition* (*i.e.*, the combination of constructs from multiple ADLs) and *ADL enhancement* (*i.e.*, the definition of new, customized ADL constructs). MDE technologies capture the concrete syntax of DSMLs

through *metamodels*. Once the metamodel for an ADL has been created, the ADL can be manipulated as needed for a given application domain. Thus, the composition and enhancement of ADLs is achieved through composition [10] and enhancement of their corresponding metamodels. Existing notations and languages can be reused to the greatest extent possible, and only incremental additions to the language are created as needed to enable a specific architectural analysis technique. In addition to reducing the burden of language development, this distinction is important for two reasons: (1) existing ADLs are based on well-understood concepts and generally provide formal semantics, which increases model understandability, and (2) utilization of common languages maximizes the potential for reuse of the tool infrastructure (modeling environments and model interpreters) across development projects and domains. Both ADL composition and enhancement are utilized in XTEAM.

ADL composition allows the various concepts and design information expressible in different ADLs to be captured in a single language, which then allows models that conform to the language to be utilized in a variety of ways and at multiple stages of the development cycle. Composition of ADL metamodels is simplified by the fact that most ADLs share a small set of common elements (*i.e.*, components, connectors, interfaces, and so on) [Medvidovic N., et al.: A Classification and Comparison Framework for Software Archite] that serve as the integration point. In Section 4, we describe in detail how features from multiple ADLs are seamlessly integrated in XTEAM.

Some system properties that are not addressed by a general-purpose ADL may be of significant concern for certain application domains. Furthermore, each type of architectural analysis requires certain types of information to be modeled and represented, which may not be supported by a general-purpose ADL. Therefore, ADL enhancement—the ability to incorporate domain-specific concepts via new extensions—is highly valuable in an architectural modeling tool. The metamodeling mechanism provided by GME makes this process straightforward and intuitive: the new information is added to the existing language by creating new elements or new attributes of existing elements. This ADL metamodel enhancement mechanism provides the means through which requirement R1 (stated in Section 1) is satisfied. We will further illustrate the motivation and utility of ADL enhancement through a detailed example in Section 4.

In this way, the mechanisms for language refinement, enhancement, and evolution are built into MDE tool-chains. As a consequence, the notations used can always be modified and the vocabulary expanded. As language extensions are developed, new types of analysis will become possible. In Section 4, we describe how a model transformation framework can be utilized to rapidly implement customized analyses.

3.2 Architectural Analyses as Model Interpreters

Typically, an ADL is accompanied by tool support that is geared specifically to the notations provided by the language, thus only allowing for limited types of analysis. On the other hand, the MDE paradigm advocates flexible tool support, where multiple model interpreters with different analysis capabilities can be utilized. This characteristic is absolutely necessary for reasoning about the varying and evolving concerns of large-scale software systems.

MDE tools, such as GME, provide interfaces for custom-built model interpreters to access and manipulate the information contained in models. However, building these

tools from scratch requires significant effort. In many cases, a complex semantic mapping between languages is required that is difficult to define and implement. Such a mapping is required to transform architecture-based models, which are at a very high level of abstraction, into simulation models, which are at a much lower level. For this reason, our approach utilizes a *model interpreter framework* that allows the software architect to rapidly implement custom analysis techniques without knowing the details of complex semantic mappings (*e.g.*, the architecture-to-simulation mapping is achieved by the framework infrastructure "under-the-hood"). The interpreter framework provides "hook" methods for which the architect provides implementations that, taken as a whole, realize a specific analysis technique. The objects available to the architect in the implementation of the analysis technique are the architectural constructs defined in ADL extensions, not low-level simulation constructs. The model interpreter framework, through its "hook" methods, greatly simplifies the development of simulation generators, and thus provides the means through which requirement R2 is satisfied. Section 4 demonstrates the use of the XTEAM model interpreter framework in the implementation of a specific analysis technique.

Analysis techniques may be static or dynamic. Static analysis techniques rely on the formalisms underlying models to provide information about system properties or expose subtle errors without executing the system [11]. Many static analyses attempt to prove the "correctness" of a system, which may be useful in many scenarios, but suffers from the difficulty that system implementation must precisely match the model in order for the analysis to be relevant. Dynamic analysis, on the other hand, attempts to execute model-based architectural representations in order to illuminate their characteristics and behaviors at run-time. Dynamic analysis is more useful for comparing high-level design possibilities early in the development cycle because it does not require that a model be completely faithful to the eventual implementation to remain relevant. Static analysis has an important place in the development of certain types of software systems, but dynamic analysis is more relevant when an architect wishes to understand the system's behavior within the context of specific execution scenarios.

For these reasons, our approach has thus far focused on dynamic analysis through system simulation. Model interpreters synthesize executable specifications that run on a simulation engine. The simulation code is instrumented to record the occurrence of events (*e.g.*, message exchanges and component failures) and measurements of system properties (*e.g.*, observed latencies and memory usage). The results of a simulation run depend heavily on the environmental context (*e.g.*, the load put on the system) and may contain elements of randomness and unpredictability (*e.g.*, the timing of client requests). Consequently, we consider our approach to be *scenario-driven*, in that a given simulation run represents only one possible execution sequence. The software architect should choose the set of scenarios to be simulated to include high-risk situations and boundary conditions. While this simulation-based approach does not provide a formal proof of system behavior, it does allow the architect to rapidly investigate the consequences of fundamental design decisions (*e.g.*, choice of architectural style or deployment architecture) in terms of their impact on non-functional properties (*e.g.*, reliability, performance, or resource utilization).

4 The XTEAM Tool-Chain

In this section, we demonstrate the approach described in Section 3 in the eXtensible Tool-chain for Evaluation of Architectural Models (XTEAM), a model-driven architectural description and simulation environment for distributed, mobile, and resource constrained software systems. XTEAM composes existing, general-purpose ADLs, enhances those ADLs to capture important features of mobile software systems, and implements simulation generators that take advantage of the ADL extensions atop a reusable model interpreter framework.

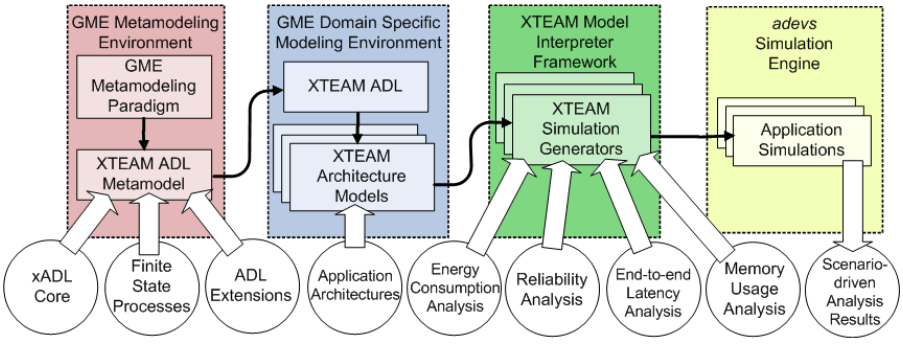

Fig. 3. The eXtensible Toolchain for Evaluation of Architecture Models

A high-level view of XTEAM is shown in Figure 3. Using GME's metamodeling environment, we created an XTEAM ADL metamodel by composing a structural ADL, the xADL Core [7], with a behavioral ADL, FSP [16]. GME uses the XTEAM metamodel to configure a domain-specific modeling environment in which XTEAM architectural models can be created. With this language basis, we were able to implement the XTEAM model interpreter framework, which provides the ability to generate simulations of application architectures that execute in the *adevs* [17] discrete event simulation engine. However, these simulations alone do not implement any architectural analysis techniques. To do so, we enhanced the XTEAM ADL metamodel with language extensions that capture system characteristics relating to energy consumption, reliability, latency, and memory usage, thereby demonstrating the fulfillment of requirement R1. We then utilized the extension mechanisms built into the model interpreter framework in such a way as to generate simulations that measure, analyze, and record the properties of interest, thereby demonstrating the fulfillment of requirement R2. We further elaborate on this process below.

4.1 Composing ADLs and Implementing a Model Interpreter Framework

Using the approach described in Section 3, XTEAM leverages previous work in software architecture through the composition of existing ADLs. We created metamodels for the xADL Core, which defines architectural structures and types common to virtually all ADLs, and FSP, which allows the specification of component behaviors. The integration of these metamodels was straightforward, as they capture largely orthogonal concerns.

For illustration, the metamodel for xADL Core is shown in Figure 4 (along with additional extensions that are discussed in the next subsection). Components and connectors represent the basic building blocks for architecture models. They contain interfaces that can be connected via links. Interface mappings denote the realization of an interface by the interface of a sub-component. The group element captures the membership of multiple components and connectors in a set. Components and connectors may contain substructures. Finally, several of the xADL Core elements include typed attributes, such as a generic description.

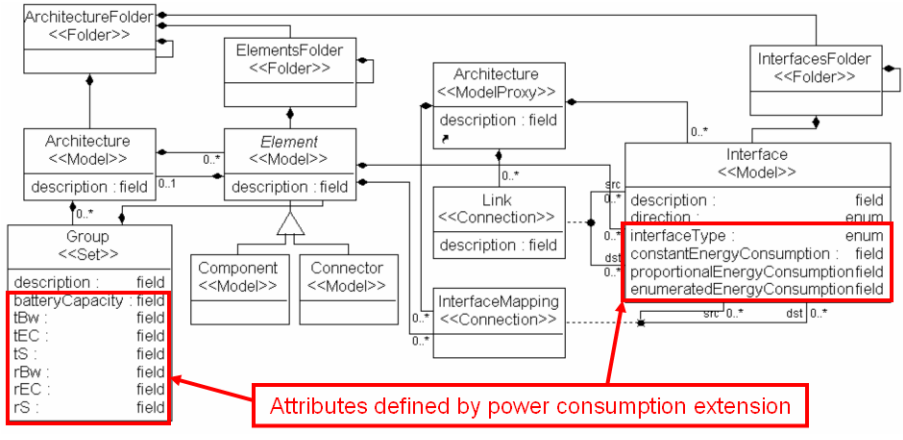

Fig. 4. The metamodel of xADL Core (with the energy consumption extension) as implemented in XTEAM

The combination of xADL and FSP allowed us to create executable architectural representations. Models conformant to the composed ADL contain sufficient information to implement a semantic mapping into low-level simulation constructs that can be executed by an off-the-shelf discrete event simulation engine [18], such as *adevs*. This semantic mapping is implemented by our model interpreter framework. The framework infrastructure synthesizes the low-level structures (*e.g.*, atomic and static digraph models) and logic (*e.g.*, state transition functions) needed by *adevs*. "Hook" methods provided by the framework allow an architect to generate the code needed to realize a wide variety of dynamic analyses. The following subsection explains this process in more detail through the use of concrete examples.

4.2 Domain-Specific Extensions and Architectural Analyses

In order to take advantage of the extensibility and flexibility afforded by the MDE approach, we implemented several domain-specific ADL extensions within the XTEAM metamodel, and then relied on the interpreter framework to efficiently implement analysis techniques that operate on the information captured in those extensions. As XTEAM targets the development of architectures for distributed, mobile, and resource constrained software systems, we chose to implement analyses that are highly relevant for that domain. Below we elaborate on our implementation of one such analysis in XTEAM and briefly describe three others.

4.2.1 Energy Consumption Extensions and Analysis

The energy consumption of executing software has traditionally been ignored by software engineers as they could rely on an uninterrupted, abundant energy source. In the mobile setting, this assumption no longer holds, and the energy consumption of software components can have an important impact on system longevity. The energy consumption estimation framework described in [13] provides a mechanism for estimating software energy consumption at the level of software architecture.

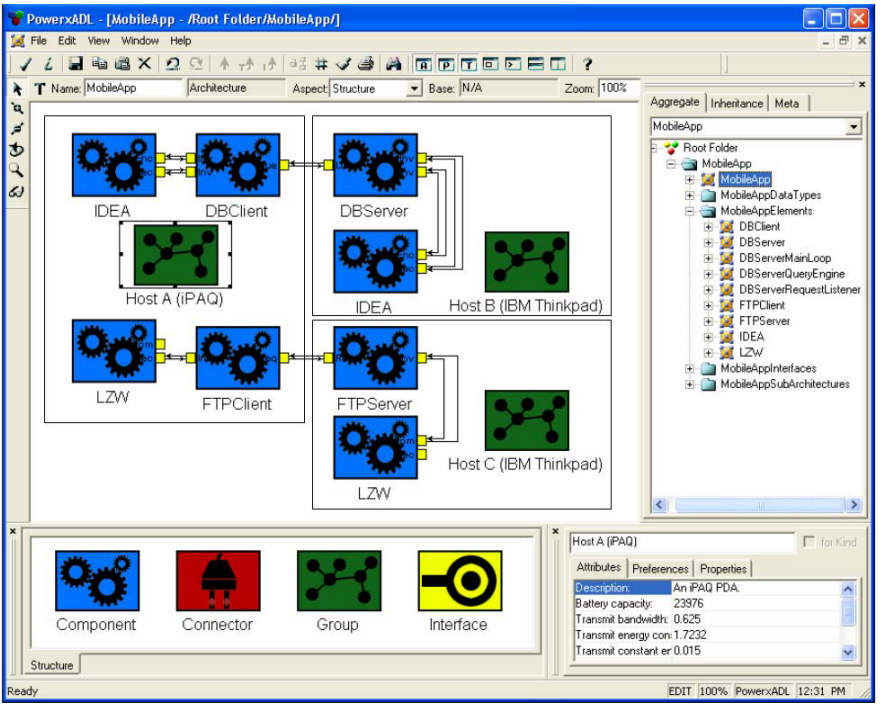

Fig. 5. Model of the mobile application architecture in XTEAM

At a high level, the energy consumption estimation framework defines the overall energy consumption of a software component as a combination of its computational and communication energy costs. The computational energy cost (due to CPU processing, memory access, I/O operations, *etc.*) is incurred whenever one of the component's interfaces is invoked, while the communication energy cost is incurred whenever data is transmitted or received over the wireless network. The estimation framework provides equations that enable the calculation of these energy costs based on a number of a parameters, including data sizes and values, the rate of energy consumption during data transmission, and network bandwidth. Enhancing the XTEAM metamodel with these values was straightforward: they were added as attributes to the appropriate elements (groups, which denote hosts in this context, and interfaces), as shown in Figure 4. Then, in our interpreter framework (recall Figure 3), we inserted the equations for energy consumption based on the parameters defined in

the model. The communication energy cost equation was inserted into the "hook" methods that correspond to the sending and receiving of data between components. If the components are on different hosts, the communication energy cost is deducted from the hosts' total battery power. Similarly, the computational energy cost equation was inserted into the "hook" method corresponding to the invocation of an interface.Whenever one of these events occurs during the simulation run, the energy consumption values are calculated and recorded.

To illustrate the use of the energy consumption ADL extension and analysis, consider the example of a small mobile application in which we have three distributed, mobile hosts: an iPAQ PDA and two IBM Thinkpad laptops. A top-level view of the XTEAM model for the mobile application is shown in the screen capture in Figure 5 (although most of the model detail cannot be seen in this view). A database client and a FTP client are deployed to the PDA, while the corresponding servers each run on one of the laptops. Components that perform encryption and compression (respectively, IDEA and LZW open-source components) are also deployed on the PDA and laptops. When the DB client wishes to query the database, it encrypts the query using the local IDEA component and sends the query over the wireless network to the DB server, where it is decrypted. The DB server then retrieves the result of the query from the database, encrypts the results, and sends them back to the DB client. An analogous path is used for the FTP client, except that compression rather than encryption is performed.

By invoking our energy consumption simulation generator (built using the interpreter framework) on the mobile application model, the energy consumption on each host can be determined dynamically. Plotting these measurements as a function of time results in the example graph shown in Figure 6.

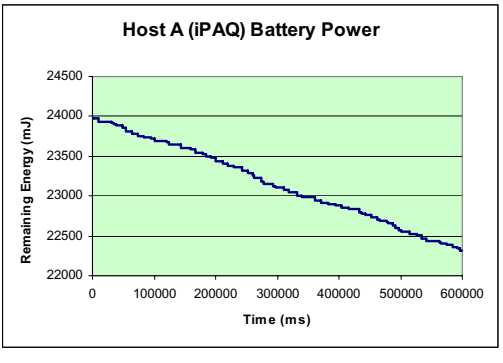

Fig. 6. Result of energy consumption analysis

This type of energy consumption estimation has a variety of uses in the scenario-driven analysis approach. First, the assignment of components to hosts, or deployment architecture, can have a significant effect on system energy consumption, and, in turn, its longevity. Our environment allows the architect to quickly model a set of potential deployment architectures, and then observe the energy consumption on each host over time. Moreover, the architect can determine whether a dynamic redeployment strategy is required in situations when the actual energy consumption rate differs significantly from expected rates. Conversely, given a requirement for the longevity of system services, the architect can begin to arrive at target energy usages for each component.

4.2.2 Other Extensions

We have also leveraged XTEAM to implement dynamic analysis capabilities for end-to-end latency, memory utilization, and component reliability. The implementation of ADL extensions and model interpreters for these analyses follows the same pattern as

that used for energy consumption, and demonstrates the fulfillment of our original requirements. For example, we implemented the component reliability extension and analysis based on the technique described in [12]. This reliability estimation approach relies on the definition of component failure types, the probabilities of those failures at different times during component execution, and the probability of and time required for failure recovery. This type of analysis meets the primary criteria for implementation in XTEAM: it estimates the reliability of components at the level of software architecture. We extended the FSP-based behavior language in XTEAM to include failure and recovery events and probabilities. We also developed an analysis that determines if and when failures occur as the components in the system progress through different tasks and states. In general, we believe that given an architectural analysis technique that is applicable in a dynamic, simulated setting, our framework can be utilized to realize that technique through implementation of the appropriate hook methods.

5 Discussion

This section discusses our approach in the context of wider architectural development processes and activities. In particular, we see three cases where our approach is particularly relevant and unique: (1) providing rationale for fundamental architectural decisions, (2) weighing trade-offs among multiple conflicting design goals, and (3) understanding the results of composing independent components developed in isolation.

5.1 Providing Design Rationale

Early in the architectural development process, software architects are, in many situations, required to rely on their own intuition and past experience when weighing fundamental design questions. For example, the choice of a particular architectural style, the distribution of components across hosts, or the functionality allocated to components can dramatically effect the ultimate behaviors and properties of a system, but architects have very limited mechanisms for arriving at such decisions beyond their own knowledge and expertise and the collective wisdom of the architecture community. In other words, rationalizing such decisions using specific processes and tools is relatively rare. Our approach to software architecture provides a means of experimentation with fundamental design decisions and the rationalization of those decisions through quantifiable means. By generating and executing simulations of a distributed system, the consequences of crucial architectural choices can be better understood.

5.2 Weighing Architectural Trade-Offs

Nearly all non-trivial architectural decisions come down to trade-offs between multiple desirable properties. The relative importance of different system properties to the user (e.g., availability or performance) can be determined prior to architectural development, but the architect is still required to engineer the right balance between conflicting goals. Emphasizing one attribute over others will eventually yield diminishing returns, and usually this “tipping-point” between different qualities is anything but obvious. For example, given a system with both fixed and mobile hosts, deploying components to a mobile host will likely increase the availability and reduce

the latency perceived by a client using that device, but will also drain the battery power faster. The "right" deployment (*i.e.*, that which maximizes the system's utility given users' quality-of-service preferences [14]) depends heavily on the wireless network characteristics, such as bandwidth and the frequency of disconnects, in addition to a number of other factors. Rather than relying on intuition or past development projects to achieve the right balance, our approach allows an architect to determine the relationships between various design goals and increase system utility experimentally.

5.3 Understanding Compositions of Off-the-Shelf Components

In the present day, independent teams or organizations are often responsible for producing components that are ultimately assembled to create a unified system. In such settings, detailed information about individual components (*e.g.*, resource consumption, failure rates) may be available, but the properties of their composition may not be well-understood. In such a case, our approach can produce accurate measurements of the emergent properties of the composed system. This knowledge ultimately enhances the architects' understanding of the system and increases their confidence in the ability of the composed system to meet end-user operational goals. Both of these outcomes serve to reduce the risk associated with a large-scale development and/or integration project.

6 Conclusions

This paper presented a software architecture-based approach to modeling and analysis of distributed architectures that leverages the domain-specific extensibility provided by model-driven engineering. Our approach addresses the significant shortcomings of previous ADLs by relying on a tool-chain that enables both modeling language and analysis extensibility. The dynamic analysis capabilities of the tool-chain allow an architect to better understand the consequences of architectural decisions, focus on aspects that have the greatest effect on a system's critical properties, weigh trade-offs between conflicting design goals, and better understand component compositions. We demonstrated and evaluated the approach on XTEAM, a suite of ADL extensions and model transformation engines targeted specifically for highly distributed, resource-constrained, and mobile computing environments. We believe our approach represents an improvement over traditional ADLs and exhibits significant differences from other MDE tools that have been developed for distributed systems development.

There are several ways in which we intend to extend this work. First, we will utilize the XTEAM tool-chain in the context of architectural development for a real-world security application that operates in an embedded, wireless environment. Second, we will integrate XTEAM with other complementary architecture-based development tools, including DeSi [14] and Prism-MW [15]. Third, we will determine more precisely the exact class of analysis techniques that can be implemented with our model interpreter framework, and evaluate the feasibility of supporting other classes of analysis techniques (*e.g.*, static analyses) via additional interpreter frameworks. Lastly, we will further define ways in which our approach can be integrated with widely-used architectural development processes, such as the Architecture Trade-off Analysis Method (ATAM).

Acknowledgments

The work described in this paper was sponsored by the National Science Foundation under Grant number ITR-0312780. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF. This material was also sponsored by Bosch. The authors wish to thank the anonymous reviewers for their detailed and helpful comments.

References

[1] Perry, D. E., Wolf, A.L.: Foundations for the Study of Software Architectures. ACM SIGSOFT Software Engineering Notes, pp. 40-52, Oct 1992.
[2] Medvidovic N., et al.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. on Software Engineering, 26(1), Jan 2000.
[3] Medvidovic, N., Dashofy, E. and Taylor, R.N.: Moving Architectural Description from Under the Technology Lamppost. Journal of Systems and Software, 2006.
[4] Schmidt, D.C.: Model-Driven Engineering. IEEE Computer, 39(2), pp. 41-47, Feb 2006.
[5] Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T.: Model-integrated development of embedded software. In Proceedings of the IEEE, 91(1), pp. 145-164, Jan 2003.
[6] Ledeczi, A., et al.: Modeling methodology for integrated simulation of embedded systems. ACM Transactions on Modeling and Computer Simulation, 13(1), pp. 82-103, Jan 2003.
[7] Dashofy, E., van der Hoek, A. and Taylor, R.N.: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. Proceedings of the 24th International Conference on Software Engineering, pp. 266 - 276, 2002.
[8] GME: The Generic Modeling Environment. http://www.isis.vanderbilt.edu/projects/gme/
[9] Paunov, S., et al.: Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System Quality-of-Service. Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2006.
[10] Ledeczi, A., et al.: On metamodel composition. Proceedings of the 2001 IEEE International Conference on Control Applications, pp. 756-760, 2001.
[11] Jackson, D., Rinard, M.: Software Analysis: A Roadmap. In The Future of Software Engineering, Anthony Finkelstein (Ed.), pp. 215-224, ACM Press 2000.
[12] Roshandel, R., et al.: Estimating Software Component Reliability by Leveraging Architectural Models. 28th International Conference on Software Engineering, May 2006.
[13] Seo C., Malek S., N. Medvidovic: An Energy Consumption Framework for Distributed Java-Based Software Systems. Tech. Report USC-CSE-2006-604, 2006.
[14] Malek, S.: A User-Centric Framework for Improving a Distributed Software System's Deployment Architecture. To appear in proceedings of the doctoral symposium at the 14th Symposium on Foundation of Software Engineering, Portland, Oregon, Nov. 2006.
[15] Malek, S., et al.: Prism-MW: A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. IEEE Trans. on Software Engineering. 31(3), Mar. 2005.
[16] Magee, J., et al.: Behaviour Analysis of Software Architectures. Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), pp. 35 - 50, 1999.
[17] Adevs: A Discrete EVent System simulator. http://www.ece.arizona.edu/~nutaro/
[18] Schriber, T. J., Brunner, D.T.: Inside Discrete-Event Simulation Software: How it Works and Why it Matters. Proceedings of the Winter Simulation Conference, 2005.