Construction of Analytic Frameworks for Component-Based Architectures

George Edwards, Chiyoung Seo, Nenad Medvidovic

University of Southern California
Los Angeles, CA 90089-0781 U.S.A.

{gedwards,cseo,neno}@usc.edu

Abstract. *Prediction of non-functional properties of software architectures requires technologies that enable the application of analytic theories to component models. However, available analytic techniques generally operate on formal models specified in notations that cannot flexibly and intuitively capture the architectures of large-scale distributed systems. The construction of model interpreters that transform architectural models into analysis models has proved to be a time-consuming and difficult task. This paper describes (1) how a reusable model interpreter framework can reduce the complexity involved in this task, and (2) how such a framework can be designed, implemented, utilized, and verified.*

1. Introduction

Modern-day component technology provides software architects with powerful mechanisms for designing, implementing, deploying, and evolving large-scale distributed systems. In particular, component technology encompasses both highly effective strategies for reuse and integration of existing software (*e.g.*, off-the-shelf components), as well as run-time platforms (*e.g.*, component-based middleware) that hide low-level complexity beneath high-level design abstractions. Perhaps more subtly, component-based software engineering also offers a basis for the construction of analysis models that enable the discovery and prediction of critical system properties, such as performance, reliability, and resource consumption. Although techniques for analyzing software systems with respect to such properties are not new, the assembly of systems from independently deployable and executable units makes these techniques more relevant and practical.

Analysis of non-functional properties is critical in weighing design alternatives. Nearly all non-trivial architectural decisions come down to trade-offs between multiple desirable properties, and a software architect is required to engineer the right balance between conflicting goals. Furthermore, subtle interactions between components can result in unforeseeable and unpredictable system behaviors. Quantitative evaluation of non-functional properties therefore provides an architect with concrete rationale for fundamental design decisions and reduces the risk associated with a large-scale development and/or integration project [2].

To effectively analyze the non-functional properties of a component-based system, methods and tools are needed that support the integration of component technologies and analysis technologies [1]. A *component technology* consists of a component model along with a development environment and/or run-time platform. The component model imposes rules that define the well-formedness of component instances and assemblies. Component technologies (such as Enterprise Java Beans) provide the basis for the modeling, implementation, and deployment of software architectures. An *analysis technology* consists of a system analysis technique and tools that support the utilization of that technique. An analysis technique is a process for applying a computational theory to

system models (such as Layered Queuing Networks, or LQNs) to enable automated prediction of system properties and behaviors. Software and system analysis techniques are required to make assumptions about the systems to which they are applied. For example, a LQN assumes that each software server accepts requests from a single queue [13]. Such an assumption can be enforced by a component middleware platform at system construction-time and at run-time. Component technologies and analysis technologies are well-suited to integration because a component technology can be used during system construction to enforce the assumptions required by an analysis technology.

Unfortunately, the integration of component and analysis technologies is anything but straightforward in practice. Component-based architectures are generally specified using high-level design languages that emphasize abstraction and flexibility, while analysis techniques operate on formal models that are frequently specified in much lower-level, more rigid notations. The consequence of this is that software architects are frequently required to construct multiple system models for different purposes. For example, the safety experts on an architecture team may build and analyze fault trees, while energy management experts construct and execute specialized power simulations.

This paper presents an approach to achieving the seamless integration of component technologies and analysis technologies. At the core of our approach is the development of highly flexible *model interpreter frameworks*, which implement semantic mappings between a component model and an analysis model, yet lend themselves to a wide variety of non-functional analyses. A model interpreter framework can be reused by a software architect to rapidly construct analyzable models from domain-specific architectures. To illustrate the approach, we describe our implementation of a model interpreter framework in XTEAM, a modeling and analysis framework targeted at mobile and resource-constrained software systems. Our evaluation of XTEAM demonstrates that (1) an interpreter framework can provide accurate predictions of the non-functional properties of a software architecture, and (2) a single interpreter framework can be used to rapidly and successfully implement a broad range of analysis techniques.

The remainder of this paper is organized as follows. Section 2 further motivates this work. Section 3 describes our approach, while Section 4 discusses in detail our implementation of the XTEAM model interpreter framework. Section 5 evaluates the framework quantitatively and illustrates its benefits through the use of an example. Overviews of related work and conclusions round out the paper.

2. Background and Motivation

Model-driven engineering (MDE) [4] offers an attractive strategy for analyzing the non-functional properties of software architectures. MDE technologies enable the construction of *domain-specific modeling languages* (DSMLs) through the use of metamodels. Metamodels capture the elements, attributes, relationships, views, and constraints present in a modeling language, and can be easily modified, adapted, composed, enhanced, and evolved [5]. In this way, MDE offers an intuitive way to incorporate the parameters of an analytic theory into both general-purpose and domain-specific component models. MDE technologies provide access to the information contained in architectural models through well-defined interfaces. Customized *model interpreters* can then be constructed that perform system analysis and visualization, automated synthesis of implementation artifacts, *etc*. Figure 1 delineates the main MDE concepts and processes.

Model interpreters can be used to implement semantic mappings, or transformations, between high-level architectural models and the low-level analysis models amenable to rigorous prediction of component assembly properties. However, numerous practical challenges remain. In order to motivate the discussion in the remainder of this paper and illustrate the need for a new approach to the construction of model interpreters, this section describes a typical MDE-based process for modeling and analyzing a software architecture, and demonstrates how this process can be simplified and improved.

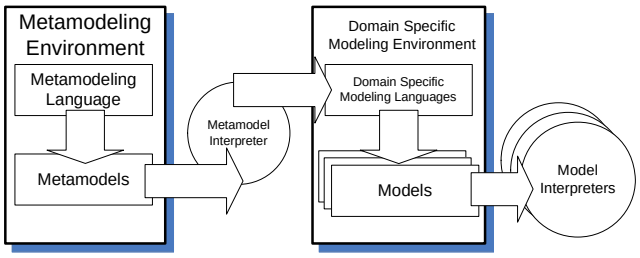

Figure 1. High-level view of the MDE process.

Consider a large-scale software development project. The software architecture team has decided to employ an MDE-based modeling and analysis process, and has consequently constructed an architectural model that includes some domain-specific elements (such as hardware and middleware) in addition to the canonical architectural constructs (component, connector, *etc.*). The team now plans to analyze the performance of the system through the use of a LQN model. Applying the standard MDE strategy, the team constructs a model interpreter that transforms the architectural model into a LQN, which is then analyzed to determine a set of performance-related metrics, such as system throughput and service utilization, under various loading conditions.

As the development program progresses, the need for additional analyses becomes apparent. For example, as the system's deployment of software components to hardware hosts is further refined, questions arise about how deploying certain components to mobile hosts will impact the system's energy consumption. The architecture team is instructed to study deployment alternatives with respect to energy consumption. To do so, they implement a new model interpreter that transforms the architecture model into the input to a cycle-accurate energy consumption simulator.

As other forms of analysis are requested, the team is forced to expend significant resources implementing model interpreters. For each new interpreter, the team must:

- Find a computational theory that derives the relevant properties from a system model.
- Determine the syntax and semantics of the modeling constructs on which the computational theory operates.
- Discover the semantic relationships between the constructs present in the architectural models and those present in the analysis models.
- Determine the compatibility between the assumptions and constraints of the architectural models and the analysis models, and resolve conflicts between the two.
- Implement a model interpreter that executes a sequence of operations to transform an architectural model into an analysis model.
- Verify the correctness of the transformation implemented by the interpreter.

This paper demonstrates how the use of a model interpreter framework allows an architecture team to perform the above tasks only *once* for a broad *family* of analysis techniques, rather than repeating the process for *each* analysis technique. The use of interpreter frameworks can significantly improve the utility and appeal of MDE-based architectural development.

3. Model Interpreter Frameworks

Applying MDE to the analysis of component-based systems requires software architects to construct semantic mappings between component models and analysis models. The primary contribution of this paper is a novel approach that can greatly reduce the complexity involved in this task. Our approach is to leverage general purpose architectural modeling constructs and a widely applicable analytic representation to construct a *model interpreter framework* that abstracts most of the semantic mapping required for analysis, while still providing the extensibility to accommodate both domain-specific modeling elements and analyses. This section, therefore, focuses on defining specifically what an interpreter framework is, what the requirements of one are, and what capabilities an interpreter framework provides. In the next section, we focus on how an interpreter framework can be implemented and give concrete examples of how one can be used.

3.1. Definition

In the model-driven engineering paradigm, a *model interpreter* is a software component that operates on the information captured in a system model to produce some useful artifact. Model interpreters invoke an API provided by a modeling environment to extract the model structure and properties. A model interpreter codifies the semantics of the modeling constructs on which it operates by defining the consequences of the use of those constructs within a given context.

A *model interpreter framework*, then, is an infrastructure for constructing a family of model interpreters. In order to be useful, such a framework must encapsulate logic or algorithms that are useful in a wide variety of contexts. However, an interpreter framework is not a library of functions; rather, it is an active component that can be extended and enhanced in specific, predefined ways. Furthermore, an interpreter framework necessarily makes assumptions about the models on which it operates, and is therefore only applicable to a certain class of models. In the context of MDE, which advocates the inclusion of domain-specific constructs in modeling languages, this implies that a common base of domain-independent constructs exists on which the framework can operate; domain-specific constructs are then handled by framework extensions.

One example of a model interpreter framework is a component that synthesizes "glue-code" for a given middleware platform from a model of a software application. Such a model likely includes both domain-independent constructs, such as objects or components, and domain-specific constructs, such as the representation of the application business logic. A model interpreter framework can be constructed that utilizes the component interface specifications and topology to generate middleware glue-code, but leaves open extension mechanisms to insert logic that interprets the domain-specific behavior (*e.g.*, to generate component implementations).

When applied to the analysis of component-based systems, a model interpreter framework enables a family of analytic techniques to be applied to a component model by constructing from a high-level architectural model a more directly analyzable representation of a system, such as a discrete event simulation or Markov chain. In this context, the domain-independent elements of the model are those concepts common to all component-based architectures. The domain-specific elements of the model are the parameters of a relevant analytic theory, plus any additional domain- and platform-specific extensions and constraints. The model interpreter framework abstracts the semantic

mapping from architectural constructs to analysis constructs, while providing the extensibility to accommodate the logic that measures and records non-functional properties according to an analytic technique. The role of a model interpreter framework in the analysis of component-based software architectures is illustrated in Figure 2.

3.2. Assumptions and Requirements

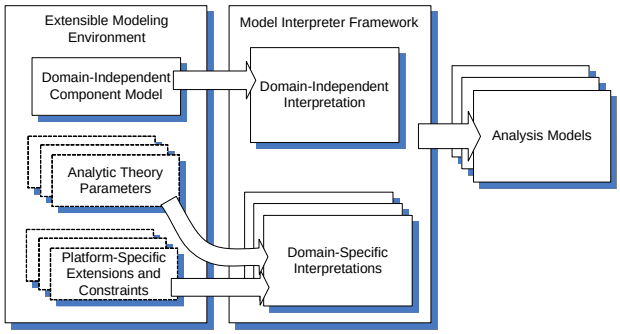

Figure 2. The role of a model interpreter framework in the analysis of component-based architectures.

As alluded to in the previous subsection, a model interpreter framework must make several important assumptions about the models to which it will be applied. It also must satisfy several requirements in order to be effective. In this subsection, we enumerate these assumptions and requirements and describe their consequences in terms of model interpreter frameworks in general. We also describe, for each assumption and requirement, the specific implications for model interpreter frameworks that provide non-functional analysis of application architectures.

3.2.1.Assumptions

Assumption 1. System models contain domain-independent elements that are sufficient to implement an interpretation. Interpreter frameworks encapsulate logic that operates on domain-independent constructs. It therefore follows that models must contain a sufficient set of domain-independent constructs to implement some useful interpretation. Modeling languages that consist exclusively of domain-specific constructs are not amenable to an interpreter framework. This assumption is clearly satisfied in the case of architecture models: the set of domain-independent elements common to most component-based architectures includes components, connectors, interfaces, and events.

Assumption 2. The interpretation of domain-independent elements is not dependent on the interpretation of domain-specific elements. The implementor of a model interpreter framework cannot know the types of domain-specific extensions that will be present in system models. Consequently, the framework logic must operate exclusively on domain-independent modeling elements, and the semantics of those elements cannot change within different domain-specific contexts. This assumption is not a significant problem for architectural models: the domain-specific modeling elements in this case are generally either the parameters of an analytic theory that will be applied to the model or platform-specific constraints on components and connectors.

Assumption 3. Domain-specific constraints do not violate domain-independent constraints. Constraints on the set of well-formed models are fundamental to every modeling language, and a model interpreter framework relies on these constraints in applying semantics to a model. Within a given domain, additional constraints are present; capturing these constraints is a crucial part of creating a domain-specific modeling language. Clearly, for a model interpreter framework to execute, these domain-specific constraints cannot contradict any domain-independent constraints.

This last assumption can, in some cases, constitute a major challenge when

applying an interpreter framework to architectural models. The constraints of a component model may be irreconcilable with the assumptions required by an analytic theory. However, more commonly, these constraints and assumptions can be brought into alignment by *co-refinement*, a process proposed by Wallnau et al. [1]. Co-refinement may weaken or strengthen the constraints of a component model, which either expands or reduces the set of well-formed models, respectively, in order to accommodate the assumptions of an analytic theory. Similarly, the assumptions of a analytic theory can be strengthened or weakened in order to reconcile conflicting component constraints.

3.2.2.Requirements

Requirement 1. The model interpreter framework abstracts the details of domain-independent interpretation. The manipulations performed by an interpreter framework are necessarily at least somewhat complex (otherwise, the reuse of the framework would be of little value). An interpreter framework should insulate architects from the details of these manipulations in order to enable reuse without forcing the architect to understand or modify the framework logic. This requirement, can, however, be relaxed in some cases, in order to increase the flexibility of the framework. Exposing the details of the interpretation process increases the complexity of utilizing the framework, but also allows the architect to implement certain analyses that would not otherwise be possible.

Requirement 2. The model interpreter framework produces an artifact useful in a wide variety of contexts. In order to maximize the benefits provided by reuse of an interpreter framework, the framework must produce a representation of the system that is flexible enough to be used for a variety of purposes. For example, some analysis models, such as discrete event simulations, enable the realization of an extensive family of analytic theories. Other analysis models, such as fault trees, are much more narrowly targeted, and enable a much smaller set of analytic theories. The latter types of analysis models are therefore not strong candidates for construction of an interpreter framework.

Requirement 3. The model interpreter framework provides extension mechanisms sufficient to accommodate domain-specific interpretation. The inclusion of extension mechanisms within an interpreter framework is the crucial feature that allows them to be applied to domain-specific models. As noted earlier, contemporary software architectures are increasingly incorporating domain-specific information as a strategy for managing complexity. Extension mechanisms are created through the use of design patterns such as Template Method, Strategy, and Functor [6]. These patterns allow domain-specific logic to be inserted into the interpreter framework at points of variability. Of course, the interpreter framework designer cannot predict every possible variability point. The choice of whether to include an extension mechanism at a potential point of variability is a design trade-off between flexibility and usability; that is, the inclusion of additional variability points makes the framework more widely applicable, but also increases the burden on a software architect utilizing the framework in a domain-specific context.

4. The Design of a Model Interpreter Framework

In this section, we describe in detail the design of a model interpreter framework we implemented as part of the eXtensible Toolchain for Evaluation of Architectural Models (XTEAM) [2], an environment that leverages the MDE paradigm to provide a reusable infrastructure for realizing domain-specific architectural analyses.

4.1. The XTEAM Toolchain

The eXtensible Toolchain for Evaluation of Architectural Models (XTEAM) allows an architect to analyze architectural models through a model interpreter framework that maps component models to executable simulations. Furthermore, XTEAM incorporates mechanisms to accommodate domain-specific extensibility at both the modeling and analysis phases of the architectural evaluation process.

A high-level view of XTEAM is shown in Figure 3. Using the Generic Modeling Environment (GME) [7], we created a domain-independent component model by composing the elements of the xADL Structures and Types ADL [19] and the Finite State Processes (FSP) ADL [20]. GME uses this component model to create a modeling environment in which architectural models that conform to the component model can be created. We also implemented a model interpreter framework that maps the component model to an analysis model — a discrete event simulation — and implements appropriate extension mechanisms, which are described further in the next subsection.

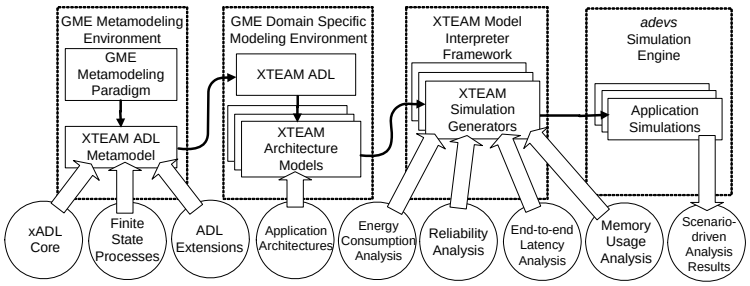

Figure 3. The eXtensible Toolchain for Evaluation of Architectural Models.

An architect takes advantage of the extensibility in XTEAM in the following way. First, the component model is enhanced to include attributes and elements that capture the parameters of a relevant analytic theory. XTEAM currently implements modeling extensions for energy consumption [11], reliability [15], latency, and memory usage analyses as examples. The architect then utilizes 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. This has been accomplished for the four analyses listed above to demonstrate the capability.

The modeling capabilities of XTEAM are described fully in [2]; the remainder of this paper focuses on the design and evaluation of the XTEAM interpreter framework.

4.2. The XTEAM Model Interpreter Framework

The XTEAM model interpreter framework implements a semantic mapping between a flexible and extensible domain-independent component model and a simulation model. As described above, XTEAM provides a modeling environment built on top of GME that allows an architect to extend the XTEAM component model by defining new elements, attributes, and constraints that (1) tailor the model to a specific component technology, such as the OSGi platform [8] or CORBA Component Model (CCM) [9] and (2) allow the inclusion of the parameters required by an analytic theory.

When invoked by an architect, the XTEAM interpreter framework traverses the architectural model, building up a discrete event simulation model in the process. The interpreter framework maps components and connectors to discrete event constructs, such as atomic models and static digraphs. The FSP-based behavioral specifications are

translated into the state transition functions employed by the discrete event simulation engine. The interpreter framework also creates discrete event entities that represent various system resources, such as threads.

The interpreter framework employs the Strategy pattern [17] to enable an architect to implement domain-specific extensions, as depicted in Figure 4. The Strategy pattern allows a set of related algorithms to be transparently interchanged within different contexts. The different algorithms are abstracted by a common interface. In the XTEAM interpreter framework, each algorithm generates code that encapsulates logic to realize a particular analytic theory. For example, the logic may implement equations that calculate non-functional system properties based on the parameters defined in the model and equations defined by the theory. The algorithms are invoked at specific times during the interpretation process, such that the code generated by those algorithms will be invoked when various events occur during an actual simulation run. These events include a component receiving or sending data, invoking an interface, initiating or completing a task, *etc*.

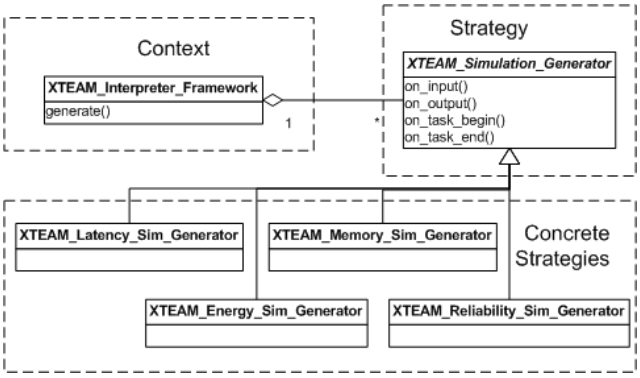

Figure 4. High-level design of the XTEAM model interpreter framework.

To illustrate the process used by an architect to realize a given analysis using an interpreter framework, we now describe the implementation of the XTEAM energy consumption simulator. The energy consumption estimation technique described in [11] provides a mechanism for estimating software energy consumption at the level of software architecture. The estimation technique provides equations that enable the calculation of energy costs based on a number of parameters, including data sizes and values, characteristics of the hardware hosts, and network bandwidth. Energy is used by the system whenever either (1) data is transmitted over the network or (2) the software is required to perform computation. Consequently, the equations defined by the energy consumption estimation technique were inserted into the Strategy methods corresponding to the sending and receiving of data and the invocation of an interface. The equations calculate the energy cost of a given data transmission or computation based on the parameters defined in the model, and record these values for later examination by architects. The implementation of the other XTEAM simulation generators follows the same approach.

5. Evaluation

This section describes how XTEAM was utilized to provide a key non-functional analysis that ultimately guided the choice of architectural style for a given application. Furthermore, this section compares the predictions of system properties made by XTEAM with measured values taken from the executing system. In our experiments, XTEAM simulations were shown to produce predicted values for system energy consumption that fell within 10% of the observed values, and guided the software architects to the correct choice of architectural style. This result illustrates the utility of XTEAM in making fundamental architectural decisions early in the development cycle.

5.1. Application Scenario

To illustrate the importance of non-functional analysis, consider the MIDAS family of sensor network applications [18]. An instance of MIDAS consists of sensors, gateways, hubs, and PDAs. Sensor nodes collect data about the environment and transmit that data to gateways over wireless links. Gateways manage groups of sensors, aggregate and fuse sensor data, and forward the fused data to hubs. Hubs analyze fused sensor data, generate visualizations of the data, and provide a user interface for configuring and managing the system. PDAs provide mobile access to the data visualizations and system management capabilities. The distributed software system, which is described and analyzed in this section, is implemented on top of a lightweight, component-based middleware platform, called Prism-MW [10], which enables architecture-based development of distributed applications in embedded and pervasive environments.

The MIDAS system is subject to a number of non-functional requirements. For this evaluation, we analyzed an instance of MIDAS that provides building monitoring services, such as intrusion and fire detection. In this scenario, the MIDAS hardware devices are not connected to a continuous power supply, but instead run on battery power. Therefore, the system's efficiency with respect to energy consumption has a critical impact on the longevity of the system services.

One of the most influential factors in the system's overall energy consumption is the cost of sending and receiving data over the wireless network. As a result, the type and frequency of interactions between software components has a major impact on the system's energy usage. Component interactions are, in turn, governed to a large extent by the choice of an architectural style. It was crucial, therefore, that the MIDAS system employ an energy efficient architectural style, while still fulfilling numerous other functional and non-functional requirements. Based on the system requirements as a whole, two candidate architectural styles, client-server and publish-subscribe, were selected. Two models of the MIDAS security application — each using one of the candidate styles — were then created in XTEAM and compared with respect to energy consumption.

5.2. Modeling and Analysis

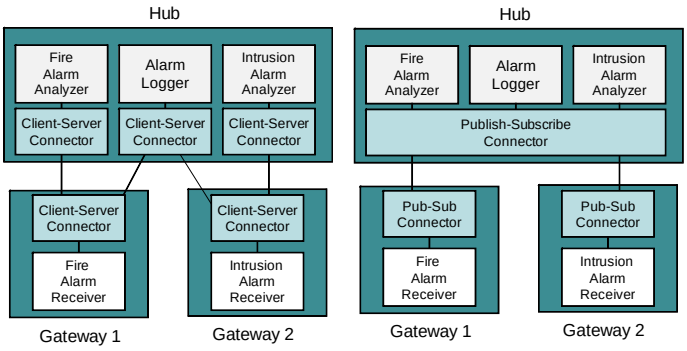

Figure 5. A subset of MIDAS designed in client-server (left) and publish-subscribe (right) styles.

Figure 5 shows the same subset of MIDAS designed using the two styles. The *FireAlarmReceiver* and *IntrusionAlarmReceiver* components deployed on the gateways translate, aggregate and fuse alarm events received from the sensors periodically, and propagate them to the components deployed on the hub. The *Analyzer* components deployed on the hub analyze the alarm data and determine whether there is actually a fire or intrusion. If the *FireAlarmAnalyzer* (*IntrusionAlarmAnlyzer*) component concludes that there is a fire (intrusion), it transmits a sensor-activation message to the *FireAlarmReceiver* (*IntrusionAlarmReceiver*) component, which in turn sends an activation signal to all the fire (intrusion) sensors.

For the client-server architecture, we modeled the behavior of client-server connectors based on a request-response protocol. Client-server connectors are frequently implemented as middleware stubs and skeletons. The behavior of the application components was also modeled according to the above scenario: the *Receiver* components act as clients and invoke interfaces on the *Analyzer* and *Logger* components via their local client-server connectors. The client-server connector on a gateway then transmits a request event (from its local *Receiver* component) to the *Analyzer* and *Logger* components separately, which indicates that each request requires two transmissions on each gateway.

For the publish-subscribe architecture, we modeled the behavior of connectors based on a typical publish-subscribe interaction protocol. For example, the *FireAlarmAnalyzer* sends a message to the connector that requests a subscription to fire alarm events. When a component, such as *FireAlarmReceiver*, publishes a fire alarm event, the connector routes the event to each subscribed component. The behavior model of each component is essentially the same as that in the client-server architecture, except that components publish and subscribe to events. For instance, the *FireAlarmAnalyzer* has the same behavior for processing fire alarm events as in the client-server architecture, but includes additional logic that transmits event subscription requests to the publish-subscribe connector. In this architecture, the publish-subscribe connector can optimize the transmission of events based on the location of publishers and subscribers (as is done in the publish-subscribe service implementations of widely-used middleware platforms [16]). Therefore, compared with the client-server architecture, the publish-subscribe architecture may require fewer events to be sent over the wireless network, but incurs the additional overhead of managing lists of publishers and subscribers.

XTEAM requires the following host-specific energy costs to analyze the above two architectural styles with respect to their energy costs:

1. *The communication energy cost on each host due to transmitting and receiving data over the network.* Previous research [3] has shown that the energy consumption of wireless communication is directly proportional to the size of transmitted and received data and can be expressed as a linear equation with the size of data exchanged. Our energy estimation tool [11] details the steps for determining the communication energy cost on a specific hardware platform.
2. *The energy consumption on each host due to processing a subscription and retrieving a set of subscribers for a published event.* These energy costs can be determined by leveraging the measurement setup described in [11].

Note that we do not need to consider the computational energy cost of most component event processing (*e.g.*, the energy consumption of the *FireAlarmAnalyzer* due to processing a fire alarm) in comparing the energy costs of the candidate architectural styles because this cost is the same for both styles.

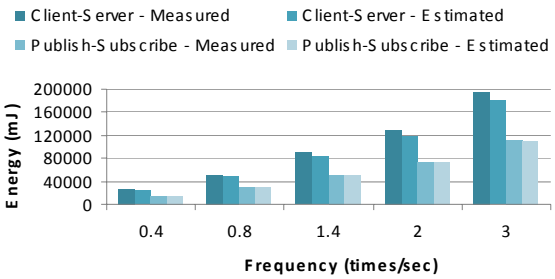

Figure 6. Comparison of the energy consumption of MIDAS using the client-server and publish-subscribe styles.

Once the architectural model was parameterized with the above information, the XTEAM energy consumption simulation generator was invoked. XTEAM allows simulations to include various stochastic behaviors, such as the frequency of client requests, the probability of cache misses, or the

sizes and values of data. In this case, the timing and size of events was determined stochastically, and four different average rates of event transmission were simulated. The results of the energy consumption simulation are shown in Figure 6. The XTEAM analysis predicted that utilizing the publish-subscribe style would result in significant energy savings. The next section describes how we verified the correctness of this result.

5.3. Verification

In order to determine the accuracy of the energy consumption estimates made by XTEAM, we need to know the *actual* energy consumption of the distributed software system. To this end, we used a digital multimeter and the experimental setup described in [11]. The MIDAS application discussed in Section 6.1 was implemented using both the client-server and publish-subscribe styles on top of Prism-MW. We used the same average frequencies and sizes of alarm events as those simulated in XTEAM, measured the energy consumption on each host, and finally calculated the software system's overall energy consumption by summing up the three hosts' energy costs.

For each candidate style, we compared the actual overall energy consumption with the energy consumption estimates generated by XTEAM for different rates of event transmission. As shown in Figure 6, the predicted energy consumption fell within 10% of the measured energy consumption in all the scenarios analyzed. In addition, the publish-subscribe style was determined to be much more energy-efficient for this scenario because (1) the publish-subscribe style requires fewer events to be sent over the wireless network and (2) the energy savings obtained by the reduced data exchange over the network exceeds the energy overhead due to processing subscription requests and retrieving the set of subscribers for each published event.

This result demonstrates that although the architectural models cannot be parameterized with perfect accuracy — especially when XTEAM is being applied early in the architectural development process — the predictions provided by XTEAM are accurate enough to enable architects to successfully determine trade-offs between relatively course-grained design alternatives, such as the choice of architectural style.

5.4. Limitations

Although the experiment described above establishes both the quality and utility of XTEAM predictions of non-functional properties, there are several limitations to the applicability of our approach. First, XTEAM's model interpreter framework relies on the ability to measure and quantify a given system property. Properties that are difficult or impossible to quantify, such as usability [23], cannot be predicted using XTEAM's discrete event simulation-based analysis. Second, the properties of a component assembly must be derivable from a composition of (1) the properties of individual components, (2) the overall software architecture, and (3) the system's usage profile. For example, properties that depend on the environment in which the system is used are not amenable to analysis in XTEAM. An example of such a property is security [22], which is heavily impacted by characteristics of the computing infrastructure (*e.g.*, network and operating system) and external, human factors. While these types of concerns can be added to XTEAM's modeling language through metamodel extensions, XTEAM's focus is on software architecture, and consequently the corresponding extensions to the model interpreter framework would likely require significant effort. Finally, XTEAM is intended to predict system run-time properties rather than lifecycle properties related to construction

activities. For example, the maintainability of a system is derivable from its software architecture [21], but is not compatible with XTEAM's dynamic analysis. The non-functional property classification scheme described in [22] provides a good mechanism for determining what properties can be effectively analyzed by XTEAM.

6. Related Work

This section establishes the broader context in which our work resides. First, we discuss a conceptual framework that provides a basis for the ideas discussed in this paper. Second, we describe a representative approach to component-based architectures.

6.1. Prediction-Enabled Component Technology

Predication-enabled component technology (PECT) is a proposed framework for the integration of component technologies and analysis technologies [1]. A PECT can be used to determine the emergent properties of a highly complex assembly of software components when certain characteristics of the individual components can be certified. PECT relies on component design tools and run-time environments to enforce the assumptions required by each analysis technique applied to the system.

A PECT instance includes a construction framework and one or more reasoning frameworks [12]. The *construction framework* constitutes the design and implementation facilities, such as modeling environments and code generators, that are used to develop a component-based system. The construction framework relies on an *abstract component technology*, or ACT, to represent component models and run-time platforms. Software architecture and design models, or *constructive models*, that conform to the ACT are created in the construction framework. A *reasoning framework*, on the other hand, constitutes the analysis facilities to be applied to the system. A reasoning framework applies system analysis techniques, or *property theories*, through the use of an *analysis environment*. Discrete event simulators and fault-tree analysis tools are examples of analysis environments. *Interpretations* transform constructive models into analysis models. Component characteristics, which constitute the parameters of property theories, are codified in a component's *analytic interface*. This interface is leveraged by the reasoning framework to apply a system-wide analysis of non-functional properties.

PECT leverages many of the core concepts of MDE to support analysis of the non-functional properties of large-scale component assemblies. PECT establishes an intuitive way of organizing and relating the elements of component technologies and analysis technologies, and outlines a strategy for integrating component models and analysis models that leverages their complementary characteristics. For these reasons, we believe PECT provides a useful conceptual framework for additional research in the modeling and analysis of component-based systems. However, PECT does not address the fundamental challenge described in Section 2; that is, it does not help a software architect discover and realize any particular domain-specific model interpretation.

6.2. CALM and Cadena

Cadena is an extensible environment for the modeling and development of component-based architectures [14]. The Cadena Architecture Language with Metamodeling (CALM) supports the specification of platform- and domain-specific component models, which are leveraged by Cadena to provide automated enforcement of architectural con-

straints. In this way, CALM and Cadena provide a modeling environment that can be readily integrated with a wide variety of component technologies.

CALM is based on a three-tiered typing system. At the *style* tier, an architect defines the kinds of components, connectors, and interfaces that exist within a particular component model or architectural style. The style tier is essentially a metamodeling layer that defines a language of architectural constructs. At the *module* tier, the component and interface types that may exist within a specific application architecture are declared. Finally, at the *scenario* tier, component types are instantiated into a particular configuration or assembly. At each tier, Cadena automatically enforces the constraints imposed by the type system defined at the tier above.

The modeling capabilities of CALM and Cadena provide a powerful and intuitive mechanism for creating application architectures that conform to domain-specific component models. Cadena also provides an integrated model-checking infrastructure, Bogor, which enables automatic verification of the logical properties of a system, such as event sequencing. However, Cadena provides little support for the implementation of additional, domain-specific types of non-functional analysis. Thus, Cadena also forces architects to develop model interpreters from scratch in most cases.

7. Conclusions

This paper presented an approach to the construction of analytic frameworks that enable the prediction of the non-functional properties of component-based systems. Such frameworks allow the rapid construction of model interpreters, which is one of the most complex and difficult activities in the model-driven engineering paradigm. In order to achieve this result, model interpreter frameworks must make several important assumptions about the models to which they are applied, and fulfill a set of design requirements. This paper also demonstrated the process of constructing, utilizing, and validating a model interpreter framework using an example.

Our ongoing work in this area is two-fold. First, we are constructing additional interpreter frameworks and integrating them into the XTEAM environment, in order to more clearly define the scope of applicability of the approach described in this paper. For example, we hope to identify a small set of analysis models for which interpreter frameworks can be constructed that will provide broad coverage of the analysis techniques present in the software architecture literature. Second, we are continuing to apply the current XTEAM interpreter framework in several R&D contexts. For example, we are utilizing XTEAM for the continuing development of the MIDAS family of applications and conducting a rigorous analysis of the impact of styles on non-functional properties.

8. References

[1] S.A. Hissam, J.A. Stafford, K.C. Wallnau (2002). Packaging Predictable Assembly. In *Proc. of the ACM Working Conf. on Component Deployment*, pp. 108-124.

[2] G. Edwards, et al. (2007). Scenario-Driven Dynamic Analysis of Distributed Architectures. In *Proc. of Fundamental Approaches to Software Engineering*.

[3] L.M. Feeney, et. al. (2001). Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proc. of IEEE INFOCOM*.

[4] D.C. Schmidt (2006). Model-Driven Engineering. *IEEE Computer*, pp. 41 - 47.

[5] A. Ledeczi, et al. (2001). On metamodel composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications*, pp. 756 - 760.

[6] M. Fayad, D. C. Schmidt (1997). Object-oriented application frameworks. *Communications of the ACM*, pp. 32 - 38.

[7] The Generic Modeling Environment. http://www.isis.vanderbilt.edu/projects/gme/

[8] OSGi: The Open Services Gateway Initiaive. http://www.osgi.org/

[9] CCM: The Corba Component Model. http://www.omg.org/

[10]S. Malek, M. Mikic-Rakic, et al. (2005). A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Trans. on Soft. Engineering*.

[11]C. Seo, et al. (2006). Energy Consumption Framework for Distributed Java-Based Software Systems. Tech. Report USC-CSE-2006-604, Univ. of Southern California.

[12]K. Wallnau (2003). Volume III: A Technology for Predictable Assembly from Certifiable Components. Tech. Report CMU/SEI-2003-TR-009, Software Eng. Institute.

[13]M. Woodside. Tutorial Introduction to Layered Modeling of Software Performance. Carleton University, http://sce.carleton.ca/rads.

[14]A. Childs, et al. (2006). CALM and Cadena: Metamodeling for Component-Based Product-Line Development. *IEEE Computer*.

[15]R. Roshandel, S. Banerjee, L. Cheung, N. Medvidovic, and L. Golubchik (2006). Estimating Software Component Reliability by Leveraging Architectural Models. In *Proc. of the 28th International Conference on Software Engineering*.

[16]G. Edwards, D. C. Schmidt, A. Gokhale, B. Natarajan (2004). Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems. *Proc. of the 42nd Annual ACM Southeast Conference*.

[17]E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[18]S. Malek, C. Seo, et al. (2007). Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. *Proc. of the 29th International Conference on Software Engineering (ICSE 2007).*

[19]E. Dashofy, et al. (2002). An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proc. of the 24th International Conference on Software Engineering*, pp. 266 - 276.

[20]J. Magee, et al. (1999). Behaviour Analysis of Software Architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture*, pp. 35 - 50.

[21]N. Lassing, et al. (2002). Experiences with ALMA: Architecture-Level Modifiability Analysis. *Journal of systems and software*, Elsevier, pp. 47-57.

[22]I. Crnkovic, et al. (2005). Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. Architecting Dependable Systems III, Springer, LNCS 3549, Editor(s): R. de Lemos et al., pp. 257-278.

[23]Eelke Folmer, et al. (2004). Software Architecture Analysis of Usability. In *Proc. of the IFIP Working Conf. on Eng. for Human-Computer Interaction*, pp. 321-339.