Journal of Universal Computer Science, vol. 14, no. 8 (2008), 1182-1206
submitted: 15/1/07, accepted: 25/3/08, appeared: 28/4/08 © J.UCS

Model Interpreter Frameworks: A Foundation for the Analysis of Domain-Specific Software Architectures

George Edwards
(University of Southern California, Los Angeles, USA
gedwards@usc.edu)

Chiyoung Seo
(University of Southern California, Los Angeles, USA
cseo@usc.edu)

Nenad Medvidovic
(University of Southern California, Los Angeles, USA
neno@usc.edu)

Abstract: Prediction of the quality attributes 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 complex and difficult task. This paper (1) describes a methodology for performing automated analysis of architectural models that simplifies the development of model interpreters and enables effective reuse of interpreter logic, and (2) demonstrates how a framework that utilizes the methodology can be designed, implemented, utilized, and evaluated.[1]

Key Words: Software architecture, model-driven engineering, component-based systems
Category: D.2.2, D.2.10, D.2.11

1 Introduction

Modern day component technology provides software architects with powerful mechanisms for designing, implementing, deploying, and evolving large-scale distributed systems. Component-based software engineering, as a whole, encompasses a number of different elements and paradigms. First, component-based *architectures* utilize high-level design abstractions to help engineers reason about large-scale software sys-

1.This paper is a significant revision and extension of G. Edwards, et al., Construction of Analytic Frameworks for Component-Based Architectures, Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), August 2007.

tems more effectively. Second, component-based *development* enables highly effective strategies for reuse and integration of existing software (in the form of off-the-shelf components and product-line architectures, for example). Finally, component-based *middleware* includes development environments and run-time platforms that support the implementation of distributed systems by providing high-level programming constructs and numerous services, such as packaging, configuration, and deployment.

Component-based software engineering also offers a basis for the construction of analysis models that enable the discovery and prediction of critical properties, or *quality attributes*, of a software system, 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 because the constraints on the design space provided by a component-based architecture make the construction and solution of analysis models feasible. Furthermore, such constraints can be enforced and certified during development and execution by component-based middleware platforms. This paper outlines a novel strategy for conducting this type of automated prediction of the quality attributes of component-based systems.

To effectively analyze the quality attributes of a component-based system, methods and tools are needed that support the integration of component technologies and analysis technologies [Hissam 2002]. 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. For example, component models define the types of interfaces that a component may expose, the patterns of interaction between components and their run-time environments, and so on. Component technologies (such as Java Enterprise Edition) 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 [Woodside 2002], 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 in the model accepts requests from a single queue. Such an assumption can be enforced by a component middleware platform both at system construction-time and at run-time. Therefore, component technologies and analysis technologies are well-suited to integration because the use of a component technology for system construction can be leveraged to ensure that the assumptions required by an analysis technology are satisfied. This increases confidence that predictions produced by an analysis technology operating on design models will remain valid for the implemented system.

Unfortunately, the integration of component and analysis technologies is anything but straightforward in practice. Component-based systems are generally specified using high-level design languages and implemented using high-level programming constructs that emphasize abstraction and flexibility, while analysis techniques, on the other hand, 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, each intended for a different purpose. 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.

The model-driven engineering (MDE) paradigm [Schmidt 2006] offers an attractive strategy for achieving the integration of component and analysis technologies. 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 particular modeling language, and can be easily modified, adapted, composed, enhanced, and evolved [Ledeczi 2001]. In this way, MDE offers a straightforward and intuitive way to incorporate the parameters of an analytic theory into both widely-used general purpose component models 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, and so on. Model interpreters can be used to implement semantic mappings, or transformations, between the high-level design models amenable to architectural reasoning and the low-level analysis models amenable to rigorous prediction of component assembly properties. Figure 1 delineates the main MDE concepts and processes.

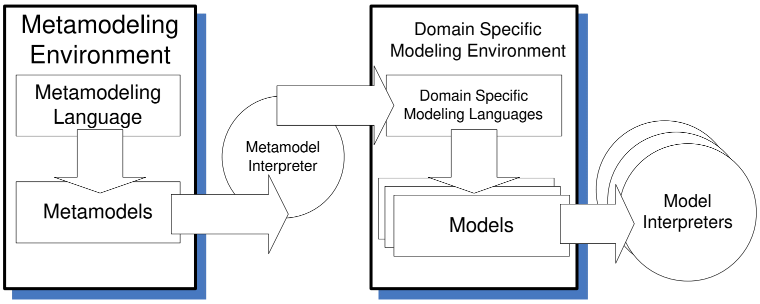

Figure 1:High-level view of the model-driven engineering (MDE) process.

We postulate that the respective benefits of architecture-centric development, model-driven engineering, and component-based middleware can be leveraged to construct a methodology for the automated analysis of the quality attributes of software systems. The remainder of this paper describes such a methodology. The core elements of the methodology are (1) an abstract component technology and (2) model interpreter frameworks. An *abstract component technology* (ACT) is an extensible architecture description language (ADL) and component model that can be adapted to the needs of a particular domain and implementation platform [Wallnau 2003]. A *model interpreter framework* (MIF) is an extensible infrastructure for implementing analysis techniques that can be used to rapidly construct analyzable models from domain-specific architectures. Our approach offers several important advantages over other methodologies, most notably (1) the ability to systematically implement domain-specific analysis techniques without having to implement complex model transformations, and (2) the ability to apply off-the-shelf analysis capabilities to domain- and platform-specific models.

To illustrate the approach, we describe our implementation of the eXtensible Toolchain for Evaluation of Architectural Models (XTEAM), an integrated modeling, analysis, and synthesis environment. Our evaluation of XTEAM demonstrates that (1) implementing an architectural analysis technique using a model interpreter framework is substantially simpler than constructing a stand-alone interpreter from scratch, (2) an interpreter framework can provide accurate predictions of the quality attributes of complex assemblies of off-the-shelf components, and (3) 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. Sections 3 and 4 describe our approach, while Section 5 discusses in detail our implementation of the XTEAM abstract component technology and model interpreter framework. Section 6 evaluates the framework and illustrates its benefits through the use of a case study. Overviews of related work and conclusions round out the paper.

2 Motivation

This section summarizes the current technology trends and state-of-the-art in architectural modeling and analysis, enumerates the shortcomings of current practices, and briefly outlines how the methodology proposed in this paper addresses these shortcomings and represents an improvement over current approaches.

Increasingly, the architectural models of large-scale distributed systems are incorporating domain- and platform-specific elements. Rather than relying on static, general-purpose languages, like UML 1.x, architects are constructing models using domain-specific modeling languages (DSMLs) that are customized precisely for the needs of a particular enterprise or project. The codification of domain concepts as first-class modeling constructs allows developers to build models using the most concise and intuitive syntax. This simplifies the modeling process and alleviates the tedious specification of low-level details. The use of DSMLs along with model transformation engines that perform system analysis and synthesis has been termed model-driven engineering (MDE).

Domain-specificity confers a number of benefits, but also incurs some drawbacks. First, the responsibility for producing the language specification or metamodel falls on application developers and architects, rather than specialized tool developers and language experts. The creation of semantically powerful, flexible, and intuitive modeling languages, even with the benefit of metamodeling environments, is inherently challenging. Furthermore, DSML development requires both domain expertise and metamodeling expertise; that is, a language developer must command a thorough understanding of the central and elemental concepts in the target domain and must be adept in the mechanisms for codification of those concepts. This poses an obstacle because domain experts are commonly not metamodeling experts, and vice-versa.

Second, because domain-specific languages are not standardized, tools that leverage models for common functions like analysis and program synthesis (code generation) must be custom-built for each language. This requires system architects and developers to become tool developers – rather than merely tool users – to achieve integrated design analysis and program synthesis. Additionally, organizations want to develop with third-party, commercially-supported tools to reduce risk and cost. The implementation of such tools, called model interpreters, frequently requires a significant investment of resources. Modeling environments provide interfaces for model interpreters to access and manipulate the information contained in models, but in many cases, a complex semantic mapping between languages is required that is difficult to define and implement. For example, such a mapping is required to transform architecture-based models, which are at a very high level of abstraction, into executable simulations or code, which are at a much lower level. 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 suggests a strategy for simplifying and improving this process.

Consider a large-scale development project for a software-intensive system. 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 devices and middleware facilities) 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 layered queuing network (LQN) model [Woodside 2002]. Applying the normal 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 architecture (*i.e.*, the assignment of software components to hardware hosts) is further refined, questions arise about how deploying certain components to mobile hosts, which have a finite battery life, will impact the system's energy consumption. The architecture team is instructed to employ the architecture model in comparing 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 additional model interpreters. For each new interpreter, the team must:

1. Find a computational theory that derives the relevant properties from a system model.
2. Determine the syntax and semantics of the modeling constructs on which the computational theory operates.

3. Discover the semantic relationships between the constructs present in the architectural models and those present in the analysis models.
4. Determine the compatibility between the assumptions and constraints of the architectural models and the analysis models, and resolve conflicts between the two.
5. Implement a model interpreter that executes a sequence of operations to transform an architectural model into an analysis model.
6. Verify the correctness of the transformation implemented by the interpreter.

The overhead and maintenance costs associated with employing the MDE process, as outlined above, make it a less attractive development strategy for small- and medium-scale software systems. For example, MDE has experienced widespread industry adoption in large-scale defense and aerospace programs, but relatively limited adoption in small business and desktop application development.

This paper demonstrates how a methodology that leverages an abstract component technology combined with a model interpreter framework allows an architecture team to (1) avoid the difficult task of inventing domain-specific languages from scratch, and (2) perform the above process only once for a broad family of analysis techniques, rather than repeating the process for each analysis technique. The methodology can significantly improve the utility and appeal of domain-specific architectural development, as the definition of domain-specific languages and the construction of model interpreters constitute the primary activities in the MDE process.

3 Abstract Component Technology

The first step in leveraging the MDE approach for the evaluation of the quality attributes of software architectures is to construct a metamodel that defines a DSML. However, as mentioned above, the creation of semantically powerful, flexible, and intuitive modeling languages is non-trivial; in fact, it requires a great deal of expertise in both metamodeling and the target domain. To overcome this challenge, we advocate an approach that avoids the creation of languages from scratch. Instead, we rely on the construction of an *abstract component technology* (ACT) to simplify language development. This section defines what an ACT is and outlines the benefits of using an ACT. In Section 5, we illustrate the role an ACT plays in an MDE toolchain and demonstrate how it can be used through a detailed discussion of the eXtensible Toolchain for Evaluation of Architectural Models (XTEAM), which implements the approach described in this paper.

3.1 Definition

There exists a set of abstractions that are common to architectural models in a wide variety of domains. Furthermore, these “standard” architectural elements (components, connectors, interfaces, *etc.*) are provided as implementation constructs in a wide variety of middleware component models. The software elements represented by each of these constructs exhibit some capabilities, constraints, and properties that remain valid across domains and platforms. At the same time, other capabilities, constraints, and properties may differ from one domain or platform to another. For example, components in most domains and platforms can be described as independently deployable software elements that interact with external entities only through well-defined interfaces. However, the types of interfaces that a component may expose and access varies from one platform to another. For example, in the CORBA Component Model (CCM), components interact via *provided* and *required* interfaces, which are method-based, and *event sources* and *event sinks*, which are message-based [see CCM]. In the OSGi platform, on the other hand, components (called *bundles*) may register *producer* and *consumer services*, which are *wired* together by the middleware [see OSGi]. The capabilities and constraints associated with OSGi bundle interfaces are different than those associated with CCM component interfaces.

An abstract component technology (ACT) is a domain- and platform-independent component model. An ACT defines a highly generic, but also highly extensible, modeling language for component-based software architectures. An ACT defines the standard architectural elements only in terms of their platform-independent properties and leaves undefined those properties that vary from one platform to another. For example, an ACT will define a *Component* type which will include interface definitions. The ACT should allow arbitrary component interfaces to be specified (in terms of their data types, synchronism, *etc.*), but require that all inter-component interactions take place via defined interfaces. This allows an ACT to serve as the basis for construction of architectural models for a wide variety of domains and platforms. The ACT can then be extended and enhanced to capture the capabilities, constraints, and properties of the architectural elements present in a particular domain or platform, or incorporate additional domain- and platform-specific constructs. In the example given, the ACT could be extended to constrain the allowed interface definitions to synchronous, method-based interactions, if such a constraint was enforced by the target middleware platform.

3.2 Composition and Extension

The metamodeling mechanisms provided by MDE technologies enable the construction and manipulation of ACTs. An ACT metamodel can either be defined from scratch, as is the case with the Pin component technology [Hissam 2005], or it can be created through composition of the metamodels [Ledeczi 2001] of multiple, general purpose architectural languages, as is the case with the XTEAM ACT (described in Section 5). Once the metamodel for an ACT has been created, the ACT can be extended as needed for a given application domain by modifying the metamodel. Thus, the composition and enhancement of ACTs is achieved through composition and enhancement of their corresponding metamodels.

There are generally two reasons for extending an ACT. First, an architect may wish to include model parameters that are required by a domain-specific analytic technique. This allows the architectural model to be transformed into an analysis model that can be evaluated with respect to a quality attribute. The set of analytic parameters associated with an architectural element are collectively referred to as the element's *analytic interface* [Hissam 2002]. For example, a performance analysis might require that a component's behavioral definition include the number of available threads or the queuing discipline applied to incoming requests. To capture analytic interfaces, the parameters required by a particular analytic theory can be grouped together as attributes of an abstract *base type* for ACT elements. This ensures that analytic interfaces can be selectively used and composed in whatever combination is required. In the above example, an abstract type named *PerformanceAnalyzable* could be defined with *ThreadPoolSize* and *QueuingDiscipline* attributes. To apply the performance analysis to architectural models, an architect defines an inheritance relationship within the metamodel that specifies that the *Component* type implements the *PerformanceAnalyzable* analytic interface.

Second, an architect may wish to include platform-specific constructs that reflect the implementation facilities provided by a middleware. This enables the automatic generation of a variety of implementation artifacts, such as application code, middleware configuration files, deployment descriptors, and so on [Gokhale 2005]. For example, some CORBA implementations allow a set of quality-of-service (QoS) parameters to be specified for component interfaces, such as event priorities and dispatching mechanisms. To create domain-specific constructs, new elements can be created that are *subclasses* of ACT elements. Platform-specific attributes and constraints can then be defined for these derived elements. This ensures that model interpreter frameworks based on the ACT (as will be described in Section 4) will handle platform constructs correctly (provided the assumptions listed in Section 4 are satisfied). Also, all platform-specific constructs will inherit any analytic interfaces defined for their respective ACT base types. In the example given, a new type named *CORBAEventSink* could be defined. This type would have constraints applied to it that would require it to be a message-based interface that only receives incoming data. The *CORBAEventSink* type could also have *Priority* and *Dispatching* attributes that specify the priorities of incoming events and the dispatching mechanism used to handle them. Architectural models employing the *CORBAEventSink* type can then be easily used to automatically generate QoS configuration files required by the underlying middleware.

3.3 Benefits

The use of an ACT allows existing notations and languages to be reused to the greatest extent possible. Only incremental additions to the language are created as needed to capture platform-specific concerns and enable specific architectural analysis techniques. This is important for two reasons. First, the reuse of an ACT reduces the burden of language development on software architects, allowing them to focus on architecture (rather than modeling language) development. For example, an ACT might define a particular formalism for modeling component behavior. Using the mechanisms described above, an architect can create new platform-specific component types and model their behavior without having to redefine a behavioral formalism. Second, as we will demonstrate in the next section, the utilization of an ACT permits the reuse of common tool infrastructures across development projects and domains. Different types of system properties are relevant within different application domains. Analysis technologies, such as discrete event simulators and model checkers, can determine the quality attributes of a system model, but each type of analysis requires that certain information be captured in the system model. A general-purpose modeling language, therefore, likely cannot capture the analytic parameters necessary for a domain-specific analysis to be applied. On the other hand, analysis tools built for domain-specific languages cannot be reused across other domains. An ACT strikes a balance between these two extremes: it exploits commonality among domains, allowing tool reuse, while permitting domain-specific extension, allowing rigorous analysis of quality attributes.

4 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 (as defined by an abstract component technology) and a widely applicable analytic representation to construct a *model interpreter framework* that abstracts away 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 objectives of an interpreter framework design 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.

4.1 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 (MIF) 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, a MIF is not a library of functions; rather, it is an active component that can be extended and enhanced in specific, predefined ways. Furthermore, a MIF 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 elements 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 MIF 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 MIF enables a family of analytic techniques to be applied to a domain-specific architectural 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 defined by an ACT. The domain-specific elements of the model are the ACT extensions that capture parameters of a relevant analytic theory, plus any additional domain- and platform-specific extensions and constraints. In this way, a MIF is heavily dependent on the definition and use of an ACT. The MIF abstracts the semantic mapping from architectural constructs to analysis constructs, while providing the extensibility to accommodate the logic that measures and records system 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.

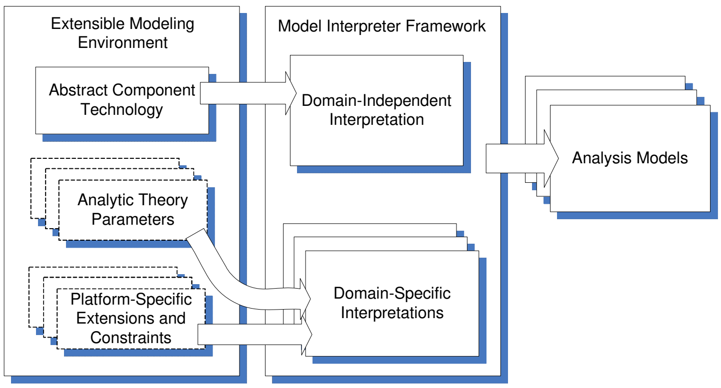

Figure 2:The role of an abstract component technology and a model interpreter framework in the analysis of component-based architectures.

4.2 Assumptions and Design Objectives

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 design objectives in order to be effective. In this subsection, we enumerate these assumptions and objectives and describe their consequences in terms of MIFs in general. We also describe, for each assumption and objective, the specific implications for MIFs that provide analysis of the quality attributes of application architectures.

4.2.1 Assumptions

Assumption 1. System models contain domain-independent elements that are sufficient to implement an interpretation. MIFs 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 can be satisfied through the definition of an ACT.

Assumption 2. The interpretation of domain-independent elements is not dependent on the interpretation of domain-specific elements. The implementor of a MIF 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 MIF 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 MIF 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 Hissam et al. [Hissam 2002]. 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. When a component model is weakened, it implies that the analytic theory can be applied to a larger class of systems than those described by the component model, and some constraints are being "thrown out." Therefore, the predictions made by the theory may be less precise or more computationally expensive than those made by a theory that leverages all constraints in the component model. On the other hand, strengthening the component model implies that the analytic theory can only be applied to a subset of systems described by the component model, and constraints must be added. The analytic theory can therefore be used only if a mechanism exists for ensuring that the executing system will abide by these additional constraints; they are not enforced by the middleware platform. Similarly, the assumptions of a analytic theory can be strengthened or weakened in order to reconcile conflicting component constraints.

The consequences of this assumption can be illustrated using an example. As we have previously stated, an ACT likely includes a constraint that requires component interactions to take place via defined interfaces. This implies that a component's internal state cannot change spontaneously without it being modified by the component itself. An analysis technique that determines whether system deadlock is possible likely relies on such an assumption, that is, that component state changes cannot happen arbitrarily, but will occur only in specified ways.

Now consider two hypothetical middleware platforms: in Platform A, components may arbitrarily modify one another's state; in Platform B, a components state may be arbitrarily changed by the underlying operating system. If an architect decides to use Platform A, a conflict exists between assumptions that can be rectified via co-refinement. The architect strengthens the set of constraints associated with the middleware component model, prohibiting components from changing each other's state directly. In this case, the architect must also institute some mechanism to ensure that the eventual implementation obeys this constraint, since the middleware does not enforce it. The deadlock detection analysis can now be safely applied. If the architect decides to use Platform B, however, a conflict exists between assumptions that cannot be remedied because the architect cannot prevent the operating system from changing a component's state. As a result, the architect is prevented from applying the deadlock analysis to his architectural models. However, this is actually a desirable result because the predictions provided by the deadlock detection analysis would have been totally invalid for a system implemented using Platform B. The ACT and MIF, by making constraints explicit, prevent the inadvertent use of invalid analyses.

4.2.2 Design Objectives

Design Objective 1. The model interpreter framework encapsulates the implementation 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 objective, 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.

Design Objective 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. Therefore, while it is possible to create a MIF for the latter types of analysis models, they are not strong candidates for construction of a MIF.

Design Objective 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. Extension points are created through the use of design patterns such as Template Method, Strategy, and Functor [Fayad 1997]. 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.

5 The XTEAM Toolchain

In this section, we describe in detail the design of an abstract component technology and model interpreter framework we implemented as part of the eXtensible Toolchain for Evaluation of Architectural Models (XTEAM) [Edwards 2007]. XTEAM is an environment that leverages the MDE paradigm to provide a reusable infrastructure for realizing domain-specific architectural analyses. 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.

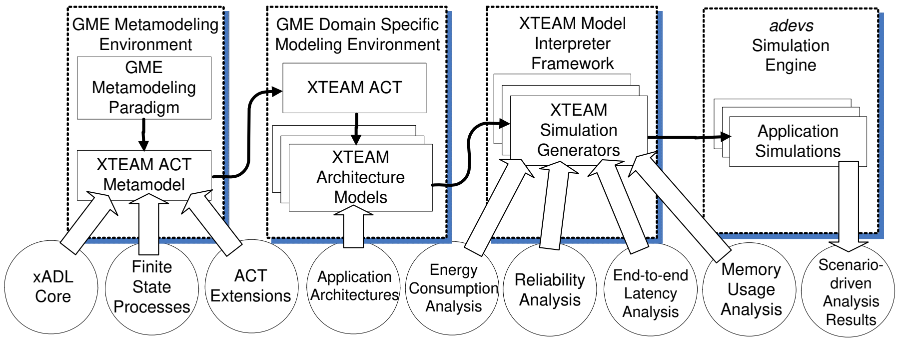

Figure 3:The eXtensible Toolchain for Evaluation of Architectural Models.

An architect takes advantage of the extensibility in XTEAM in the following way. First, the ACT is enhanced to include attributes and elements that capture the parameters of a relevant analytic theory. XTEAM currently implements modeling extensions for energy consumption [Seo 2006], reliability [Roshandel 2006], latency [Woodside 2002], and memory usage analyses as examples. The architect then utilizes the extension mechanisms built into the MIF 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.

5.1 The XTEAM Abstract Component Technology

Using the Generic Modeling Environment (GME) [see GME], we created an abstract component technology by composing the elements of the xADL Structures and Types ADL [Dashofy 2002] and the Finite State Processes (FSP) ADL [Magee 1999]. GME uses this ACT to create a modeling environment in which architectural models that conform to the ACT can be created. The XTEAM environment allows an architect to extend the ACT by defining new elements, attributes, and constraints that (1) tailor the model to a specific component technology, such as the OSGi platform or CORBA Component Model (CCM) and (2) allow the inclusion of the parameters required by an analytic theory.

For illustration, a partial view of the metamodel for the XTEAM ACT is shown in Figure 4, along with a set of analysis and platform ACT extensions. ACT elements are tagged with the stereotype <<ACTElement>>. Analysis and platform extensions are defined according to the process outlined in Section 3.2, and are tagged as <<AnalyticInterface>> or <<PlatformExtension>>, respectively. The *PowerAnalyzable* analytic interface captures system parameters related to power consumption, as defined by a published power consumption analysis technique [Seo 2006]. These parameters are defined as equations that describe distributions of values and include other system variables, such as the size of data objects being exchanged. (Note that the full set of power consumption parameters is not shown in Figure 4.) The *PrismComponent* plat-

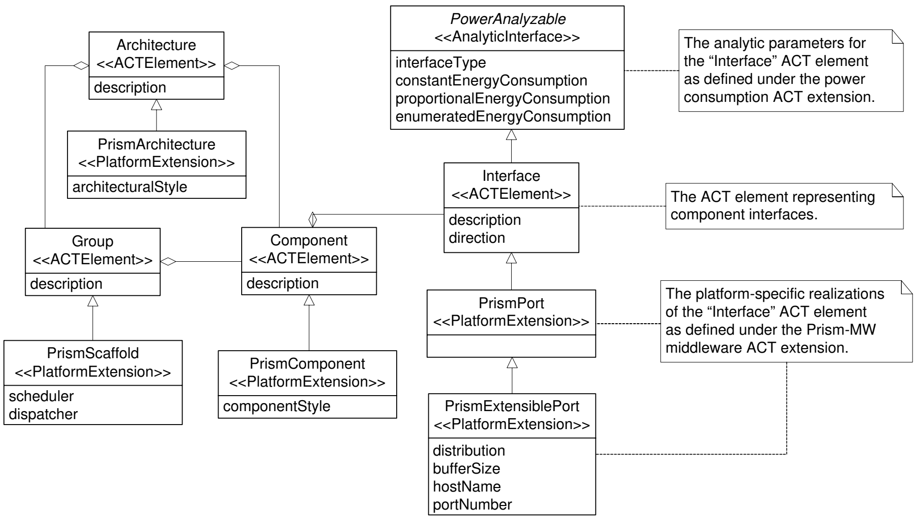

Figure 4:A small subset of the metamodel of the XTEAM abstract component technology (along with a set of analysis and platform extensions).

form extension models the capabilities and constraints of components that run on the Prism-MW middleware platform [Malek 2005]. For example, Prism-MW components can be tagged with a role within a given architectural style; Prism-MW then automatically ensures that the component obeys stylistic constraints associated with that role. This capability is captured by the *componentStyle* attribute.

The combination of xADL and FSP in the XTEAM ACT allowed us to create executable architectural representations. Models conformant to the XTEAM ACT 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 [Schriber 2005]. This semantic mapping is implemented by our model interpreter framework.

5.2 The XTEAM Model Interpreter Framework

XTEAM implements a model interpreter framework that maps the ACT to an analysis model — a discrete event simulation — and implements appropriate extension mechanisms. When invoked by an architect, the XTEAM MIF traverses the architectural model, building up a discrete event simulation model in the process. The MIF 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 MIF also creates discrete event entities that represent various system resources, such as threads.

The interpreter framework employs the Strategy pattern [Gamma 1995] to enable an architect to implement domain-specific extensions, as depicted in Figure 5. 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 XTEAM, a MIF extension is implemented as a Strategy. Each Strategy generates code that encapsulates logic to realize a particular analytic theory. For example, the logic may implement equations that calculate quality attribute metrics based on the parameters defined in the model and equations defined by the theory. The inputs required by a given analytic theory must be determined by the implementer of a MIF

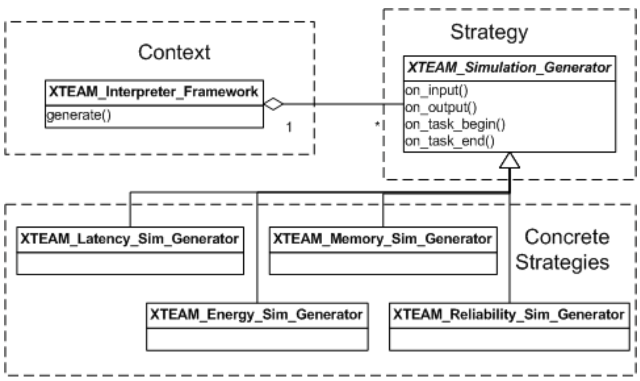

Figure 5:High-level design of the XTEAM model interpreter framework.

extension and extracted from the model using the APIs provided by the modeling environment. The algorithms encapsulated by a Strategy 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.

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 [Seo 2006] 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.

6 Evaluation

This section evaluates the approach described in this paper in two ways. We begin with a discussion of the utility of using an abstract component technology and model interpreter framework, in terms of the savings in effort experienced by software architects adopting the approach. Next, we evaluate our implementation of the approach quantitatively in the context of a specific case study.

6.1 Utility

The ultimate savings in time and effort achieved through the use of our methodology is dependent on the ACT and MIFs being utilized. The design of both ACTs and MIFs involves an important trade-off between flexibility and usability. An ACT or MIF that defines more completely a language or transformation provides more off-the-shelf capabilities, reducing the burden on software architects, but also has less potential for customization. An ACT or MIF that leaves more of the language or transformation undefined provides fewer off-the-shelf capabilities, but is highly customizable. Therefore, ACTs and MIFs can be viewed as existing on a spectrum, from those that implement a nearly complete language or transformation to those that provide only a starting point for implementing a language or transformation. The savings in effort perceived by architects depends on where the ACT and MIF in question lie on this spectrum.

Our experiences with XTEAM provide some indication of how much effort is saved through the use of an ACT and MIF. The XTEAM ACT and MIF lie somewhere in the middle of the spectrum described above. The ACT provides a high degree of flexibility in terms of its structural elements (*i.e.*, the xADL portion of the metamodel), but the behavioral representations (*i.e.*, the FSP portion of the metamodel) are more brittle. Similarly, the MIF provides good flexibility in terms of what and how run-time properties are monitored, but less flexibility in the semantics of event queuing and processing (these portions of the MIF can be customized, but with greater effort). The core transformation logic of the XTEAM MIF constitutes approximately 7000 SLOC, while the latency, reliability, memory usage, and power consumption extensions are implemented in only 325, 800, 300, and 350 SLOC, respectively. This indicates that the implementation size of the transformation logic captured in the MIF is an order of magnitude greater than the implementation size of the logic that must be written by individual software architects.

We do not possess concrete data regarding the amount of resources that organizations adopting the MDE approach are forced to invest in language and interpreter development, so it is difficult to highlight our arguments with exact numbers. However, the overall resources invested in modeling efforts using domain-specific languages on large-scale development programs are substantial. Furthermore, it is apparent that language and interpreter development constitute a significant portion of the work required to implement MDE. Therefore, we argue that it is reasonable to conclude that any approach that simplifies this challenge would result in substantial savings for organizations utilizing MDE.

6.2 Case Study

This section describes how XTEAM was utilized to provide a key quality attribute 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.

6.2.1 Application Scenarios

To illustrate the importance of quality attribute analysis, consider the MIDAS family of sensor network applications [Malek 2007]. 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 [Malek 2005], which enables architecture-based development of distributed applications in embedded and pervasive environments.

The MIDAS system is subject to a number of quality attribute 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.

6.2.2 Modeling and Analysis

Figure 6 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 or IntrusionAlarmAnlyzer component concludes that there is a fire or intrusion, it transmits a sensor activation message to the appropriate Receiver component, which in turn sends an activation signal to all the sensors.

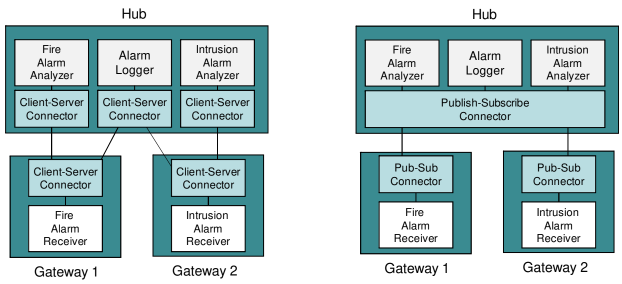

Figure 6:A subset of MIDAS designed in client-server (left) and publish-subscribe (right) styles.

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 [Edwards 2004]). 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 [Feeney 2001] 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 [Seo 2006] 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 [Seo 2006].

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.

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 7. The XTEAM analysis predicted that utilizing the publish-subscribe style would result in significant energy savings. The next subsection describes how we verified the correctness of this result.

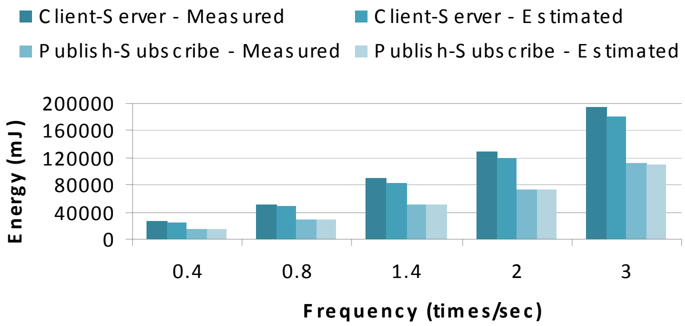

Figure 7:Comparison of the energy consumption of MIDAS using the client-server and publish-subscribe styles.

6.2.3 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 [Seo 2006]. The MIDAS application discussed in Section 6.2.1 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 7, 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.

6.3 Limitations

Although the experiment described above establishes both the quality and utility of XTEAM predictions of quality attributes, there are several limitations to the applicability of XTEAM's discrete event simulation MIF. 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 [Folmer 2004], 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 [Crnkovic 2005], 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's MIF 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 [Lassing 2002], but is not compatible with XTEAM's dynamic analysis. However, these types of analyses can be supported via additional MIFs.

7 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 modeling and analysis component-based architectures.

7.1 Prediction-Enabled Component Technology

Predication-Enabled Component Technology (PECT) is a proposed framework for the integration of component technologies and analysis technologies [Hissam 2002]. 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 [Wallnau 2003]. 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* to represent component models and run-time platforms. The ACT can be either explicitly defined, *e.g.*, by a component metamodel, or implicitly defined via the capabilities of and constraints on components developed using the construction framework. 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 quality attributes.

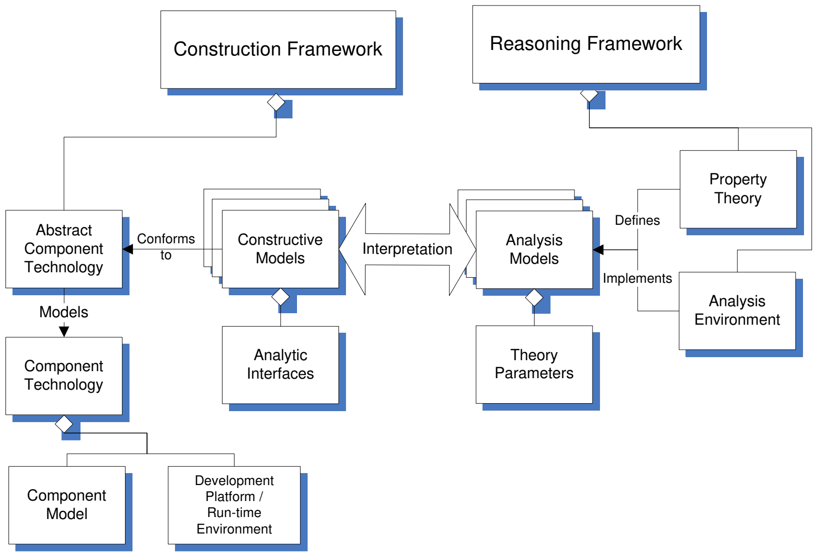

Figure 8:High-level view of the elements of prediction-enabled component technology (PECT).

PECT leverages many of the core concepts of MDE within the context of component-based systems to support analysis of the quality attributes of large-scale component assemblies. PECT establishes a clear and intuitive way of organizing and relating the salient elements and features 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.

7.2 CALM and Cadena

Cadena is an extensible environment for the modeling and development of component-based architectures [Childs 2006]. 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 constraints. 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 quality attribute analysis.

7.3 xADL and ArchStudio

ArchStudio is a software architecture development environment based on the xADL architecture description language. xADL is designed to be highly extensible, and is defined by modular XML schemas; a "core" schema specifies basic architectural structures and types, while "extension" schemas specify new modeling elements as needed. Extension schemas can define the syntax of domain- or platform-specific constructs. ArchStudio includes a number of tools that support the creation, manipulation, and utilization of models that conform to xADL schemas. For example, ArchStudio includes the ArchLight analysis framework, which allows model checkers to be applied to xADL models.

xADL represents a promising step towards supporting the creation of domain-specific architectural languages. However, xADL schemas only define the syntax of modeling constructs, and do not provide any mechanism for specifying the semantics of domain concepts. Consequently, the ArchStudio toolset consists of parsers and other syntactic tools that are semantically unaware. For example, ArchLight provides an interface for extracting model data in XML format, and an interface for reporting analysis results within a graphical interface, but provides no support for any semantic transformation that must be implemented to convert xADL models into the input expected by off-the-shelf analysis engines. Instead, xADL requires architects to develop, from scratch and without guidance, "wrappers" (a.k.a., model interpreters) that perform the transformation. Therefore, xADL does not address the challenge described in Section 2.

8 Conclusions

This paper presented a methodology for the construction of analytic frameworks that enable the prediction of the quality attributes 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 they must fulfill a set of design requirements. This paper also demonstrated the process of constructing, utilizing, and validating a model interpreter framework using a case study.

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 quality attributes.

Acknowledgements

This material is based upon work sponsored by Bosch and the NSF 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.

References

[Boehm 1981] Boehm, B.: Software Engineering Economics, Prentice-Hall, 1981.

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

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

[Crnkovic 2005] Crnkovic, I., et al.: 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, 2005.

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

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

[Edwards 2007] Edwards, G., et al.: Scenario-Driven Dynamic Analysis of Distributed Architectures. Proc. of the Fundamental Approaches to Soft. Eng, 2007.

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

[Feeney 2001] Feeney, L.M., et. al.: Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. Proc. of IEEE INFOCOM, pp. 1548-1557, 2001.
[Folmer 2004] Folmer, E., et al.: Software Architecture Analysis of Usability. Proc. of the IFIP Working Conf. on Eng. for Human-Computer Interaction, pp. 321-339, 2004.
[Gamma 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[GME] The Generic Modeling Environment. http://www.isis.vanderbilt.edu/projects/gme/
[Gokhale 2005] Gokhale, A., et al.: Model-Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications, Elsevier Journal of the Science of Computer Programming: Special Issue on Model Driven Architecture, 2005.
[Hissam 2002] Hissam, S.A., Stafford, J.A., Wallnau K.C.: Packaging Predictable Assembly. Proc. of the ACM Working Conf. on Component Deployment, pp. 108-124, 2002.
[Hissam 2005] Hissam, S., et al.: Pin Component Technology (V1.0) and Its C Interface. Tech. Report CMU/SEI-2005-TN-001, Software Eng. Institute, 2005.
[Lassing 2002] Lassing, N., et al.: Experiences with ALMA: Architecture-Level Modifiability Analysis. Journal of systems and software, Elsevier, pp. 47-57, 2002.
[Ledeczi 2001] Ledeczi, A., et al.: On metamodel composition. Proceedings of the 2001 IEEE International Conference on Control Applications, pp. 756 - 760, 2001.
[Magee 1999] Magee, J., et al.: Behaviour Analysis of Software Architectures. Proceedings of the TC2 First Working IFIP Conference on Software Architecture, pp. 35 - 50, 1999.
[Malek 2005] Malek, S., Mikic-Rakic, M., et al.: A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. IEEE Trans. on Soft. Engineering, 2005.
[Malek 2007] Malek, S., Seo, C., et al. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. Proc. of the 29th International Conference on Software Engineering, 2007.
[OSGi] The Open Services Gateway Initiative. http://www.osgi.org/
[Roshandel 2006] Roshandel, R., Banerjee, S., Cheung, L., Medvidovic, M., and Golubchik, L.: Estimating Software Component Reliability by Leveraging Architectural Models. Proc. of the 28th International Conference on Software Engineering, 2006.
[Schmidt 2006] Schmidt, D.C.: Model-Driven Engineering. IEEE Computer, pp. 41 - 47, 2006.
[Schriber 2005] 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.
[Seo 2006] Seo, C., et al.: Energy Consumption Framework for Distributed Java-Based Software Systems. Tech. Report USC-CSE-2006-604, Univ. of Southern California, 2006.
[UML] The Unified Modeling Language. http://www.omg.org/spec/UML/1.4
[Wallnau 2003] Wallnau, K.: Volume III: A Technology for Predictable Assembly from Certifiable Components. Tech. Report CMU/SEI-2003-TR-009, Software Eng. Institute, 2003.
[Woodside 2002] Woodside, M.: Tutorial Introduction to Layered Modeling of Software Performance. Carleton University, http://sce.carleton.ca/rads.