MODEL INTERPRETER FRAMEWORKS

A NEW APPROACH TO LEVERAGING DOMAIN-SPECIFIC MODELS

George Edwards and Nenad Medvidovic, *Computer Science Department, University of Southern California*

One of the practical challenges in utilizing domain-specific modeling technologies is the construction of model interpreters that leverage domain-specific models for code generation and quality analysis. Model interpreter frameworks (MIFs) simplify and reduce model interpreter development effort by providing off-the-shelf analysis and synthesis capabilities, while still permitting domain-specific customization. The XTEAM model-driven design environment implements highly extensible MIFs that generate system simulations and source code from domain-specific models. Drawing on our experience building MIFs within XTEAM, as well as the related (though independent) experience of other research groups, we identify the shared, essential characteristics of MIFs.

INTRODUCTION

Organizations that develop software-intensive systems that demand rigorous architecture and design modeling, such as aerospace and defense systems, sensor network applications, and enterprise services, have discovered the benefits of domain-specific software modeling technologies. Software architects build domain-specific design models for the same reasons that they build design models in general: to automate system construction and improve system quality. In contrast to standardized modeling languages like UML, domain-specific modeling languages are customized for use within a specific context. This flexibility allows software engineers to capture design decisions in a more concise and intuitive way, using the concepts and abstractions of their choosing. Domain-specific languages allow engineers to focus on the key design decisions that impact system quality in their domain, and avoid specification of details that are less important in that domain.

Despite these advantages, industry adoption of domain-specific modeling technologies lags far behind that of UML. We believe the principal reason for this lack of adoption is that creating domain-specific languages is far easier than using them. Today's metamodeling technologies actually make the construction of domain-specific modeling environments relatively easy (see Figure 1). Building the code generators, analysis tools, and simulation engines necessary to leverage the resulting domain-specific models, however, is difficult and costly.

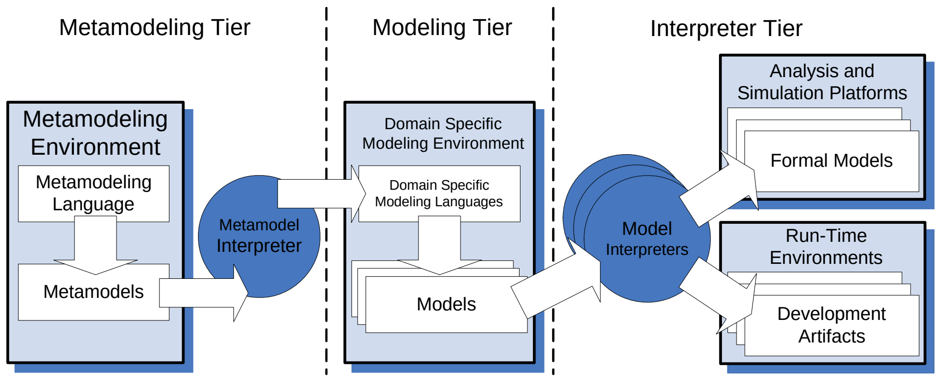


Figure 1: Software engineers can rapidly construct domain-specific modeling environments by creating a metamodel, which is a specification of a domain-specific language. State-of-the-art modeling tools use the metamodel to automatically synthesize a domain-specific modeling environment. Domain-specific code generators and analysis engines, on the other hand, have to be manually constructed.

The ability to apply code generators and analysis tools to assist and automate development activities represents a major reason for modeling software designs in the first place. Since domain-specific languages contain unique modeling constructs and semantics, software engineers must build their own specialized code generators and analysis tools. On the other hand, a large body of UML-based code generators and analysis tools is available off-the-shelf.

Once built, the maintenance of domain-specific code generators and analysis tools – collectively referred to as *model interpreters* – becomes an ongoing effort, as they must continually be updated to reflect the evolution of the corresponding domain-specific language. When a new domain-specific language is required, these tools have to be rebuilt.

Constructing and maintaining domain-specific model interpreters is no trivial task. Domain-specific modeling environments generally provide an API for custom-built model interpreters to access and extract the information captured in models, but their support for interpreter development stops there. That leaves the tough job of determining how to correctly and efficiently implement, say, a mapping from a design model to source code or an algorithm that analyzes system quality, to the user of a domain-specific language. To do so requires an understanding of both the relevant domain concepts and the particulars of model transformation, yet domain experts are rarely experts in model transformation, and vice versa. Design processes and guidelines for model interpreters are notably lacking.

What is needed is a way to provide pre-built, off-the-shelf code generators and analysis tools that can be applied to domain-specific models and permit domain-specific customization. We have accomplished this by incorporating *model interpreter frameworks* into domain-specific modeling tool-chains. Model interpreter frameworks, or MIFs, leverage the commonality among domain-specific languages to provide partial implementations of useful model transformations off-the-shelf, and in doing so, free engineers from much of the complexity of model interpreter development. At the same time, MIFs, like traditional application frameworks, employ object-oriented design patterns to create extensibility points that permit customization and incorporate domain-specific semantics.

We have implemented MIFs as the foundation of XTEAM [4], our domain-specific modeling, analysis, and synthesis tool-chain. Other research efforts in the area of domain-specific modeling, such as Cadena [6] and PACC [13], are also developing MIFs (although different names are used, such as "reasoning framework" or "analysis framework"). These related projects have developed MIFs with powerful capabilities and useful features, but have not fully recognized the broad applicability and significant ramifications of the MIF concept. That has led us to study the similarities and differences among MIFs, characterize the essential features of MIFs, and discover the relative advantages of MIF design and implementation strategies. We believe this new generation of MIF-enabled domain-specific modeling technologies has the potential to dramatically expand the types of development projects to which domain-specific modeling is practical and beneficial.

MIFs represent a compromise option for model-driven development between the two extremes available today – off-the-shelf tools that are difficult or impossible to modify, and custom-built tools that require substantial implementation effort. The use of MIFs involves a trade-off for software engineers. In return for sacrificing some flexibility in a domain-specific language, an MIF alleviates much of the cost and effort of utilizing domain-specific models for design analysis and code generation.

As we explain in the following sections, MIFs provide substantial benefits in many cases, but challenging research questions remain. We begin with a brief description of how model interpreters are normally built and used. We then introduce the central elements of our new approach to constructing model interpreters and summarize critical design and implementation choices in MIFs. To illustrate our ideas, we describe selected components of XTEAM, our domain-specific modeling and analysis tool-chain, and we detail the advantages we have observed using XTEAM on industrial systems. Where appropriate, we compare and contrast our methods and tools from both the state-of-the-practice and other emerging domain-specific tool-chains that incorporate MIFs. We conclude the article with a discussion of remaining research challenges and open problems.

MODEL TRANSFORMATION AND MODEL INTERPRETERS

To appreciate how MIFs improve domain-specific modeling tool-chains, it is necessary to understand the processes implemented within model interpreters. Model interpreters implement *model transformations*. Model transformation is one of the central activities in model-driven development paradigms such as model-driven architecture (MDA) and model-driven engineering (MDE). Model transformation allows a single model to be used for a variety of purposes. Model transformation also allows a "high-level" language (like UML) to be automatically translated into a "low-level" language (like C), just as compilers translate object-oriented programs into machine code. In practical terms, a model interpreter is usually implemented as a plug-in to a modeling environment that performs operations on models created in that environment.

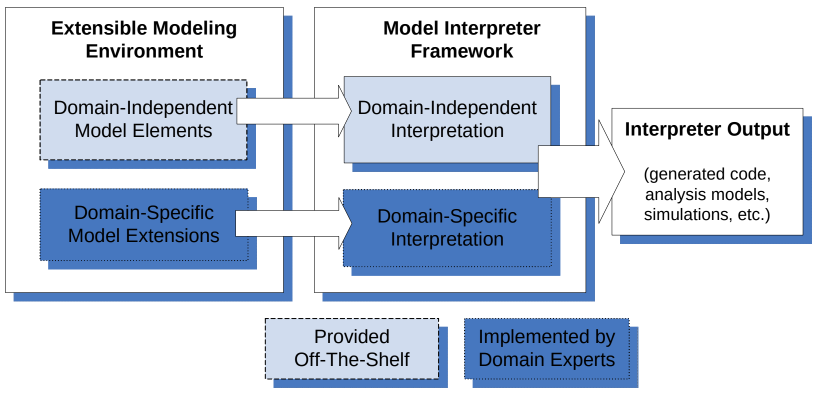
A model interpreter performs a *semantic* transformation on models. That is, a model interpreter performs a transformation that requires definition and understanding of the semantics, or meaning, of model elements. A model interpreter realizes the semantics of the modeling constructs on which it operates by defining the consequences of the use of those constructs within a given context. For example, translation between modeling languages whose constructs have possibly similar though not identical meanings is a semantic transformation. Translation of a graphical representation of a model into an equivalent textual form is not a semantic transformation.

Current metamodeling environments allow users to specify the syntax of domain-specific languages. However, they remain totally agnostic as to the semantics of those languages. The developers of these metamodeling tools assume that users will define language semantics in some external artifact, such as a natural language specification document. This metamodeling approach has important consequences: tool users are given complete freedom to define any semantics they wish, which is desirable in principle, but tool developers are unable to provide any built-in model interpreters, which is a high price to pay. In practice, most tool users do not need total freedom to define arbitrary semantics, and they would gladly trade some flexibility for built-in model interpreter support.

CORE MODELING APPROACH

The keys to enabling the application of pre-built, off-the-shelf interpreters to customized modeling languages are: (1) a standardized way to associate semantics with domain-specific language elements, (2) an effective way for pre-built interpreters to leverage those defined semantics to perform model transformations.

Our approach to achieving these two objectives is to separate the domain-independent features of a model from the domain-specific features. The domain-independent features have standardized semantics, while the domain-specific features have customized semantics. An off-the-shelf MIF can operate on the domain-independent model elements, while permitting extensions that accommodate domain-specific elements, the interpretation of which are not known *a priori*. Figure 2 shows a conceptual illustration of our approach.



Extensible Modeling Environment
Domain-Independent Model Elements
Domain-Specific Model Extensions
Model Interpreter Framework
Domain-Independent Interpretation
Domain-Specific Interpretation
Interpreter Output
(generated code, analysis models, simulations, etc.)
Provided Off-The-Shelf
Implemented by Domain Experts

Figure 2: By separating the domain-independent elements of a model from domain-specific elements, partial model transformations, implemented by a model interpreter framework, can be provided off-the-shelf.

ABSTRACT COMPONENT TECHNOLOGY

Several emerging domain-specific software modeling technologies achieve the separation of domain-independent and domain-specific model features through an *abstract component technology*, or ACT. An ACT is a highly extensible modeling language that is based on the central design elements employed in software architectures: software components and their interfaces, configurations, and interactions.

While the properties and capabilities of software components vary in some respects across application domains and middleware platforms, in other respects their semantics remain constant. For example, software components are independent system elements with internal state and precise interfaces. An ACT defines a “software component” modeling element (and other building blocks for design models) in these domain-independent terms, and defers to software engineers the specification of any domain-specific properties and features of components.

ACTs are specialized for modeling component-based software architectures, and they are not intended to be used for modeling other types of systems (such as biological systems or mechanical systems). By restricting the types of systems to which an ACT is applicable, an ACT designer can associate meaningful semantics with metamodel types. This compromise between semantic definition and flexibility stands in contrast to current metamodeling languages and environments, which, as noted above, provide near total flexibility to define arbitrary modeling languages but suffer from a lack of any semantic definition.

MODEL INTERPRETER FRAMEWORKS

Domain-specific languages that are defined using an ACT are “MIF-ready.” These languages have been customized, but they can still be used with MIFs created for that ACT. Just like a standard model interpreter, the output of an MIF may be programming language code, configuration files, formal models, system simulations, architecture documentation, or other artifacts.

An MIF performs a model transformation by applying both a pre-defined transformation of domain-independent ACT elements and a user-defined transformation of domain-specific ACT extensions. The logic that interprets the domain-independent elements constitutes the core of the MIF. The logic that interprets domain-specific elements is implemented in MIF extensions. Depending on the design of the MIF, extensions may take a variety of forms. Extension points are often created using object-oriented design patterns such as Strategy, Template Method, and Functor.

DESIGN AND IMPLEMENTATION

There are multiple strategies for designing and implementing ACTs and MIFs, which affect their usability, extensibility, and other properties. Here we outline some of the key issues in ACT and MIF construction, and describe the design of XTEAM, our domain-specific modeling, analysis, and synthesis tool-chain.

XTEAM is built on top of the Generic Modeling Environment (GME) [3], an off-the-shelf domain-specific modeling tool. We defined the XTEAM ACT using the metamodeling language of GME, and built the XTEAM MIFs as extensible plug-ins to GME. Figure 3 depicts the XTEAM tool-chain.

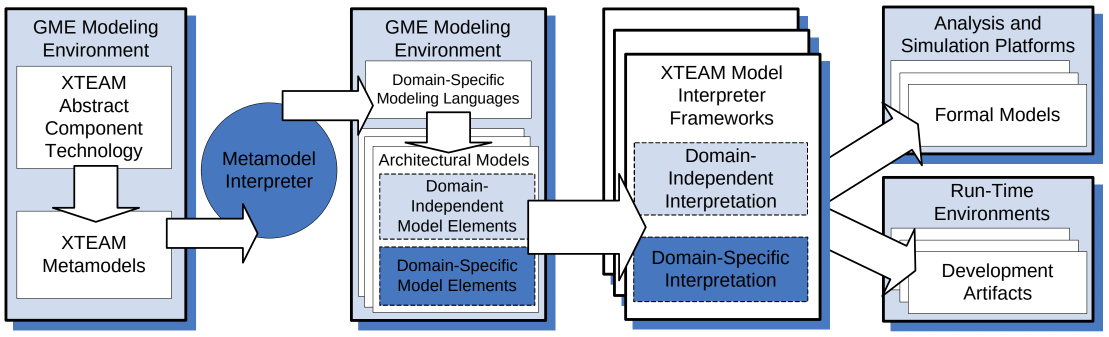

GME Modeling Environment
XTEAM Abstract Component Technology
XTEAM Metamodels
Metamodel Interpreter
GME Modeling Environment
Domain-Specific Modeling Languages
Architectural Models
Domain-Independent Model Elements
Domain-Specific Model Elements
XTEAM Model Interpreter Frameworks
Domain-Independent Interpretation
Domain-Specific Interpretation
Analysis and Simulation Platforms
Formal Models
Run-Time Environments
Development Artifacts

Figure 3: Conceptual depiction of the XTEAM tool-chain. The XTEAM abstract component technology is implemented as a metamodeling language. Software engineers define domain-specific architectural modeling languages by creating XTEAM metamodels. XTEAM model interpreter frameworks can then be applied to domain-specific architectural models to analyze quality properties and synthesize source code.

ACT DESIGN

The designer of an ACT decides what modeling types to include in an ACT and what semantics to associate with those types. A more fully-defined ACT reduces the flexibility of ACT users to customize domain-specific elements, but enables the construction of more powerful and complete MIFs. Conversely, a less fully-defined ACT permits greater domain-specific customization, but reduces the built-in capabilities of associated MIFs. There are two approaches to implementing and using an ACT:

ACT Metamodel. An ACT may be defined as a metamodel. The ACT metamodel contains only domain-independent elements. Software engineers define domain-specific languages by adding new elements, relationships, properties, constraints, etc., to the ACT metamodel. Figure 4(a) shows a fragment of an ACT metamodel with elements named *Architecture*, *Group*, and *Component* (tagged with the <<ACTElement>> stereotype). The ACT metamodel has been extended with domain-specific types (tagged with the <<ACTExtension>> stereotype). The domain-specific types have additional properties associated with them (*e.g.*, the *architecturalStyle* property of *PrismArchitecture*).

ACT Metalanguage. Alternatively, an ACT may be defined as a metamodeling language (or *metalanguage*). The ACT metalanguage defines domain-independent types. Software engineers define domain-specific languages by instantiating ACT metalanguage types as domain-specific metamodel elements. Figure 4(b) shows a fragment of a metamodel defined using an ACT metalanguage. The domain-specific elements are instances of ACT metalanguage types, which are depicted as stereotypes in the figure.

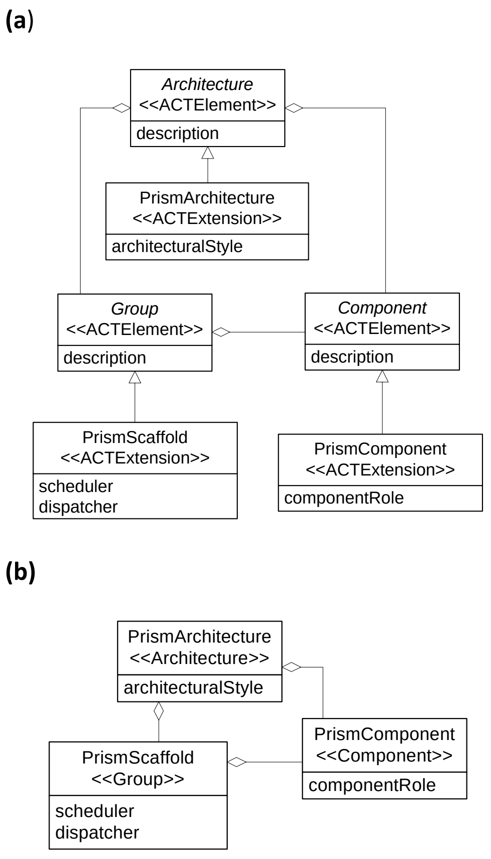


Figure 4: Examples of domain-specific metamodel fragments defined by (a) extension of an ACT metamodel and (b) use of an ACT metalanguage.

The two ACT implementation approaches each have advantages. The primary advantage of a metamodel-based ACT is that the relationships between ACT elements are accessible and explicit from the point of view of ACT users because they are apparent within the user metamodeling environment. Also, the overall tool-chain is conceptually simpler because it involves only two tiers (a metamodel tier and a model tier) rather than adding a third tier (the ACT metalanguage). The advantage of a metalanguage-based ACT is that the ACT metamodel is decoupled from user metamodels. As a result, user metamodels are smaller and simpler, and evolution of the ACT does not impact users as severely and directly.

THE XTEAM ACT

The first version of XTEAM was implemented a metamodel-based ACT [5]. This ACT was based on the composition of two software architecture descriptions languages: xADL Core [2], which defines architectural structures and types, and FSP [7], which defines architectural behaviors. The newest version of XTEAM relies on a metalanguage-based ACT [4], but incorporates most of the elements of the original ACT. The decision to migrate to an ACT metalanguage was made to simplify the XTEAM user interface by reducing the complexity of domain-specific metamodels built by users and abstracting the details of ACT implementation.

The XTEAM ACT defines types that correspond to architectural design constructs:

- An *Architecture* represents a collection of components and connectors that have been instantiated in a specific topology or configuration.
- A *Component* represents a locus of computation. Components interact with other components and connectors through interfaces.
- A *Connector* implements communication and coordination facilities. Like components, connectors interact with external entities via interfaces.
- An *Interface* represents an interaction point between a component or connector and external entities. Interfaces define entry points to component services in terms of data exchange and transfer of control.
- A *Link* represents a logical connection between components or connectors over which information and/or control is exchanged via interfaces.
- A *Datum* represents an object exchanged between components and connectors.
- A *Process* represents a thread of execution or sequence of operations within a component or connector.

The types included in the XTEAM ACT are intended to represent the fundamental units used to define software systems. This canonical set of constructs captures the consensus of the software architecture community and commonly accepted architectural practices and principles. As such, we do not make a formal argument of the completeness of this set of types, but rely on the evidence of their usefulness and expressiveness accumulated from the past two decades of software architecture research and practice.

OTHER ACTS

Other model-driven design, analysis, and synthesis environments also employ ACTs, although their goals and areas of emphasis vary.

The Cadena Architecture Language with Metamodeling. The Cadena Architecture Language with Metamodeling (CALM) [6] is an ACT implemented by the Cadena modeling and development framework. 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. 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. The CALM style tier constitutes a metalanguage-based ACT. CALM is based on component, connector, and interface metatypes, with comparable semantics to the equivalent XTEAM elements.

Construction and Composition Language. The Construction and Composition Language (CCL) [13] is a design language for specifying components and their behaviors and assemblies. CCL also includes constructs for the specification of connectors as services provided by a component execution environment. CCL is leveraged within model-driven development tools produced by the Predictable Assembly from Certifiable Components (PACC) Initiative, such as the PACC Starter Kit, to specify domain-specific component models and analysis properties. CCL has a strong emphasis on precise behavioral semantics for rigorous analysis, and is less concerned with domain-specific extensibility.

MIF DESIGN

An MIF consists of a framework core and framework extension points. The framework core should hide as much of the complexity of the transformation process as possible; this makes it easier to use the framework.

Figure 5 illustrates a notional design of an MIF. The MIF navigates through an architectural model, interpreting each element according to the domain-independent semantics of the associated ACT type. During the interpretation of each element, a framework extension is invoked that leverages domain-specific information in the model to change or augment the interpreter output.

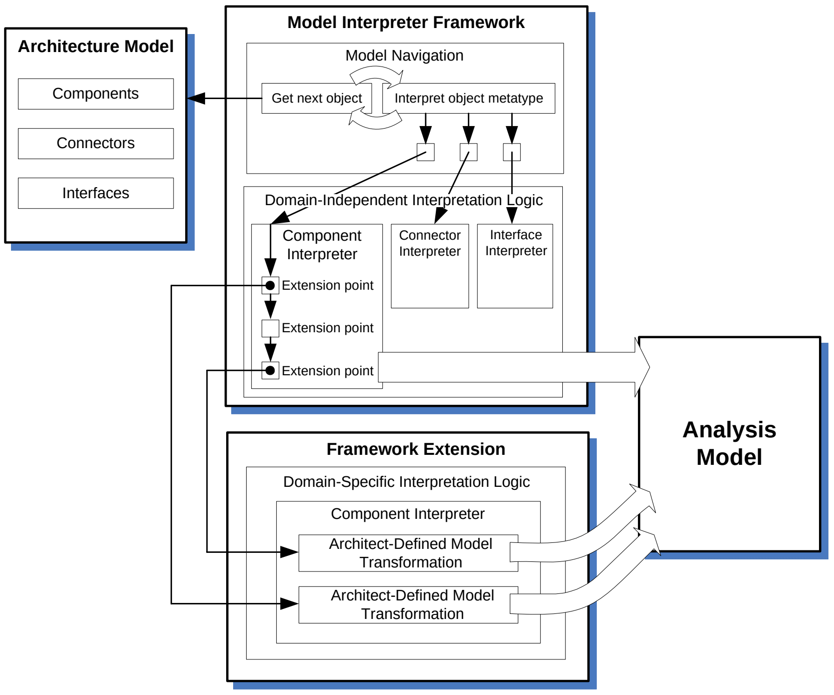


Figure 5: Notional depiction of the design of a model interpreter framework.

Implementing an MIF is more difficult than implementing a standard interpreter because an MIF must be structured in a way that both hides operations that users do not want to be exposed to and exposes operations that users want to customize in different situations. MIFs that produce widely applicable artifacts (*i.e.*, artifacts that are used for multiple purposes in multiple domains) provide the highest return on investment. MIFs that produce domain-specific artifacts (*e.g.*, code that runs on a domain-specific middleware platform) provide limited opportunities for reuse, and consequently, limited cost, time, and effort savings.

THE XTEAM MIFS

XTEAM provides two MIFs. A simulation MIF transforms architectural models into scenario-based simulations that run on the *adevs* discrete event simulation engine [1]. A code generation MIF transforms architectural models into source code that runs on the Prism-MW middleware platform [10].

Discrete Event Simulation. The XTEAM MIF for discrete event simulation employs the Strategy pattern to enable software engineers to implement domain-specific extensions. 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.

Each extension to the discrete event simulation MIF generates code that realizes a particular type of analysis. To demonstrate this capability, we have implemented several dynamic architectural analyses using the discrete event simulation MIF. We highlight the following three analyses here:

- *Performance.* The MIF extension for performance analysis creates a Layered Queuing Network (LQN) simulation. LQNs are used to calculate performance metrics such as latency, throughput, and utilization. Our LQN simulation measures and records the end-to-end latency of request-response interactions. The observed latencies depend on numerous factors, including the load applied to the system, the computational resources available, the size of data sets, and other stochastic factors.
- *Reliability*. The MIF extension for reliability analysis leverages a Hidden Markov Model (HMM) simulation to calculate component reliability. Component reliability is defined as the percentage of time the component spends in a normal operational mode. Potential faults specified in the architectural model occur with the probabilities defined by the architect. In response, the system may take recovery actions to mitigate the fault, such as instantiating back-up components or changing the system deployment. While the system performs these recovery actions, it is considered to be in a failure mode, and the reliability drops accordingly.
- *Energy consumption*. The energy consumption MIF extension uses an energy consumption estimation technique [11] that defines equations to calculate the energy used by the executing software based on a number of application-specific and platform-specific parameters. The total energy cost is the sum of the computational energy cost, due to CPU and memory usage, and the communication energy cost, due to sending and receiving data over a wireless network.

Prism-MW Code Generation. The XTEAM MIF for Prism-MW synthesizes C++ source code from architectural models. Domain-specific extensions are incorporated into the Prism-MW code generation MIF using the Template Method design pattern. The Template Method pattern allows the structure of an algorithm to be defined, while the implementation of certain steps within the algorithm is deferred to subclasses.

Extension points within the Prism-MW code generation MIF permit the customization of the synthesis process. The Prism-MW middleware platform is itself highly extensible, so the extension capabilities within the Prism-MW code generation MIF complement and enhance the corresponding features in the middleware.

- *Resource allocation*. The MIF extension for resource allocation permits fine-grained control over how runtime objects are allocated and destroyed. For example, objects may be allocated on-demand or they may be pre-allocated in a resource pool. This MIF extension allows engineers to specify the resource allocation policies for different components and services, so that system implementations generated from the model exhibit the desired behaviors.
- *Network management.* The network management MIF extension generates code required to setup and configure network connections. This allows applications to transparently substitute the default IP network protocol used by Prism-MW with domain-specific network and transport protocols. One such protocol we have relied on in constructing a family of embedded systems [9] is the Controller Area Network (CAN) protocol commonly used in the automotive domain.

OTHER MIFS

MIFs are being incorporated into other advanced analysis and synthesis tool-chains.

Bogor. Bogor is a highly extensible model checking framework that can be applied to Cadena models. Like other MIFs, Bogor allows engineers to reuse the majority of the analysis infrastructure while customizing selected constructs and behaviors. Bogor allows engineers to extend the input language to the model checking framework with domain-specific abstractions. The semantics of each extension is defined in a user-provided Java package. Creating extensions of this type makes the generation of finite state models from high-level, domain-specific design models much more straightforward. The extension mechanism also allows engineers to expose only those aspects of domain-specific abstractions that are relevant to a particular analysis, which can greatly improve the efficiency of the model checking infrastructure.

PSK Reasoning Frameworks. The PACC Starter Kit (PSK) is a model-driven development environment that utilizes MIFs, which the developers term "reasoning frameworks." PSK includes a performance framework and a model-checking framework. The performance framework transforms component-based architectural models into a form that supports rate monotonic analysis for prediction of worst-case response times. The model-checking framework transforms architectural models into the input language of a software model checker that verifies safety and security properties.

The appearance of MIFs in multiple model-driven environments highlights their shared characteristics and provides further evidence of their potential and usefulness. However, to our knowledge, the developers of these environments have not taken a broad a view of MIFs, and have not studied the features, constraints, and applicability of MIFs in general.

EXPERIENCE

We, and others, have applied XTEAM within the context of multiple software development efforts. We used XTEAM to model, analyze, and generate code for the MIDAS product line, a family of sensor network applications developed in collaboration with Bosch Research and Technology Center [9]. More recently, we applied XTEAM to a mobile robotics application to evaluate design alternatives in terms of quality metrics [8]. XTEAM has been used as an instructional tool in a graduate-level software engineering course [12]. Finally, researchers at George Mason University have been working with the US Army to identify performance bottlenecks in the Army's data-intensive software systems through construction of detailed architectural models in XTEAM. Here we summarize our experience with the MIDAS application family.

MIDAS is a product line architecture for sensor network applications comprising approximately 100 KSLOC. MIDAS applications execute on networks of sensors, gateways, hubs, and handheld devices. XTEAM was used to create a domain-specific development infrastructure for MIDAS applications, which provided several benefits.

Application components in MIDAS have specific properties, capabilities, and constraints associated with them, depending on their role within the MIDAS reference architecture. For example, gateway components manage groups of sensors, aggregate and fuse sensor data, and forward the fused data to hubs. Using the XTEAM ACT, we were able to capture the MIDAS reference architecture in a way that allowed developers to leverage the domain-specific patterns and abstractions defined by that reference architecture. Component types were created in the MIDAS metamodel corresponding to each role in the reference architecture (such as the "gateway component" role). Each component type was then customized with special attributes and prescribed interactions.

MIDAS applications are subject to a number of quality requirements. For instance, in a MIDAS application that provides building monitoring services, such as intrusion and fire detection, some hardware devices are not connected to a continuous power supply, but instead run on battery power. As a result, the system's efficiency with respect to energy consumption has a critical impact on the longevity of the system services. Applying the XTEAM discrete event simulation MIF, we were able to determine the most efficient way to implement interaction between distributed software components from a set of candidate designs. A subsequent comparison of XTEAM's predictions to measured values taken from the executing system showed that XTEAM's estimates were within 7% of the actual energy used.

We synthesized source code for MIDAS components directly from XTEAM models. These components were part of a smart home system, in which building monitoring and control of appliances are integrated in a single application. The XTEAM Prism-MW generator produced program code that runs on embedded devices for eight application components comprising several thousand lines of source code.

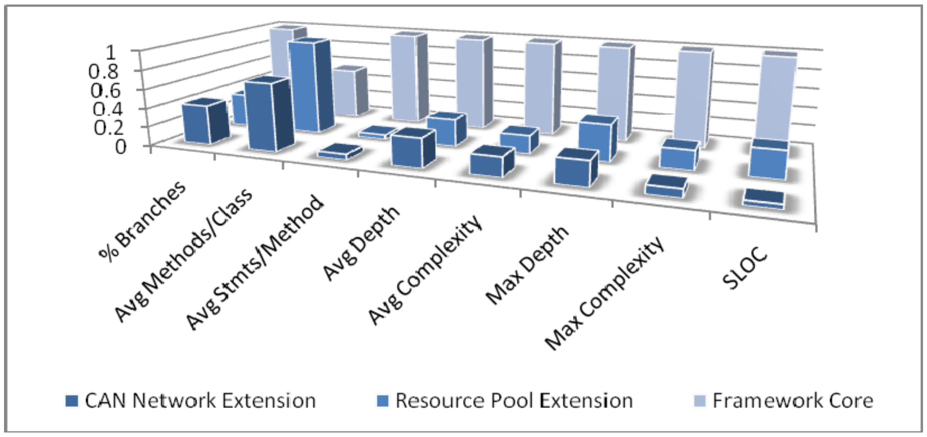
BENEFITS

Developing model interpreters on top of an MIF involves implementing MIF extensions, but the model transformation logic within the MIF core does not need to be reprogrammed. In contrast, standard model interpreter development requires implementation of both MIF extensions and an MIF core. Comparing the complexity of MIF extensions to that of the corresponding MIF core quantifies the advantage of using an MIF.

Figure 6 graphs the relative complexities of each of the XTEAM model interpreter frameworks and their extensions along eight dimensions. The first seven metrics show average and maximum values for commonly-used complexity indicators. These metrics are not affected by the size of the code base, but are instead calculated on a per-method or per-class basis. The last metric, SLOC, shows the implementation size of each module. The units of each metric are different, so the values have been normalized to permit them to be displayed on a single graph.

The data indicates that the size of the source code for a MIF core is an order of magnitude larger than the size of framework extensions, and the code in an MIF core is more complex than the code in an MIF extension. In other words, the MIF represents most of the overall interpreter code and the hardest part of the interpreter to implement. Consequently, the reuse afforded by MIFs allows software architects to bypass a substantial portion of the typical model interpreter implementation effort.

(a)



(b)

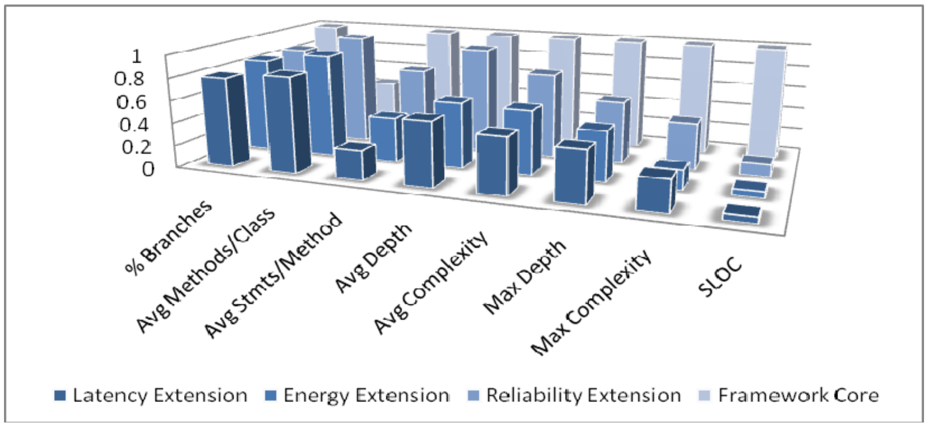


Figure 6: Comparison of source code size and complexity between the MIF core and MIF extensions for the (a) Prism-MW code generation MIF and (b) Adevs discrete event simulation MIF.

CONCLUSIONS

A number of interesting research problems related to ACTs and MIFs remain open to further investigation. The classes of model transformations (and corresponding analysis and code generation facilities) that are amenable to implementation in an MIF are not fully known. Model checking frameworks and simulation frameworks have been implemented, but it is not clear whether other types of analysis that do not fit within these frameworks can, and should, be implemented within MIFs. Also, researchers should study how the semantic definition of ACT elements impacts the flexibility and usability of corresponding MIFs. An over-defined ACT will likely reduce the flexibility of MIFs, while an under-defined ACT will allow for limited reuse of transformation logic within an MIF; however, the most practical balance between these conflicting goals has not been demonstrated empirically or formally as of yet. Finally, the potential of MIFs for code generation remains largely unexplored. XTEAM's code generation MIF draws its flexibility from the highly extensible nature of the underlying platform, Prism-MW. It remains to be seen whether code generation for other, less extensible platforms can be implemented as effectively in an MIF.

REFERENCES

1 Adevs: A Discrete EVent System simulator. http://www.ornl.gov/~1qn/adevs/index.html.

2 Dashofy, E., van der Hoek, A., and Taylor, R. N. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In Proc. of the 24th International Conference on Software Engineering (ICSE), May 2002.

3 GME: The Generic Modeling Environment. http://www.isis.vanderbilt.edu/Projects/gme/.

4 Edwards, G. and Medvidovic, N. A Methodology and Framework for Creating Domain-Specific Development Infrastructures. In Proc. of the 23rd IEEE ACM International Conference on Automated Software Engineering (ASE), September 2008.

5 Edwards, G., Malek, S., and Medvidovic, N. Scenario-Driven Dynamic Analysis of Distributed Architectures. In Proc. of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE), March 2007.

6 Jung, G. and Hatcliff, J. A Type-centric Framework for Specifying Heterogeneous, Large-scale, Component-oriented, Architectures. In Proc. of the 6th International Conference on Generative Programming and Component Engineering (GPCE), October 2007.

7 Magee, J., Kramer, J., and Giannakopoulou, D. Behaviour Analysis of Software Architectures. In Proc. of the TC2 First Working IFIP Conference on Software Architecture (WICSA), February 1999.

8 Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., and Sukhatme, G. An Architecture-Driven Software Mobility Framework. Submitted to the Journal of Software and Systems: Special Issue on Software Architecture and Mobility, 2009.

9 Malek, S., Seo, C., Ravula, S., Petrus, B., and Medvidovic, N. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. In Proc. of the 29th International Conference on Software Engineering (ICSE), May 2007.

10 Medvidovic, N., Mikic-Rakic, M., Mehta, N., and Malek, S. Software Architectural Support for Handheld Computing. IEEE Computer – Special Issue on Handheld Computing, September 2003.

11 Seo, C., Malek, S., and Medvidovic, N. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In Proc. of the International Symposium on Component-Based Software Engineering (CBSE), October 2008.

12 Software Engineering for Embedded Systems – CSCI 589. University of Southern California, Fall 2006. http://csse.usc.edu/classes/cs589_2006/.

13 Wallnau, Kurt C. Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, 2003.