A Methodology and Framework for Creating Domain-Specific Development Infrastructures

George Edwards
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
gedwards@usc.edu

Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
neno@usc.edu

Abstract

Domain-specific architectures, middleware platforms, and analysis techniques leverage domain knowledge to help engineers build systems more effectively. An integrated set of these elements is called a domain-specific development infrastructure (DSDI). DSDIs are commonly created in a costly, ad-hoc fashion because current model-driven engineering (MDE) technologies lack sufficient mechanisms for capturing the semantics of domain concepts. In this paper, we propose a methodology for incorporating semantics within MDE frameworks to simplify and automate DSDI integration. We also present and evaluate a framework, called XTEAM, that implements our approach, resulting in structured processes and enforceable guidelines for DSDI integration. We have applied our approach to several DSDIs, and report on the benefits accrued.

1. Introduction

The goal of domain-specific software engineering (DSSE) methods is to leverage the characteristics of an application domain in order to create high-level abstractions that can be employed in the design and implementation of software systems. DSSE captures domain knowledge in several ways to enable effective reuse of design solutions and implementation artifacts:

- Domain-specific reference architectures (e.g., Bold Stroke [29], MIDAS [21]) define generalized software designs that can be customized, parameterized, and specialized for use within a certain context.
- Domain-specific middleware platforms (e.g., RT CORBA [4], Prism-MW [20]) provide implementation constructs and run-time services tailored for the needs of a domain, supplying applications with reusable abstractions of recurring patterns and algorithms.
- Domain-specific analysis technologies (e.g., fault-tree analysis in safety-critical systems [31]) apply computational theories to system models in order to derive information about properties or behaviors of a system that are of particular importance for a given domain.

In this paper, we refer to the combination of the above three elements as a domain-specific development infrastructure (DSDI). The integration of DSDI elements offers a number of attractive benefits. Integrating a reference architecture with a middleware platform allows architects to describe their systems in terms of the design elements defined by the reference architecture and automatically generate the implementation constructs supported by the middleware [12]. Similarly, integrating a reference architecture with analysis technologies allows architects to use a single design model to investigate multiple functional and non-functional properties of a system, rather than building and synchronizing multiple different models [17]. Finally, integrating a middleware platform with an analysis technology allows architects to ensure that the assumptions relied upon by predictive analytic theories applied to design models are satisfied by the implemented system at run-time [3].

This paper defines and evaluates a methodology for constructing metaprogrammable, extensible frameworks that simplify, streamline, and automate DSDI specification and integration. Our methodology embraces the model-driven engineering (MDE) paradigm [27], which provides a basis for the requisite mechanisms. MDE technologies enable the construction of domain-specific modeling languages (DSMLs) through the use of metamodels. DSMLs can capture the structures, views, and constraints present in a domain-specific reference architecture, the facilities and services provided by a domain-specific middleware, and so on. Domain-specific analysis and code generation is achieved through the use of model interpreters, custom-built components that extract and manipulate information present in system models.

The practical use of MDE to construct DSDIs is hampered by a lack of automation, however. First, MDE tools are unable to automate important DSDI integration activities like metamodel composition and validation because these tasks require an understanding of metamodel semantics. Current metamodeling languages specify only the syntax of language elements, forcing architects to rely on manual DSDI integration processes. Moreover, because MDE tools cannot assume any semantics for the DSMLs they will be used to construct, they are prevented from providing automated analysis
and synthesis capabilities off-the-shelf. Instead, MDE tools merely provide APIs for architects to build model interpreters themselves, or tools that perform simple syntactic translations (e.g., from a graphical diagram to XML).

In order to deal with these shortcomings, we enhance the standard MDE paradigm with semantics at both the language and interpreter layers. First, our methodology improves upon the standard MDE metamodeling technique by employing an abstract component technology (ACT), a metamodeling language specialized for software architectures that associates semantics with metamodel constructs. Additionally, our approach leverages the ACT to construct model interpreter frameworks (MIFs), reusable design structures for implementing architectural analyses and system synthesis capabilities. This methodology is accompanied by a metalevel framework, called the eXtensible Toolchain for Evaluation of Architectural Models (XTEAM).

Our methodology represents the first integrated solution for complete DSDI realization. In this paper, we show how the methodology and the XTEAM framework were used to create several DSDIs, including (recursively) XTEAM itself. Our experience suggests that this approach exhibits a number of benefits, including support for principled development of DSDIs, elimination of redundant effort, and simplification of DSDI maintenance and evolution.

The rest of this paper is organized as follows. Section 2 introduces a case study that illustrates the concepts described in later sections. Section 3 enumerates DSDI integration challenges. We explain our methodology in Section 4 and describe its implementation in the XTEAM framework in Section 5. Section 6 presents an evaluation of this work. Discussions of related and future research conclude the paper.

2. Case Study

DSDIs are gaining increasing adoption in enterprises that develop families of large-scale distributed systems with unique requirements. However, DSDIs are usually constructed in a piecemeal, ad-hoc fashion and assembled manually. For example, a DSDI might consist of (1) a published reference architecture, (2) a middleware platform consisting of standards-based COTS products opportunistically enhanced with domain-specific infrastructure components, and (3) algorithms that implement customer-mandated architectural analyses. Some or all of the elements of a DSDI may have been created in isolation from the others, with little or no planning as to how their integration can occur. In this section, we introduce a DSDI for sensor network software we built in collaboration with an external organization. This DSDI consists of the MIDAS reference architecture [21], the Prism-MW middleware [20], and a real-time performance analysis [32]. The design and implementation of this DSDI are used to illustrate the methodology described in this paper. Only a brief summary is given here, while specific details are provided in subsequent sections.

Recall that the MDE paradigm provides two mechanisms for constructing DSDIs: the definition of metamodels and the construction of model interpreters. The roles, patterns, and configurations of a reference architecture can be captured by a metamodel. This allows architects to model a system using high-level design constructs and ensures that the system conforms to the reference architecture. Figure 1a shows a partial metamodel of the MIDAS reference architecture [21], using a generic entity-relationship notation (syntactic and presentation details have been removed for readability). MIDAS is a layered architecture in which well-defined interfaces (not shown) are implemented at each layer and invoked by the layer above.

The type system of a middleware platform (called a

Figure 1. Metamodels for (a) MIDAS reference architecture, (b) Prism-MW component model, and (c) LQN analysis model.
component model) can also be captured by a metamodel. This lets architects specify how middleware services are utilized. Figure 1b captures a portion of the metamodel of Prism-MW [20], an architectural style-aware middleware. Formal specifications of style constraints imposed by Prism-MW are part of the complete metamodel.

Finally, the formal language used by an analysis tool can be captured by a metamodel. This ensures that architectural models described by the metamodel include the system parameters necessary for analysis and obey the constraints imposed by the analysis technique. Figure 1c depicts the metamodel of the Layered Queueing Network (LQN) [32] performance analysis model. LQN tasks represent computational entities that implement a queue for incoming service requests, which are handled by a thread pool.

Using these metamodels, code that targets a middleware platform can be automatically generated from architectural models via a model interpreter. Interpreters can also synthesize middleware configuration files, deployment descriptors, and other implementation artifacts. In the MIDAS DSDI, model interpreters are used to generate system implementations that consist of component logic and “glue code” in C++, as well as middleware configuration files that specify quality-of-service (QoS) settings such as the sizes of resource pools.

Likewise, models that conform to the languages used by analysis technologies can be automatically generated from architectural models. This allows a single architectural model to be analyzed with respect to numerous quality attributes. The MIDAS DSDI implements model interpreters that perform end-to-end latency, reliability, and energy consumption analyses.

3. Challenges

MDE provides the basis for the integration of domain-specific reference architectures, middleware, and analysis technologies (i.e., the creation of DSDIs) through (1) the composition of their respective metamodels and (2) the construction of interpreters that implement model transformations. However, existing MDE technologies cannot provide satisfactory automation of these activities because of their lack of integrated semantics. This section describes the challenges created by this reliance on manual DSDI assembly.

3.1. Metamodel Integration Challenges

The metamodel composition process is one of determining and verifying the nature of the relationship between each pair of metamodel constructs. Relationships are defined using composition operators. We have adopted the operators proposed in [16]:

- **Equivalence** merges two types, including their properties (i.e., attributes, operations, and constraints) and relationships (associations), into a single type.
- **Full inheritance** creates a derived type that inherits both the properties and relationships of the base type.
- **Implementation inheritance** creates a derived type that inherits just the properties of the base type.
- **Interface inheritance** creates a derived type that inherits just the relationships of the base type.

The relationships that exist between the constructs of different metamodels are based on their semantics (i.e., what a construct represents in the real-world system being modeled). However, current metamodeling languages avoid definition of semantics because they are designed to be highly flexible and widely applicable. For example, in the Generic Modeling Environment (GME) [11], the core metamodeling types are the *model*, an object that has parts; the *atom*, an object with no parts; the *set*, a collection of objects; the *reference*, a pointer to an object; and the *connection*, a link between objects. Defining a domain concept as a GME model specifies that the concept may “contain” other objects, but it says nothing about what that containment means. The lack of semantics within metamodeling languages leads directly to the following practical problems:

**Imprecise determination of semantic relationships.** The elements that must be integrated within a DSDI – a reference architecture, middleware, and analysis technologies – comprise a diverse set of concepts. The semantic relationships between these disparate elements are not at all obvious in most cases. To compose metamodels such as those excerpted in Figure 1, an architect may be forced to improvise and rely on some external source (e.g., the middleware specification) in order to determine those concepts’ semantics.

**Onerous manual composition of metamodels.** Current MDE tools provide only manual metamodel composition mechanisms because they cannot leverage semantics of metamodel elements for automatic composition. The current state-of-the-art is to exhaustively specify the pairwise relationships between the constructs of different metamodels such as those shown in Figure 1.

**Lack of rigorous and automated validation mechanisms.** MDE tools cannot validate the correctness of metamodel integrations without knowledge of semantics. As a result, metamodel composition is error-prone.

In Section 4.1, we describe how endowing metamodels with semantics results in an elegant solution to the above problems.

3.2. Model Transformation Challenges

A DSDI depends on model interpreters to perform transformations on design models, thereby enabling system analysis and synthesis. The built-in transformation capabilities supported by current MDE tools are purely
syntax in nature. As such, they can transform a model that conforms to one language into a model specified in another language, as long as the semantics of the constructs in the two languages are the same (e.g., converting graphical models into XML or table form). The lack of semantic definition within MDE technologies causes the following problems:

High design and implementation complexity. Developing custom-built model interpreters for analysis and synthesis requires developing complex semantic mappings between languages. Model transformation is recognized as a sufficiently difficult problem that implementations of specific transformations, such as tools that map design models to performance or reliability models, are commonly published in research literature [2,5]. Furthermore, model interpreters have been found to be the primary source of errors in MDE toolchains that apply analytic theories to architectural models [15].

Disproportionate maintenance and evolution costs. Due to their tight coupling, whenever a modification is made to a DSML, a corresponding non-trivial change must be made to every interpreter. The ability to easily evolve metamodels and languages is, in theory, one of the most attractive benefits of MDE, so the difficulty in evolving interpreters is a significant obstacle.

Redundant development effort. Model interpreters are often developed in a “one-off” fashion due to a lack of effective mechanisms for facilitating reuse of interpreter logic. If a new development project begins that uses a different reference architecture but requires the same types of architectural analysis, every interpreter must be rebuilt. Opportunities for factoring out reusable transformation operations are thus discarded.

We describe how model interpreter frameworks leverage built-in semantics to resolve these challenges in Section 4.2.

4. Solution Approach

This section introduces our approach to addressing the two categories of challenges discussed above.

4.1. Metamodel Integration

To address the metamodel integration challenges, we propose the construction of MDE environments that employ an abstract component technology (ACT) as their metamodeling language. We define an ACT to be a metamodeling language, or metalanguage, that specifically targets modeling languages for software architectures. As such, an ACT can be thought of as a domain-specific metamodel. This allows an ACT designer to associate meaningful semantics with metamodel types, or metatypes. An ACT defines metatypes that correspond to the fundamental concepts in software architecture as well as the implementation units of middleware platforms, such as component, connector, interface, and link. The exact set of metatypes included in an ACT, as well as the implied syntax and semantics of the DSML constructs they represent, are decided by the ACT designer, but should be guided by the commonly accepted architectural practices and principles. ACT metatypes are defined only in terms of the properties and constraints that are common across application domains and platforms. For instance, a component might be defined as an independent unit of computation with internal state and a well-defined interface; however, the precise types of interfaces that components may provide would not be defined by the ACT because this varies greatly among platforms and domains.

Figure 2 portrays the Prism-MW component model (first introduced in Figure 1b) as it is captured using an ACT. The entities and relationships in the original diagram have not changed, but they are now strongly typed with ACT metatypes (shown as stereotypes). By typing the metamodel elements, we immediately associate semantics with each domain-specific construct by indicating its membership in a specific class of entities (e.g., a PrismPort represents a type of software interface). Since the (domain-independent) semantics of interfaces in general are known a priori (e.g., interfaces define component interactions), automated tools may assume that these semantics also apply to PrismPorts.

An architect obtains the benefits of an ACT in the following way. First, the architect creates metamodels that use ACT metatypes to capture domain-specific reference architectures, middleware type systems, and analysis formalisms, as in Figure 2. These metamodels inherit the built-in semantics of the ACT, allowing them to be automatically composed into an integrated metamodel (through the process detailed below) and to serve as a foundation for quality attribute analysis and system synthesis (as explained in Section 4.2). Using the standard MDE process [27], the integrated metamodel is used to generate a domain-specific modeling environment that represents the core of a DSDI.

In the remainder of this section, we show that by defining reference architectures, middleware compo-
nent models, and analysis models using an ACT, architects achieve the following benefits, which directly address the challenges discussed in Section 3:

- traceable characterization of metamodel relationships, which improves the semantic consistency of composed metamodels;
- automation of metamodel composition activities, which avoids error-prone manual composition; and
- constraints that define a set of well-formed metamodels, which prevents mistakes and omissions.

Characterization of metamodel relationships. The designer of an ACT can deduce some compositional relationships between DSML constructs that are instances of ACT metatypes. For example, each component type in the reference architecture must be implemented as one of the component types provided by the chosen middleware platform. However, reference architecture components may be subject to restrictions on their relationships that are not true of generic middleware components. Thus, we can infer that each reference component type has an implementation inheritance relationship with one or more middleware component types.

Other relationships can be inferred to exist between the constructs of reference architecture, middleware, and analysis metamodels. These relationships can be codified into metamodel composition rules, as demonstrated in Table 1. The set of relationships listed is intended to be instructive, rather than definitive or complete. The exact set of relationships and composition rules will depend on the precise syntax and semantics of the ACT.

Table 1: Metamodel Composition Rules

<table>
<thead>
<tr>
<th>Source Element</th>
<th>Destination Element</th>
<th>Relationship</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware component</td>
<td>Reference component</td>
<td>Implementation inheritance</td>
<td>At least 1</td>
</tr>
<tr>
<td>Analysis component</td>
<td>Middleware component</td>
<td>Implementation inheritance</td>
<td>At least 1</td>
</tr>
<tr>
<td>Middleware interface</td>
<td>Reference interface</td>
<td>Full inheritance</td>
<td>At most 1</td>
</tr>
<tr>
<td>Analysis interface</td>
<td>Middleware interface</td>
<td>Full inheritance</td>
<td>At most 1</td>
</tr>
<tr>
<td>Middleware group</td>
<td>Reference group</td>
<td>Implementation inheritance</td>
<td>Any</td>
</tr>
<tr>
<td>Analysis group</td>
<td>Middleware group</td>
<td>Implementation inheritance</td>
<td>Any</td>
</tr>
<tr>
<td>Middleware behavior</td>
<td>Reference behavior</td>
<td>Interface inheritance</td>
<td>Exactly 1</td>
</tr>
<tr>
<td>Analysis behavior</td>
<td>Middleware behavior</td>
<td>Interface inheritance</td>
<td>Exactly 1</td>
</tr>
</tbody>
</table>

Automation of metamodel composition. Inferred relationships allow the composition of ACT metamodels to be (partially) automated. Each metamodel must be labelled as a reference architecture, middleware, or analysis model. The composition proceeds by identifying pairs of elements in different metamodels whose metatypes are the same; these represent candidates for composition. By referring to a table of rules, such as the one given above, the MDE environment inserts the appropriate composition operator into the composed metamodel.

Figure 3 illustrates a composition of the metamodel excerpts introduced in Figure 1. The ACT metatype of each metamodel entity is shown as a stereotype. Using a set of composition rules like those listed in Table 1, relationships between the types of different metamodels may be automatically specified. For example, whereas before an architect needed to manually specify the relationship between the services offered by components in a reference architecture and the interfaces provided by a middleware, an MDE tool utilizing an ACT can infer the relationship on the basis that the metatype of both is Interface. Without the use of an ACT, an MDE tool has no way (in the general case) of determining whether an entity is a component, an interface, or some other type.

Definition of compositional constraints. Just as an ACT designer can infer relationships between metatypes, the designer can also determine a set of relationships that can never hold, and codify this knowledge as a set of automatically enforceable constraints. For example, an ACT might disallow an interface from containing a component.

4.2. Model Transformation

Our objective is to leverage the semantics of ACT metatypes to simplify interpreter development and address the challenges described in Section 3.2 of trans-
Interpret object metatype. In turn, this would require that the MIDAS archi-
caltered to accommodate this domain-specific require-
logic within the interpreter would need to be directly
the conventional fashion, the intricate transformation
MW interpreter were built as a monolithic component in
proprietary modular virtual machine (MVM). If the Prism-
cate objects from a memory pool implemented by a pro-
real-time requirements, MIDAS applications must allo-
transformations every time they define a new DSML.

MIFs offer the same benefits as traditional applica-
tion frameworks: reusability, extensibility, modularity,
and inversion of control [9]. Below we discuss how
employing an MIF as the basis for domain-specific
model interpreters results in:

- the ability to apply off-the-shelf transformation
  engines to domain-specific models;
- decreased maintenance requirements for interpreters;
- reusable modules that encapsulate transformation
  logic.

Simplified interpreter design and implementa-
tion. An MIF may be extended only in predefined ways.
This controlled extensibility promotes structured inter-
preter development and systematic implementation of
domain-specific analysis techniques, without having to
implement complex model transformations. A single
MIF can thus be easily customized for a variety of pur-
poses. For example, an MIF that produces state-based
models could be used for safety or performance analysis.

Decreased maintenance requirements. An MIF
inherently creates modularity within interpreters by
abstracting away the complex implementation details of
a semantic transformation, and by decoupling the
domain-independent interpretation logic from domain-
specific interpretation logic. The potential impact of any
given change is thus scoped, while the complexity of
each module is greatly reduced. Moreover, MIFs are
independent of one another, so no integration of model
transformations is necessary.

Improved reusability of interpreter logic. MIFs
define semantic transformations from architectural mod-
els to analysis models by relying on the semantics
defined for ACT metatypes. Since these semantics do
not vary across domains and platforms, MIFs can be
built into MDE tools or provided off-the-shelf and then
applied to domain-specific models. For example, an MIF
might transform an architectural model into a Petri Net,
Markovian model, or process algebra. The result is that
architects do not need to discover, implement, and vali-
date these transformations themselves, or redevelop
transformations every time they define a new DSML.

5. ACT and MIF Realization

Thus far, we have described a generalized solution
approach to constructing DSDIs that leverages an ACT

Figure 4. The high level design of an MIF.

forming domain-specific architectural models into either
formal analysis models or code. To that end, we propose
the construction of model interpreter frameworks
(MIFs). An MIF is a partially complete model inter-
preter. It implements the logic necessary to convert ACT
types into analysis types or implementation constructs,
but leaves open extension mechanisms that permit an
architect to inject additional logic for handling domain
concepts. This allows architects to implement domain-
specific analysis and synthesis.

The high-level design of an MIF is depicted in Fig-
ure 4. The MIF is an active component that navigates
through an architecture model, interpreting model ele-
ments based on their metatypes. For each ACT meta-
type, the MIF implements a transformation into
constructs of the target language based on domain-inde-
dendent semantics. At specific points in the interpreta-
tion process, the MIF invokes extension points (“hook
methods”) that may be implemented in a framework
extension. Framework extensions transform model ele-
ments into constructs of the target language based on
domain-specific semantics defined by the architect.

The MIDAS DSDI case study introduced in Section
2 offers a good example of how an MIF can be used. The
MIDAS DSDI uses a model interpreter to generate
Prism-MW code. Most of this code can be synthesized
according to the “standard” (i.e., domain-independent)
transformation logic. However, due to domain-specific,
real-time requirements, MIDAS applications must allo-
cate objects from a memory pool implemented by a pro-
prietary modular virtual machine (MVM). If the Prism-
MW interpreter were built as a monolithic component in
the conventional fashion, the intricate transformation
logic within the interpreter would need to be directly
altered to accommodate this domain-specific require-
ment. In turn, this would require that the MIDAS archi-
tects understand the transformation process implemented
within the interpreter, and it would restrict the modified
interpreter to systems that use the MVM. Also, the origi-
nal and new interpreters would need to be maintained
separately. On the other hand, if the Prism-MW inter-
preter is implemented as an MIF, the code that performs
resource pooling may be synthesized by a framework
extension, eliminating all of the above difficulties.

MIFs offer the same benefits as traditional applica-
tion frameworks: reusability, extensibility, modularity,
and inversion of control [9]. Below we discuss how
employing an MIF as the basis for domain-specific
model interpreters results in:

- the ability to apply off-the-shelf transformation
  engines to domain-specific models;
- decreased maintenance requirements for interpreters;
- reusable modules that encapsulate transformation
  logic.

Simplified interpreter design and implementa-
tion. An MIF may be extended only in predefined ways.
This controlled extensibility promotes structured inter-
preter development and systematic implementation of
domain-specific analysis techniques, without having to
implement complex model transformations. A single
MIF can thus be easily customized for a variety of pur-
poses. For example, an MIF that produces state-based
models could be used for safety or performance analysis.

Decreased maintenance requirements. An MIF
inherently creates modularity within interpreters by
abstracting away the complex implementation details of
a semantic transformation, and by decoupling the
domain-independent interpretation logic from domain-
specific interpretation logic. The potential impact of any
given change is thus scoped, while the complexity of
each module is greatly reduced. Moreover, MIFs are
independent of one another, so no integration of model
transformations is necessary.

Improved reusability of interpreter logic. MIFs
define semantic transformations from architectural mod-
els to analysis models by relying on the semantics
defined for ACT metatypes. Since these semantics do
not vary across domains and platforms, MIFs can be
built into MDE tools or provided off-the-shelf and then
applied to domain-specific models. For example, an MIF
might transform an architectural model into a Petri Net,
Markovian model, or process algebra. The result is that
architects do not need to discover, implement, and vali-
date these transformations themselves, or redevelop
transformations every time they define a new DSML.

5. ACT and MIF Realization

Thus far, we have described a generalized solution
approach to constructing DSDIs that leverages an ACT

Figure 4. The high level design of an MIF.
5.1. ACT Realization in XTEAM

XTREAM defines an ACT with these metatypes:

- 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. A component either contains a sub-architecture (consisting of lower-level components and connectors), or it contains a behavioral model (defined in terms of processes). Components interact with other components and connectors through interfaces.
- A Connector implements communication and coordination facilities [22]. Like components, connectors interact with external entities via interfaces.
- A Group represents a logical grouping of components and connectors. It specifies membership in a set whose semantics are defined by domain architects. For example, groups could represent elements that access a pool of shared resources.
- An Interface represents an interaction point between a component or connector and external entities. It defines 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.
- An Event represents the input or output of a datum on an interface.
- A Process represents a unit of behavior within a component or connector.

A simplified XTEAM ACT meta-metamodel is illustrated in Figure 5. This diagram shows the principal structural elements and their containment relationships. In the interest of readability, this diagram has omitted a number of details. First, the Process and Datum elements are actually implemented as references to a behavioral and a data metamodel, respectively. Processes in XTEAM are defined using the Finite State Processes (FSP) formalism [18], which concisely captures the behavior of concurrent systems [19]. Data in XTEAM is modeled using UML Class Diagrams. Second, the ability of each type to participate in inheritance relationships is not depicted in the diagram. Third, syntactic and presentation details, such as the way each element is rendered in the XTEAM metamodeling environment, have been elided. Finally, constraints that ensure the well-formedness of metamodels, specified in UML’s Object Constraint Language [25], are not shown.

XTREAM provides a metamodeling environment for architects to construct reference architecture, middleware, and analysis metamodels using the XTEAM ACT. Naturally, we used an MDE tool to implement the XTEAM metamodeling environment. The process is illustrated in Figure 6. We defined the XTEAM ACT meta-metamodel using the metamodeling language of GME. In a sense, we employed GME’s metamodeling language not as a metalanguage but as a meta-metallanguage. Next, GME’s metamodel interpreter transforms the ACT meta-metamodel into a language description file that GME uses to create a modeling environment in which models that conform to the ACT can be created – this is the XTEAM metamodeling environment. Although the GME-based meta-metamodel of the ACT is a purely syntactic description, the semantics of the ACT are applied through two XTEAM metamodel interpreters. One metamodel interpreter applies the ACT semantics and composition rules (such as those listed in Table 1) to metamodels to perform automatic compositions. The other metamodel interpreter transforms a (possibly composed) ACT metamodel into a domain-specific language description file (just like the standard GME metamodel interpreter, except that it uses a different source metalanguage). System architectures are then modeled using this domain-specific language.

The composition of the three metamodels introduced in Figure 1 can be automated by applying composition rules derived from the XTEAM ACT semantics, according to the process explained in Section 4. The

![Figure 5. Simplified version of the XTEAM ACT.](image)

![Figure 6. Conceptual depiction of the XTEAM metamodeling and modeling processes.](image)
result is the integrated metamodel shown in Figure 3, which serves as the foundation for a number of desirable outcomes associated with DSDIs, including:

- The component types provided to the modeler are exactly those defined by the reference architecture. Automatically enforced constraints on the interactions between component types ensure type safety.
- Middleware configuration data that is normally specified in code can instead be input into the model, automatically checked for consistency, and then used to generate the actual configuration files.
- Designs can be easily mapped to performance models that predict system throughput, resource utilization, and end-to-end latency.

5.2. MIF Realization in XTEAM

XTEAM implements two MIFs. The first MIF transforms architectural models into discrete event simulations (DEVS) [28]. The second MIF generates Prism-MW code from architectural models. The XTEAM DEVS MIF provides (via extensions) several dynamic architectural analyses, including performance [32], reliability [26], and energy consumption [30]. The implementation of the Layered Queueing Network (LQN) performance analysis on top of the DEVS MIF provides a representative view of the process of implementing a domain-specific analysis technique. First, the LQN analysis model is codified within a metamodel using the ACT metatypes. Next, this metamodel is composed with reference architecture and middleware metamodels. This results in the system parameters required by the LQN simulation to be associated with the appropriate elements in the domain-specific architectural model. Utilizing the extension mechanisms in the DEVS MIF, an architect inserts logic that calculates performance metrics as the simulation proceeds. For example, using the Strategy [10] pattern, the architect can insert code that is invoked whenever a component sends or receives an object via an interface. By recording the times of these events, the response times for service requests can be easily calculated. Thus, the architect has implemented an end-to-end latency analysis without having to develop a model transformation.

The XTEAM Prism-MW MIF synthesizes system implementations from architectural models whose metamodel (1) conforms to the XTEAM ACT and (2) includes (via composition) the Prism-MW component model (recall Figure 1b). The generated system implementation consists of component logic and “glue code” in C++, as well as middleware configuration files that specify quality-of-service (QoS) settings such as the size of resource pools.

Extension points in the Prism-MW MIF permit the customization of the synthesis process. The Prism-MW middleware platform is itself designed to be highly extensible, so the extension capabilities within the framework complement and enhance the corresponding features in the middleware. Prism-MW allows a developer to create application-specific behaviors for a variety of concerns, including architectural awareness, real-time requirements, distributability, security, heterogeneity, data compression, delivery guarantees, and mobility [20]. The model transformation logic necessary to automatically instantiate and use such application-specific extensions can be incorporated into the Prism-MW MIF using the Template Method design pattern [10]. For example, Prism-MW provides an extension mechanism that allows applications to transparently substitute the default IP network protocol with domain-specific network and transport protocols, such as the Controller Area Network (CAN) protocol commonly used in the automotive domain. Data required to setup and configure a CAN network can be included in architectural models by defining the appropriate metamodel elements. This data can then be interpreted by our Prism-MW MIF extension for CAN that automatically generates the code to perform the CAN network management.

6. Discussion and Evaluation

Previous research [3,12,17] has established the benefits of utilizing a DSDI for the development of large-scale distributed architectures. In this paper, we have argued that metamodel composition and interpreter construction can be automated to a large extent, reducing the cost and effort required to build DSDIs. This section attempts to quantify these benefits.

To date, we have constructed three DSDIs – MIDAS [21], Bold Stroke [29], and XTEAM [8] – using the approach described in Section 4, and realized them via our framework from Section 5. MIDAS, a DSDI for the domain of sensor network-based applications, was discussed and several of its facets illustrated throughout the preceding sections. Bold Stroke is a DSDI targeted at the avionics domain. We have obtained and modeled its reference architecture from publicly available documentation, and have integrated it with the RT CORBA middleware component model [4]. Finally, as indicated in Section 5.1, XTEAM is a DSDI for the domain of DSDI construction, and as such was (recursively) applied on itself. Details of these three DSDIs are omitted due to space constraints, but can be found at [33].

As postulated in Section 4, the use of the XTEAM ACT completely eliminated or significantly reduced the number of metatype relationships that needed to determined and specified for each DSDI. The ACT also served as the foundation for the MIFs developed for the three DSDIs. We discuss several specific quantitative
benefits incurred in the process below.

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 recreated. In contrast, the manual development of model interpreters requires implementing the transformation logic of both MIF extensions and an MIF core. Therefore, to quantify the advantage of using an MIF, we compared the complexity of our MIF extensions to the complexity of the corresponding MIF cores.

Our complexity measurements do not include a large amount of peripheral code included within the MIF that performs functions such as data structure initialization and interface adaptation. We exclude this code because it is reasonable to assume that a developer may reuse such code even without an ACT and MIF. On the other hand, the core transformation logic cannot be readily reused without the use of a strategy, such as the one proposed in this paper, that enhances in some way the standard interpreter development paradigm.

Figure 7 graphs the relative complexities of each of the model interpreter frameworks and their extensions along eight dimensions. The first seven metrics show average and maximum values for commonly-used complexity indicators [14]. 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. Because the units of each metric are different, the values have been normalized to permit them to be displayed on a single graph.

Inspection of the data indicates that (1) the size of the implemented code for a MIF core is an order of magnitude larger than the size of framework extensions, and (2) the code in an MIF core is more complex than the code in an MIF extension according to nearly every metric (the one exception is average methods per class). 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. Furthermore, this result directly addresses the challenges outlined in Section 3.2.

7. Related Work

This section compares our work to other architectural languages with metamodeling capabilities, and with tools that seek to improve model transformation.

xADL [7] is an extensible architecture description language based on modular and composable XML schemata. xADL provides some of the features of an ACT: it allows architects to create domain-specific architectural modeling languages and enables features from different languages to be combined. However, xADL schemas function as metamodel fragments whose metalanguage is XML. As such, xADL provides essentially the same capabilities as other MDE environments, such as GME and MetaEdit+ [23]: it does not constrain the creation of metamodels or associate any semantics with domain-specific metamodel elements.

The Cadena Architecture Language with Metamodeling (CALM) [13] is an ACT implemented by the Cadena MDE environment. Cadena allows architects to create domain-specific architectural languages and enforces type-based constraints specified in metamodels. CALM and Cadena offer many of the benefits of an ACT. CALM’s set of specialized metatypes simplifies the construction of metamodels. Moreover, CALM leverages metatype semantics to support automated metamodel composition and built-in type checking and model checking. However, CALM focuses on structure alone, while many important architectural constraints and quality attributes depend on behavior. Also, CALM composes metamodels by defining “bridging elements”, which allow interaction between languages (metamodel interfacing) rather than their semantic integration (metamodel merging).
The Graph Rewriting and Transformation (GReAT) toolset [1] and the Visual Automated model TRAnSformations (VIATRA) framework [6] enable the specification of model transformations in graphical languages. Automated tools leverage these specifications to generate model interpreters that implement the corresponding transformations. However, neither approach aids in determining relationships between languages, achieves automated validation based on semantics, or provides convenient reuse of transformation logic.

8. Conclusion

This paper presented a methodology for creating DSDIs by (1) composing the metamodels of reference architectures, middleware platforms, and analysis technologies, and (2) applying model transformations that perform analysis and synthesis. We explained how a lack of automated mechanisms for these two DSDI integration activities leads to a number of practical challenges. Our methodology improves the support within MDE environments for automated composition and validation of metamodels and automated model transformation by integrating and leveraging semantics both at the metamodeling phase and the interpretation phase of the MDE paradigm. This paper also described how we implemented our methodology in the XTEAM framework, and evaluated the utility of XTEAM in the context of a DSDI for sensor network applications.

9. Acknowledgments

This work has been supported by Robert Bosch RTC and by the U.S. National Science Foundation under grant numbers 0312780, 0509539, and 0720612.

10. References