Automated Analysis and Code Generation for Domain-Specific Models

George Edwards
University of Southern California
gedwards@usc.edu

Yuriy Brun
University of Massachusetts
brun@cs.umass.edu

Nenad Medvidovic
University of Southern California
neno@usc.edu

***Abstract*—Domain-specific languages (DSLs) concisely express the essential features of system designs. However, using a DSL for automated analysis and code generation requires developing specialized tools. We describe how to create model analysis and code generation tools that can be applied to a large family of DSLs, and show how we created the LIGHT platform, a suite of such tools for the family of software architecture-based DSLs. These tools can be easily reused off-the-shelf with new DSLs, freeing engineers from having to custom-develop them. The key innovation underlying our strategy is to enhance DSL metamodels with additional semantics, and then automatically synthesize configurations and plug-ins for flexible analysis and code generation frameworks. Our evaluation shows that, for a DSL of typical size, using our strategy relieves software engineers of developing approximately 17,500 lines of code, which amounts to several person-months of programming work.**

I. Introduction

Some software-intensive systems, such as aerospace systems and sensor network applications, require rigorous design modeling. *Domain-specific* modeling tools and languages allow meticulous design analysis and automatic code generation, improving system quality. In contrast to standardized modeling languages like UML, domain-specific languages (DSLs) allow engineers to focus on the design decisions relevant to the domain and use the most suitable concepts and abstractions.

Today's model-driven engineering (MDE) platforms, such as the Generic Modeling Environment (GME) [10] and the Eclipse Graphical Modeling Framework (GMF) [11], ease the creation of custom model editors for DSLs. Software engineers only need to define a *metamodel* — a formal specification of a DSL — and these platforms automatically synthesize a model editor that uses the DSL's symbols and enforces its syntax.

Nevertheless, industry adoption of domain-specific modeling technologies has been more limited than that of standardized modeling solutions, particularly UML. One key reason for this disparity is that DSL *analysis* and *code generation* tools (often referred to as *model interpreters*) must be constructed manually. Meanwhile, UML-based analysis and code generation tools are available off-the-shelf. While domain-specific tools can perform more targeted analysis and more complete code generation, the difficulty of tool creation and maintenance reduces the appeal of domain-specific modeling, particularly for small- and medium-scale software systems [18].

In this paper, we present a solution that greatly reduces the costs of model interpreter creation and maintenance. We show how specific enhancements to metamodels can allow an MDE platform to automatically synthesize analysis, simulation, and code generation tools, just as existing MDE platforms automatically synthesize model editors. While our approach applies to DSLs in general, in this paper we focus on software architecture-based modeling. The field of software architecture is characterized by a large number of DSLs, arising from different architectural styles, design patterns, modeling notations, analysis tools, middleware platforms, and frameworks [23]. We implement our approach in an MDE platform called LIGHT (Leveraging Isomorphism to Generate Heterogeneous DSL Toolchains). LIGHT allows engineers to customize architecture-based DSLs for their particular project modeling needs and then automatically synthesize (1) a model editor, (2) a system simulator for analyzing latency, memory usage, energy consumption, and reliability, and (3) a middleware-based system implementation. LIGHT eliminates a substantial amount of tool building and maintenance work required by existing MDE platforms.

Consider a team designing the software module for a moon-landing spacecraft. The team wants to (1) create models to formalize their designs, (2) evaluate those designs in simulation, and (3) automatically generate code from the model to ensure the implementation conforms with the model. Recognizing the highly specialized nature of the software and the need to focus on design decisions specific to the domain, the team decides to use an architecture DSL.

Using existing MDE tools, the team specifies the DSL metamodel and uses the generated editor to develop a series of architecture models. They then develop a custom model interpreter that translates their domain-specific models into simulation code. As we discuss in Section V, this coding effort can amount to as much as four person-months of work for a DSL of moderate size and complexity.

In contrast, using LIGHT, the team only needs to specify a slightly expanded DSL metamodel, and LIGHT generates the model editing, simulation, and code generation tools automatically from the metamodel. If the team makes changes to the DSL, the tools are automatically updated to conform to the new semantics. As we discuss in Section V, the effort required to create the expanded metamodel is minor compared to the effort of creating model interpreters.

The insight that allows LIGHT to automate model interpreter creation and maintenance is that model editors and model interpreters are *isomorphic* [6]: rendering models within an editor is just another form of model interpretation. The implication of this insight is that model editors and interpreters can be treated as analogs conceptually and architecturally — a hypothesis we set out to validate by designing, implementing, and evaluating LIGHT.

978-0-7695-4827-2/12 $26.00 © 2012 IEEE
DOI 10.1109/WICSA-ECSA.212.24

Our work's contributions are a *novel architecture* for creating modeling, analysis, and code generation toolchains from DSL metamodels; LIGHT, an *instantiation of the architecture* targeted at architecture-based development; an *evaluation* of LIGHT's utility across nine different architectural DSLs; and an *elaboration of the trade-offs* associated with LIGHT.

In the rest of the paper, Section II outlines the problem we aim to solve; Section III describes our solution and its applicability, while Section IV details the implementation of LIGHT; Section V evaluates LIGHT; Section VI compares our approach to related work; Section VII concludes the paper.

II. The DSL Model Interpreter Problem

Throughout this paper, we will rely on Lunar Lander (LL), the moon-landing spacecraft software introduced in Section I, to illustrate important concepts. LL has been used as an instructional tool for software architecture concepts, via its many variations [23]. Suppose an engineering team is tasked with building LL. They analyze the requirements and arrive at high-level design goals, including: components should be independent (facilitating reuse) and dynamically configurable (enabling runtime system adaptations), and critical components should be replicable (trading off efficiency for reliability).

Based on these goals, the team employs the Myx [2] architectural style. Myx supports layered architectural composition, flexible construction of distributed systems, and their dynamic adaptation. Furthermore, the team elects to rigorously model candidate designs to (1) document the architecture, (2) evaluate alternative designs with respect to latency and reliability, and (3) generate code for the implementation. For example, the team would like to explore alternative component replication strategies in an off-the-shelf simulation tool.

Recognizing LL's highly specialized nature and the need to focus on design decisions specific to the domain, the team uses a DSL to model candidate designs. Using a DSL frees the team from the constraints and feature bloat of a standardized language like UML, allows them to customize the look-and-feel of diagrams, and allows a single model to contain all relevant information (e.g., parameters needed for latency analysis).

The team decides to use an MDE platform, such as GME or GMF. To do so, the team specifies a metamodel for their DSL in a provided metamodel editor. The metamodel encodes the rules of the Myx style and defines the necessary latency and reliability analysis parameters. The metamodel is specified using the MDE platform's metamodeling language (or *metalanguage*); objects in the metamodel are instances of the metalanguage types (called *metatypes*). Each metatype instance captures the definition for a domain-specific type. For example, the team can define a `MyxLink` type in their DSL that represents an association between component interfaces. In a GME metamodel, the `MyxLink` type could be an instance of the `Connection` metatype. In the team's models of candidate LL designs, individual associations between components are then represented by instances of the `MyxLink` type.

Using the metamodel, existing MDE platforms automatically create a custom model editor by generating configuration files or plug-ins for a *model editor framework*. The editor renders the model and allows model manipulations while enforcing the DSL constraints. For example, instances of the `MyxLink` type might be rendered as dashed lines. Using an MDE platform in this way allows the LL team to automatically generate a custom graphical editor for their DSL.

However, existing MDE platforms provide only a partial remedy: they do not provide built-in analysis and code generation support. Instead, they require engineers to implement custom model interpreters. For example, if the LL team wants to use an off-the-shelf simulation tool (e.g., MATLAB) to analyze candidate designs, they must implement an interpreter that translates their DSL models into the tool's input format. Similarly, to generate code for a runtime platform (e.g., .NET), the team must implement an interpreter to produce that code. While some existing MDE platforms provide specialized APIs and languages for implementing interpreters (for example, Java Emitter Templates used with GMF), these only reduce, and do not eliminate, interpreter implementation effort. MDE platforms cannot automate this process because the metatypes provided by these platforms lack sufficient semantics to be automatically mapped to other forms. Their semantics are limited to those needed to synthesize model editors (i.e., map metatypes to graphical display elements). As we show in Section V, creating the interpreter to generate executable simulations from Myx models requires the LL team to write over 17.5K non-trivial lines of code.

Thus, the consequences of existing MDE's limitations are: (1) Analysis and generation tools must be manually constructed, which is difficult [13], [21]. (2) Language and tool reuse is hard, as using them in new contexts can require significant rework. (3) The maintenance of domain-specific analysis tools and code generators is burdensome, as they must be updated whenever the corresponding DSL changes.

III. Automated Synthesis of Model Interpreters

This section describes our approach to building an MDE platform. The approach is aimed at drastically reducing the effort required to create end-to-end domain-specific modeling toolchains. Our guiding idea is to leverage additional semantics within metamodels to enable generation of configuration files and plug-ins for extensible analysis and code generation frameworks. Our approach comprises three activities: (1) the MDE platform developers select the analysis and code generation capabilities the platform will support, (2) they extend the semantics of the platform metatypes to enable those capabilities, and (3) they create a *metainterpreter* and *model interpreter framework* that together perform the synthesis of tools that implement those capabilities. This process only needs to be carried out by MDE platform *developers*; platform *users* (such as the LL team) obtain the MDE platform off-the-shelf and automatically generate tools for their DSL.

A. Capability Selection

The developers of an MDE platform must first select which analysis and code generation capabilities to support, based on the expected usage of the platform. This decision is fundamental to our proposed MDE platform architecture because it determines which semantics the metamodels will include. We do not address the process of making this decision in this paper and instead focus on how to implement an already chosen set of capabilities. For our LIGHT reference implementation of the approach, we chose to support a model editor, a middleware platform, and a simulation engine that analyzes latency, memory usage, energy consumption, and reliability.

B. Metatype Semantics Extension

Whereas today's approaches require engineers using a DSL to manually build model interpreters that encode semantics, our approach embeds these semantics in the DSL metamodel. We describe these metamodels in Section IV-A. A set of metainterpreters then automatically generates the model interpreters from the metamodel. We describe the metainterpreters in Section IV-B. Since the metamodel is crucial for the interpreter generation, it is important to determine the exact semantics the metamodel needs to capture. To automatically synthesize model interpreters, the metamodel must define a complete *semantic mapping* from domain-specific elements to target platform elements (e.g., graphical elements for an editor or language constructs for a code generator).

For example, consider the `MyxLink` DSL type from the Myx metamodel developed by the LL team. In an existing MDE platform such as GME, this type might be defined by an instance of the `Connection` metatype. GME is pre-programmed with the *presentation semantics* (model editor behavior) of instances of the `Connection` metatype: drawing a line between two objects. Thus, GME maps `MyxLink` instances in LL models to classes that render a dashed line. GME does not know, e.g., the *simulation semantics* of instances of the `Connection` metatype. On the other hand, using our approach, as implemented in LIGHT, `MyxLink` is an instance of a specially defined `Link` metatype that includes presentation semantics as well as simulation and *code generation semantics*.

Our approach directly attaches semantics to metatypes as (1) semantic *assumptions*, which are behavior definitions that hold for all domain-specific types that are instances of that metatype, and (2) semantic *properties*, which are typed attributes and associations with other metatypes that allow an engineer to select among various behavior options. For example, in LIGHT, a semantic assumption is that all domain-specific types defined by instances of the `Link` metatype exhibit the behavior of transferring data between two other entities. A semantic property of the `Link` metatype, called `capacity`, allows engineers to specify, in the metamodel, whether the data transfer channel can become full and the resulting behavior.

This approach has the advantage that metamodel developers do not need to write intricate formal specifications. However, the metatype assumptions and properties are chosen by the metalanguage designers, thus the space of semantics that can be captured is fixed. This results in a trade-off between the ability to synthesize supporting toolsets and DSL flexibility. Attaching additional semantics to metatypes increases the space of model editors, analysis engines, and code generators that can be synthesized. For example, current MDE platforms only synthesize model editors because their metatype semantics include only visualization and editing concerns. On the other hand, attaching additional semantics to metatypes decreases the space of DSLs that can be specified in a domain-specific modeling platform. For example, if metatypes include fixed semantics for graphical rendering and editing, they cannot be (easily) used to specify a textual language. Section IV-A discusses the semantics captured in LIGHT's metalanguage.

C. Interpretation Component Development

Our approach fundamentally differs from existing approaches, including our own previous work, in its mechanism for model interpretation. The approach does not require any manually coded components to perform system analysis or to generate executable code. Figure 1 depicts the roles and interactions of the MDE platform components that participate in analysis and code generation. Each interpretation capability is implemented through a paired *metainterpreter* and *model interpreter framework* (MIF). As noted earlier, we have implemented three such interpretation capabilities in LIGHT (a model editor generator, a simulation generator, and a code generator) using this general architecture. We summarize the function of metainterpreters and MIFs here, while Section IV-B discusses their details.

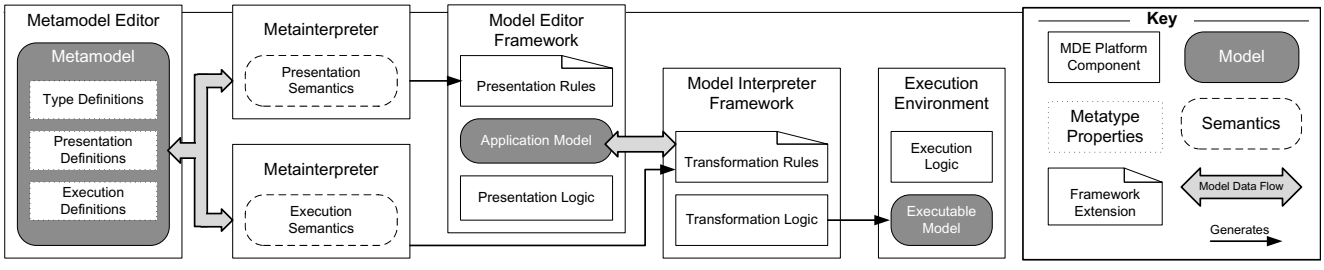


Figure 1. Our approach to automated tool generation for DSLs.

Existing MDE platforms lack MIFs and their associated metainterpreters. Metainterpreters (1) take as input a metamodel, including the metatype properties needed to map domain-specific models to a target platform, (2) use those metatype properties to derive domain-specific type semantics, and (3) determine a set of rules for transforming each domain-specific type to the analysis, simulation, or execution platform language. Metainterpreters encode the transformation rules in an automatically-generated *MIF extension* for a specific MIF; the MIF implements the actual transformation logic.

Each MIF is a template for a family of model interpreters. To be reusable, a MIF encapsulates transformation logic or algorithms that are useful in a variety of contexts and can be flexibly applied in different ways to achieve different semantics. A MIF is analogous to a virtual machine in that it provides and executes an instruction set composed of model transformation operations. Transformation steps that only depend on semantic assumptions (i.e., not metatype properties) are hard-coded into the MIF, while transformation steps that vary based on metatype properties are programmable via extension points. In this way, our approach is distinct from a product-line approach, as new tools ("products") are built from custom generated code, rather than compositions of pre-built components.

A domain-specific type's semantics are a subset of all possible semantics permitted by the MDE platform; each possible semantic option corresponds to a different usage of MIF extension points. Therefore, a metainterpreter includes (1) a mapping of the metatype instance property values to a set of semantic definitions, and (2) a mapping of semantic definitions to a set of MIF extension point usages. A generated MIF extension, which may take the form of configuration files or plug-ins, modifies, extends, and controls the functionality of the MIF using extension points built into the MIF. The extended MIF converts a domain-specific model into an executable or analyzable program for the target platform.

IV. The LIGHT Platform

To verify the feasibility of our approach, we built the LIGHT MDE platform. LIGHT is intended for software architecture-based modeling, analysis, and code generation. LIGHT automatically generates model interpreters for any DSL defined by a LIGHT metamodel, by configuring a model interpreter framework (MIF) with a domain-specific MIF extension generated by a corresponding metainterpreter. Each such metainterpreter-MIF pair generates a different type of interpreter. LIGHT contains three such pairs that target, respectively, GME's open-source model editor [10], a variant of the Adevs discrete event simulation engine [19], and the Prism-MW middleware platform [17]. Engineers can use LIGHT to create domain-specific models in a customized model editor, analyze those models in the simulator, and generate code for Prism-MW with virtually no tool-building overhead.

To use LIGHT, engineers first create a DSL metamodel via LIGHT's provided metamodel editor. Then, to generate the three interpreters, LIGHT first invokes the appropriate metainterpreter, which produces an MIF extension (a set of C++ plug-in classes) by deriving the simulation or implementation semantics of the DSL types in the metamodel. LIGHT then compiles the provided MIF (also implemented in C++) with the extension. The output of the compilation is a domain-specific model editor, simulation generator, or Prism-MW code generator, already configured with DSL's custom semantics.

Next, we discuss LIGHT's metamodeling facilities (Section IV-A) and the implementation of LIGHT's model interpretation components (Section IV-B). We then focus on using simulations generated by LIGHT to analyze models with respect to latency, reliability, and other qualities (Section IV-C). Details of the Prism-MW code generator are elided for space.

A. Metamodeling

LIGHT maps DSL types to runtime objects (e.g., simulation and Prism-MW objects) through the use of a metalanguage enhanced with semantic assumptions and properties (recall Section III). We designed the LIGHT metalanguage in a two-step process. First, since LIGHT is intended for software architecture-based modeling, analysis, and code generation, we conducted a literature review to identify the common elements, abstractions, and patterns used for architectural models. We identified ten important metatypes: *architecture*, *component*, *resource*, *interface*, *link*, *implementation*, *operation*, *task*, *data type*, and *property*. Second, we defined the semantic assumptions and properties for each metatype. Recall that semantic assumptions are behavior definitions that are inherent to each LIGHT metatype and hold for all domain-specific types that are instances of that metatype (and thus do not need to be specified in metamodels). Semantic assumptions can be further classified as *capabilities* and *responsibilities*. Capabilities describe behaviors that instances of the metatype exhibit by default, while responsibilities describe behavior constraints that instances of the metatype must respect. Semantic properties are typed attributes and associations that capture semantic variations and options. Software engineers customize the semantics of a DSL by setting the values of semantic properties in a LIGHT metamodel.

Figure 2 depicts the ten LIGHT metatypes (and an `Entity` supertype) and their semantic properties. An `Architecture` defines a system as a collection of components, resources, and other types, and their relationships. A `Component` defines a set of interfaces and maps each interface to either the interface of a sub-component or an implementation. Each `Interface` defines a component interaction point. `Implementation`s capture computational logic and state in terms of sequences of instructions (e.g., methods) or state-transition systems (e.g., Finite State Processes). `Link`s represent logical connections among interfaces used to exchange information and control. `Resource`s are entities provided by the execution environment that components use to perform tasks. `Operation`s define component service access points in terms of data and control

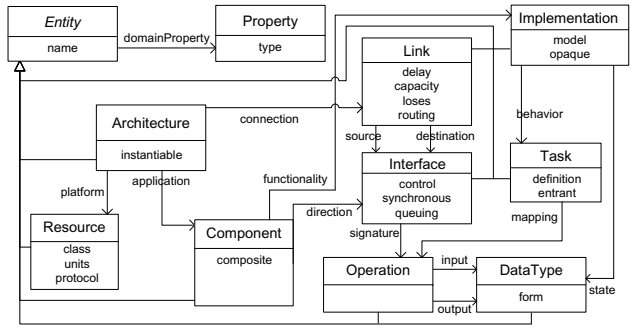


Figure 2. Summary of the LIGHT metatypes.

exchange and are grouped together within interfaces. `Tasks` are reusable units of functionality within a component. Finally, `DataTypes` represent objects exchanged between components or maintained as part of a component's state.

To understand the space of DSLs that can be captured in a LIGHT metamodel (and for which a custom tools can be automatically synthesized) is it necessary to understand the semantic assumptions and properties of LIGHT metatypes. Any language whose semantics are (1) compatible with the semantic assumptions, and (2) within the semantic options provided by the metatype properties can be described. Due to space constraints, we cannot explain the semantic assumptions and properties of all ten metatypes here. Instead, we will highlight three LIGHT metatypes: `Component`, `Interface`, and `Resource` (see Figure 3). We elaborate on these three types next; we refer the reader to [15] for a complete table of all metatypes and [8] for their detailed descriptions.

A LIGHT component is an independently instantiable and deployable unit of computation and information that encapsulates reusable blocks of logic. Domain-specific type definitions for components specify compatible implementation and interface types. Examples of domain-specific component types include "JavaBean," "web server," or, in the case of the LL, "MyxComponent." Components define interaction points in the form of LIGHT interfaces. The domain-specific type definition for an interface restricts the allowed types of data and control exchange between components. Examples of domain-specific interface types include "HTTP port," "publish-subscribe API," or, in the case of the LL, "MyxAsynchronousInterface."

The relationship between an interface and a component has a designated *direction* (recall Figure 2) of either `implements` or `invokes`. The direction affects the semantics of the interface in several ways. First, the flow of information in the two directions is opposite: an invoking component sends the interface's inputs and receives its outputs; an implementing component receives the inputs and sends the outputs. Second, the *mode of interaction* — either *method-based* or *message-based* — is derived from the directions of the interfaces that are connected via a given link: method-based interactions occur when an invoked interface (*e.g.*, an object reference) is linked to an implemented interface (*e.g.*, an object), while message-based interactions occur when two invoked interfaces are linked (*e.g.*, two send-message/receive-message interfaces).

Resources are provided by the computing environment and are used by application implementations to perform tasks. Resources require (simulated) time in order to fulfill task requests, and contention over resources results in the emergent behavior of applications. Each resource may optionally permit an arbitrary level of parallelism; in other words, a resource may be capable of servicing multiple requests simultaneously. Resources are required to specify the available quantity of the resource (the *capacity*), which may be a positive real number if the resource is continuous or a natural number if the resource is discrete. For example, bandwidth may be modeled as a continuous resource, while threads in a thread pool are an example of a discrete resource. Resources may be processing (e.g., CPUs or threads), communication (e.g., network interfaces), or data resources (e.g., files or buffers).

To illustrate how custom semantics are specified in a LIGHT metamodel, we return to the Myx metamodel introduced in Section II. Myx allows for multiple types of interfaces with different semantics. The following semantic definitions are taken verbatim from the Myx specification (emphasis added):

> 1. The default form for an *interface* in Myx is a set of one or more *methods* that can be called.
> 2. Every interface is designated as either a *provided* or a *required* interface...
> 3. *Links* have exactly two endpoints. Each link connects exactly one required interface to one provided interface.
> 4. All bricks have two "domains" called *top* and *bottom*. All interfaces on a brick must be assigned to one of these domains.
> 5. In a *synchronous* invocation ... the calling component passes its thread of control to the called component, [which] completes its invocation and returns control to the calling component.
> 6. In an *asynchronous* invocation, the invoking brick continues processing after initiating the invocation, which proceeds concurrently in a separate thread of control.

The excerpt of the Myx metamodel shown in Figure 4 illustrates how the above semantics are captured in LIGHT. The metatype of each object is indicated using stereotypes, but this is merely a syntactic choice — for example, different shapes or colors could be used to differentiate the metatypes. `MyxInterface`, an abstract base type for other interfaces, declares one or more `JavaMethodDecl` operations (capturing #1 from the table above). The `direction` of the `port` properties of `MyxBrick` is `implements` for `MyxProvidedInterface` and `invokes` for `MyxRequiredInterface` (#2). `MyxSyncLink` and `MyxAsynchLink` connect exactly one `MyxProvidedInterface` to exactly one `MyxRequiredInterface` (#3). `MyxBrick` has `topDomain` and `bottomDomain` properties, which are sets of `MyxInterfaces` (#4). A `MyxSynchronousInterface` passes a thread of control, mandates that the invoker and invokee experience interactions simultaneously, and disallows queueing of interactions (#5). A `MyxAsynchronousInterface`, on the other hand, does not pass a thread of control, allows the invoker to initiate an interaction that is experienced by the invokee at a later time, and allows interaction queueing (#6).

The metamodel for a different architecture DSL would have

	Embedded Semantic Assumptions		Properties	
	Capabilities	Responsibilities	Name (Type)	Description
`Component`	• Manage and prioritize interactions between internal implementations and external links • Multiplex/demultiplex, filter, and monitor interactions • Delegate externally initiated interactions to component implementations • Transmit internally initiated interactions to external entities via established links	• Specify mappings from implemented interfaces to subcomponent interfaces or tasks • Ensure interaction takes place via interfaces and protect internal information and behavior from being manipulated directly	`port` (Interface association)	Specifies the types of interfaces through which the component's implementations may interact
			`functionality` (Implementation association)	Designates the types of implementations that may be used to realize the component's provided services
			`composite` (Boolean attribute)	Indicates that the component is a hierarchy of subcomponents
`Interface`	• Ensure type conformance (data adheres to input and output data type definitions) • Ensure mode conformance (participants in an interaction are uniformly method- or message-based) • Ensure control conformance (an execution thread is transferred iff both source and target interfaces expect it) • Block an execution thread to create synchrony	• Declare at least one operation	`signature` (Operation association)	Indicates the operation types that may be specified by the interface type
			`control` (Boolean attribute)	Enables exchange of control flow across the interface type
			`synchronous` (Boolean attribute)	Forces synchronization across interface; caller(s), callee(s) experience interactions simultaneously
			`queuing` (Boolean attribute)	Allows queuing of interactions over the interface
`Resource`	• Accept, queue, execute, and return service requests • Manage the allocation of pooled or divisible resources to requests • Maximize request execution parallelism as allowed by the metatype property specification	• Specify the available quantity/capacity • Specify a scheduling discipline	`class` (Enumeration attribute)	Specifies whether the resource is a computation, communication, or information (data) resource
			`units` (Enumeration attribute)	Defines whether the units of the resource are continuous or discrete
			`protocol` (Enumeration attribute)	Allows the resource to be held until released by the service requester
			`instantiable` (Boolean attribute)	Permits the resource to be created on demand

Figure 3. Metatype semantics for the `Component`, `Interface`, and `Resource` metatypes. A complete table with all metatypes appears in [15].

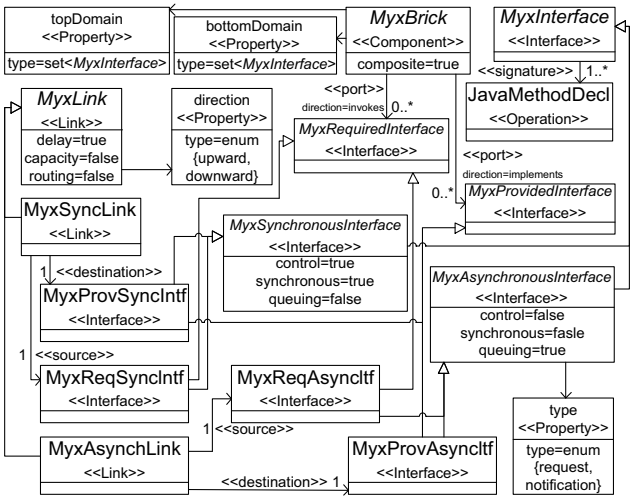


Figure 4. An excerpt of the Myx metamodel that defines `Interface` types.

different types and semantics defined. For example, the metamodel of AADL [9], which is targeted at embedded real-time systems, includes execution platform components, such as processors and memory. AADL Interfaces are called *ports*. One type of port, the *event port*, causes an immediate transfer of control, but also can be queued at the recipient, resulting in different semantics than the interfaces found in Myx.

Using LIGHT, if engineers wish to change the semantics of their DSL, or add new DSL types, they can simply update the metamodel and regenerate the supporting tools. For example, if the LL development team decided that they wanted to alter the standard Myx semantics in some way (e.g., by making `MyxLinks` delay-free), they could do so by changing a single property, as opposed to dealing with low-level interpreter code.

Finally, since the semantic variability permitted by the LIGHT metalanguage is restricted by the set of defined semantic assumptions and properties for each metatype — meaning that some DSL semantics cannot be captured in a LIGHT metamodel — LIGHT provides a "backdoor" mechanism for defining semantics: engineers are able to insert custom behavioral definitions directly at the modeling level. LIGHT allows any default behavior to be overridden with a custom definition by providing a pointer to an external semantic specification file. For example, LIGHT does not provide a means for links to *correlate* interaction events, that is, to hold one event indefinitely until another event satisfying a condition occurs. However, such semantics can be added by specifying event correlating behavior directly in the native language of the target runtime platform. Custom semantics must be directly executable because LIGHT cannot translate this logic and treats custom behavior definitions as a black box.

B. Interpretation

Recall from Section III that domain-specific interpreters are composed of an MIF and generated code for the MIF (an MIF extension). This section focuses on the implementation of MIFs and MIF extensions in LIGHT, with a particular emphasis on the reusable aspects of their design. Given the space limitations, we attempt to highlight the most important elements of MIF design, and omit the details of less interesting aspects of MIFs.

The architecture of LIGHT interpreters uses the *visitor-traverser* style [14]. In accordance with this style, each interpreter is structured as two distinct modules: one to traverse models and another to generate code (see Figure 5). The model traversal module contains the logic for navigating through the model and invoking the code generator on each model object. The code generator module contains the logic for outputting code for the target analysis, simulation, or execution platform.

The use of the visitor-traverser style provides a basis for increasing the flexibility of MIFs by decoupling domain-

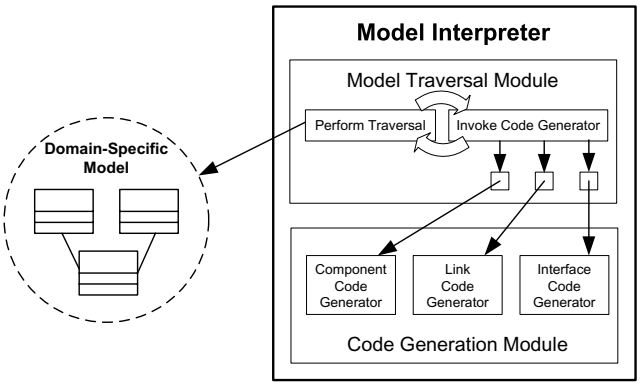


Figure 5. LIGHT model interpreter functions.

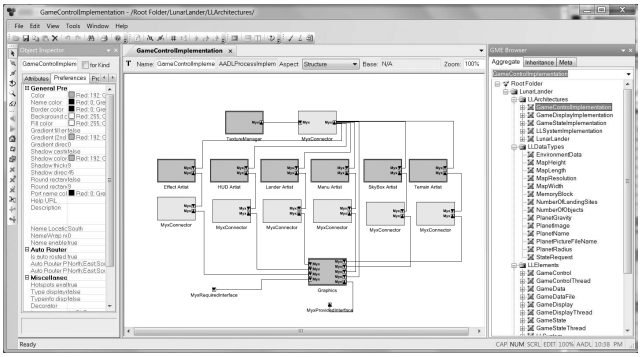


Figure 6. A screenshot of an LL model in the automatically generated Myx model editor.

independent interpretation logic (i.e., logic that does not vary from one metamodel to another) from domain-specific logic (i.e., logic that depends on the types and properties defined in a metamodel). This decoupling is necessary because classes implementing the domain-specific logic must be individually generated and customized for each metamodel.

The code generation module of a LIGHT interpreter is a combination of classes built into the MIF and generated classes in an MIF extension. Returning to Figure 1, the built-in, domain-independent classes are represented by the *Transformation Logic* within the MIF; these classes implement code generation operations. The generated, domain-specific classes are represented by the *Transformation Rules*; these classes contain instructions for applying code generation operations.

At the implementation level, the domain-independent logic is contained within MIF classes corresponding to each metatype, called *metatype classes*, while the domain-specific logic is contained within MIF extension classes corresponding to each metatype instance (i.e., each domain-specific type defined in a metamodel), called *type classes*. The metatype classes define *template methods* that implement specific interpretation tasks, and these methods are selectively invoked and parameterized by the type classes. In this way, the MIF implements the high-level structure of interpretation algorithms but allows customization of the algorithms by MIF extensions.

To illustrate how semantics specified in a LIGHT metamodel are encoded as transformation rules within an MIF extension, we return to the Myx version of the LL model. Recall that, in the Myx architectural framework, component interfaces may be synchronous or asynchronous. Synchronous interfaces have the same semantics as Java method calls, and accordingly the following properties are specified in the Myx metamodel for the `MyxSynchronousInterface` type: `control=true`, `synchronous=true`, `queuing=false`. Consequently, the Myx MIF extension for LIGHT's simulation generator MIF contains a `MyxSynchronousInterface` type class, which invokes the `generateThreadPasser()` method of the `Interface` metatype class, but does not invoke its `generateEventQueue()` method.

The described design results in several important benefits:

- The design factors out the common structure of model interpretation algorithms to simplify code reuse, while still providing a straightforward way for domain-specific logic to customize the interpretation process.
- The design allows the MIF to limit and control the ways in which interpretation may be customized, ensuring that a MIF extension does not "break" the interpretation process or violate the constraints of the target runtime platform.
- The design ensures consistency of metatype properties and avoids duplication of common properties by allowing metatype properties and domain-specific properties to be separated within distinct classes.

C. Simulation

LIGHT automatically generates fully configured, domain-specific interpreters that today have to be programmed manually. One type of interpreter generated by LIGHT is simulation generators for XDEVS, a stand-alone simulator for analyzing the dynamic behavior of complex systems [7]. XDEVS is a component-based variant of the widely used Adevs [19] event-based simulation platform. LIGHT's simulation generators consist of the XDEVS MIF (built into LIGHT) and domain-specific XDEVS MIF extension code (autogenerated by LIGHT). We have used the LIGHT-generated XDEVS simulators to perform latency [24], memory usage, energy consumption [22], and reliability [20] analyses. Here, we briefly describe how simulation can be used to analyze the Myx LL.

Suppose an engineer wishes to evaluate how replication of a critical LL component might affect performance and reliability. While running an additional replica will likely require more resources and decrease performance, the system reliability will improve. Comparing the two possible designs is difficult because performance and reliability depend on the complex interactions of numerous components, the execution environment, and other factors.

Figure 6 shows a screenshot of the automatically generated Myx model editor with one of the two possible LL designs. We used LIGHT to generate XDEVS simulation code from

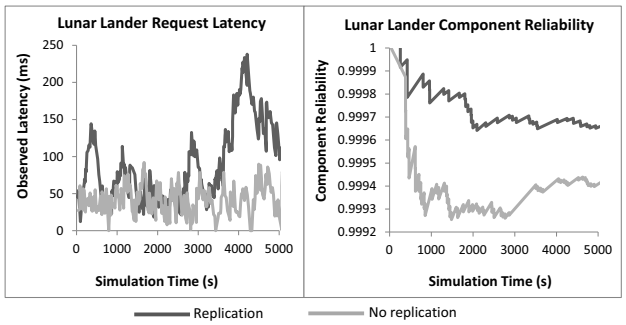


Figure 7. Latency and error rates computed in a Lunar Lander simulation.

Metamodel	Description
AADL	Modeling language targeted for specification and analysis of real-time embedded systems.
xADL 2.0 Core	A common set of fundamental modeling abstractions for software architectures.
Ecore	Eclipse metamodel for object-oriented systems.
C2	Component- and message-based architectural style for flexible, extensible software systems.
Client/server	Ubiquitous request/response architectural style.
Pipe-and-filter	Concurrent data-stream architectural style.
Publish-subscribe	Asynchronous and anonymous message distribution architectural style.
Myx	Layered architectural style for flexible construction of distributed systems.
Prism	Event-based style for embedded, mobile applications.

Figure 8. Nine metamodels used in the evaluation.

both LL designs, modeled in Myx. Each model consists of approximately 20 software components. Each component includes several lines of code (using the "backdoor" mechanism described in Section IV-A) to map input messages to output messages. We used XDEVS's native facilities to instrument the simulation with probes and quantify the two properties of interest. Figure 7 summarizes the end-to-end latency and failure rate data from the simulations. The two graphs quantify the intuition stated above. The graphs allow an engineer to make an informed architectural decision and provide quantitative rationale regarding whether the performance cost of replication is acceptable, or whether the system reliability requirements can be met without replication.

V. Evaluation

To evaluate the utility of LIGHT, we created metamodels for nine software architecture DSLs. We simulated domain-specific system models conforming to these metamodels using LIGHT's XDEVS simulation generator components and the XDEVS discrete event simulation engine.[1] Note that the ability to generate simulations from architectural models is not novel; rather, the novelty comes from the ability to do so without writing any custom code by hand. Therefore, we evaluated LIGHT according to programming effort it saved. We did not evaluate the accuracy of analysis results produced, as this is a product of the targeted analysis tool (XDEVS in this case), and is not related to the LIGHT approach itself. In this section, we define our evaluation metrics and present the results of applying LIGHT to the nine DSLs.[2]

A. Metrics

We used two sets of evaluation metrics: *Implementation effort* metrics measure the effort saved through code generation and reuse. *Maintenance effort* metrics measure the relative ease of performing DSL modifications in a metamodel, as afforded by LIGHT, rather than in interpreter source code, as is necessary in other MDE platforms.

[1]We also generated executable Prism-MW code. The results were comparable to those we show for simulation and we omit them for space.

[2]The LIGHT source code and all metamodels used in the evaluation can be downloaded from http://softarch.usc.edu/~gedwards/LIGHT/LIGHT.zip.

The implementation effort metrics are:

- *Number of domain-specific types* in the metamodel, which is a measure of metamodel size and complexity.
- *Lines of generated interpreter code* by the metainterpreter in the form of MIF extensions.
- *Total lines of reused interpreter code*, which is the sum of (1) the 16,243 SLOC in the XDEVS MIF (built into LIGHT for reuse) and (2) the generated SLOC in the MIF extension for each metamodel.
- *Lines of generated code per domain-specific type.*
- *Lines of reused code per domain-specific type.*

The maintenance effort metrics were computed by (1) performing a set of modifications to a DSL, (2) regenerating the XDEVS MIF extension for the DSL, and (3) diffing the previous and new MIF extensions to determine the impact of the DSL changes. The maintenance effort metrics are:

- *The number of metamodel objects altered.*
- *The number of interpreter classes altered.*
- *The number of interpreter methods altered.*
- *Lines of code altered per domain-specific type.*
- *Total lines of code altered.*

The first metric provides an indication of the level of effort needed to achieve a modification using the LIGHT approach; the remaining metrics indicate the level of effort that would be required using a conventional approach.

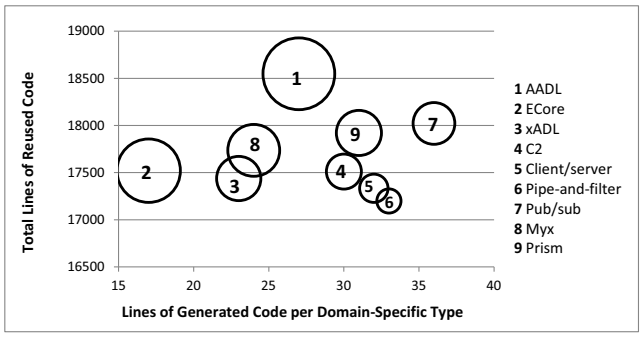
The implementation and maintenance effort metrics are language-specific. We suggest below a mechanism for converting these metrics into more language-independent programmer person-months.

B. Results

Figure 8 summarizes our nine subject metamodels. For each metamodel, Figure 9a displays the implementation effort metrics for the XDEVS simulation generator components. Note that the model editor components and code generated for a specific application model, such as LL, would also be automatically generated using previous approaches, so these metrics are not included. All the code included in the counts in Figure 9a would need to be written by hand using a previous approach. Figure 9a shows that the DSLs range in the number

metamodel	domain-specific types	sloc generated	total sloc reuse	generated sloc per type	total reuse per type
AADL	86	2,306	18,549	27	216
Ecore	76	1,277	17,520	17	231
xADL Core	53	1,195	17,438	23	329
C2	42	1,268	17,511	30	417
Client/server	34	1,091	17,334	32	510
Pipe-and-filter	29	957	17,200	33	593
Pub-sub	50	1,779	18,022	36	360
Myx	62	1,493	17,736	24	286
Prism	54	1,678	17,921	31	332

(a)



(b)

Figure 9. The collected implementation effort metrics (a) are summarized based on metamodel size, total reused code, and generated code per type (b).

of domain-specific types defined (a rough indicator of the metamodeling effort required). This is to be expected as the nine DSLs vary widely in their scope and features. Using a conventional approach, the number of domain-specific types would be the same, but each domain-specific type would be somewhat easier to define because there are fewer metatype property values to set. This is the single added cost of using our approach as compared to existing techniques; however, this cost is eclipsed by the resulting savings, as discussed below.

Figure 9b illustrates the relationship between metamodel size, generated code per type, and total code reuse. The diameter of each circle represents the size of the corresponding metamodel. The larger metamodels generally result in greater overall reuse, but the reuse benefits are amortized over a larger metamodeling effort, resulting in a smaller benefit per domain-specific type. This reinforces the intuition that, for very large metamodels, more implementation effort is avoided, but metamodeling consumes a larger share of the overall effort.

To estimate the savings incurred by using LIGHT, we applied the widely-used COCOMO II model [1]. As indicated in Figure 9a, the average amount of code reused in the implementation of domain-specific simulation generators for the nine DSLs is ∼17,500 SLOC. Let us assume that this amount of code had to be written manually.[3] COCOMO II's estimates of the required effort for a project of this size range between 4.2 and 23.4 person-months, depending on the project parameters.

Figure 10 shows the maintenance effort metrics for seven DSL modifications. To obtain a broad sample, we modified different metamodels in different ways, including introducing new inheritance and containment relationships, changing metamodel properties, and adding or removing types. The resulting changes in the generated code depended on the specific modifications. For example, inheritance modifications tended to affect a large number of classes but may not have changed any methods within those classes (as in the case of client/server). In contrast, containment modifications affected a small number of classes but resulted in more new code (as in the case of AADL).

While the number of lines of code altered in response to the changes was, in some cases, quite small, these changes tended to be spread over a relatively large number of classes and methods in comparison to the relatively small number of metamodel objects that were altered. This implies that modifications required for DSL evolution are more widely scattered, and therefore more time-consuming to implement, in interpreter source code than in a metamodel. Moreover, a metamodel may be more understandable and easier for a new engineer to modify, which is important when the DSL must be maintained over a period of time.

VI. Related Work

Our work builds upon prior research in architecture modeling, product line architectures, model-driven engineering, model-driven architecture, system analysis and simulation, and application frameworks. Here, we specifically highlight three notable DSL-based approaches that attempt to automate generation of model interpreters. We also discuss the relationship of LIGHT to our own previous work.

Cadena [12] is a model-driven toolchain for component-based systems. Bogor is an extensible model checker that can be applied to Cadena models. Similarly to LIGHT, Bogor allows engineers to reuse the analysis infrastructure while customizing selected constructs and behaviors. However, some of the mechanisms that enable extension and reuse in Bogor are specific to model checking, making them difficult to generalize and apply to other types of analysis or code generation.

DUALLY [16] supports interoperability among architecture DSLs. DUALLY leverages engineer-defined mappings between DSL metamodels and a core set of architectural concepts codified in a metamodel, called A_0. Thus, the analysis and code generation tools available for any DSL with a defined mapping can be applied to architectural models specified in other DSLs. Although DUALLY eliminates the need to manually program model transformations, it still requires defining the mappings between languages. Also, introducing additional model transformations makes traceability more challenging.

ALFAMA [21] automates the construction of DSLs for application frameworks. ALFAMA leverages an aspect-oriented domain-specific modeling (DSM) layer that defines *specialization aspects* that modularize framework hot-spots (extension points) and associates those aspects with DSL concepts. The

[3]While we do not have access to manually implemented versions of these simulation generators, we expect that their size would be proportional to the size of the automatically generated versions.

metamodel	types affected	classes altered	methods altered	sloc altered per type	sloc altered
AADL	2	3	10	64.5	129
Client/server	4	10	1	5	20
C2	2	4	1	7	14
Ecore	1	4	4	19	19
Pub-sub	1	3	4	22	22
Myx	5	8	12	20.4	102
Prism	4	7	14	21	84

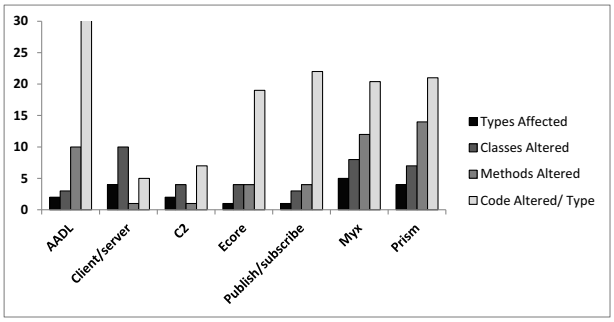


Figure 10. The collected maintenance effort metrics.

DSM layer allows engineers to bypass metamodel development. This has many merits but also some drawbacks. First, high-level metamodels are more maintainable than low-level source code. Second, inferring DSLs from framework hotspots tightly couples the DSL to a particular framework. As with LIGHT, the automation gains of the ALFAMA approach reduce DSL flexibility in some circumstances.

We previously studied factoring out and modularizing domain-independent and domain-specific model interpretation logic, an approach we implemented in the XTEAM platform [3]–[5]. XTEAM promoted reuse of significant portions of model interpreter implementations, but still required engineers to manually program certain domain-specific logic. In contrast, LIGHT leverages the isomorphism and generates model interpreters using the same method that has already proven so successful for the construction of model editors.

VII. Contributions

Automatically synthesizing modeling tools reduces the cost of using DSLs. This makes the benefits of domain-specific modeling, traditionally enjoyed by large-scale development projects, accessible to small- and medium-sized projects. We implemented our approach in LIGHT and evaluated LIGHT by generating three model interpreters for each of nine different DSLs to demonstrate that it significantly reduces the required programming effort and system maintenance.

LIGHT does not come without limitations. One seeming limitation is a slightly increased metamodeling burden over existing MDE approaches. However, this is more than compensated for by the implementation savings. Another, real limitation stems from the trade-off between the expressiveness and flexibility of the supported DSLs on the one hand, and the ability to automatically synthesize tools on the other. Finally, we have not evaluated the impact of changes to the semantics of the LIGHT metatypes, but such a change would clearly require significant effort by the MDE platform developers.

References

[1] B. Boehm *et al.*, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[2] E. Dashofy, "Supporting stakeholder-driven, multi-view software architecture modeling," Ph.D. dissertation, UCI, 2007.

[3] G. Edwards *et al.*, "Construction of analytic frameworks for component-based architectures," in *SBCARS*, 2007.

[4] ——, "Scenario-driven dynamic analysis of distributed architectures," in *FASE*, 2007.

[5] G. Edwards and N. Medvidovic, "A methodology and framework for creating domain-specific development infrastructures," in *ASE*, 2008.

[6] G. Edwards, Y. Brun, and N. Medvidovic, "Isomorphism in model tools and editors," in *ASE*, 2011.

[7] G. Edwards *et al.*, "A highly extensible simulation framework for domain-specific architectures," USC, Tech. Rep. USC-CSSE-2009-511, 2009.

[8] G. Edwards, "Automated synthesis of domain-specific model interpreters," Ph.D. dissertation, USC, 2010.

[9] P. H. Feiler *et al.*, "The architecture analysis & design language: An introduction," Software Engineering Institute, Tech. Rep. CMU/SEI-2006-TN-011, 2006.

[10] "The generic modeling environment," http://www.isis.vanderbilt.edu/Projects/gme/.

[11] "The Eclipse graphical modeling framework," http://www.eclipse.org/modeling/gmf/.

[12] G. Jung *et al.*, "A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures," in *GPCE*, 2007, p. 42.

[13] G. Karsai *et al.*, "On the use of graph transformation in the formal specification of model interpreters," *Journal of Universal Computer Science*, vol. 9, no. 11, pp. 1296–1321, 2003.

[14] G. Karsai, "Structured specification of model interpreters," in *ECBS*, 1999, pp. 84–90.

[15] "LIGHT metatype quick reference guide," http://softarch.usc.edu/~gedwards/xteam2/MetatypeTable.pdf.

[16] I. Malavolta *et al.*, "Providing architectural languages and tools interoperability through model transformation technologies," *IEEE TSE*, vol. 36, no. 1, pp. 119–140, 2010.

[17] S. Malek *et al.*, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE TSE*, vol. 31, no. 3, pp. 256–272, 2005.

[18] P. Mohagheghi and V. Dehlen, "Where is the proof? — A review of experiences from applying MDE in industry," in *MDA Foundations & Applications*, 2008.

[19] J. J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley, 2010.

[20] R. Roshandel *et al.*, "Estimating software component reliability by leveraging architectural models," in *ICSE*, 2006.

[21] A. L. Santos *et al.*, "Automating the construction of domain-specific modeling languages for object-oriented frameworks," *JSS*, 2010.

[22] C. Seo, "Prediction of energy consumption behavior in component-based distributed systems," Ph.D. dissertation, USC, 2008.

[23] R. N. Taylor *et al.*, *Software Architecture: Foundations, Theory and Practice*. Wiley Publishing, 2009.

[24] M. Woodside, "Tutorial introduction to layered modeling of software performance," Carleton University, Tech. Rep., 2002.