Using Parameterized Attributes to Improve Testing Capabilities with Domain-specific Modeling Languages

James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

***Abstract*—Domain-specific modeling languages (DSMLs) show promise in improving model-based testing and experimentation (T&E) capabilities for software systems. This is because its intuitive graphical languages reduce complexities associated with error-prone, tedious, and time-consuming tasks. Despite the benefits of using DSMLs to facilitate model-based T&E, it is hard for testers to capture many variations of similar tests without *manually* duplicating modeling effort.**

This paper therefore presents a method called *parameterized attributes* that is used to capture points-of-variation in models. It also shows how parameterized attributes is realized in an open-source tool named the Generic Modeling Environment (GME) Template Engine. Finally, this paper quantitatively evaluates applying parameterized attributes to T&E of a representative distributed software system. Experience and results so show that parameterized attributes can reduce modeling effort after an initial model (or design) is constructed.

I. Introduction

Domain-specific modeling languages (DSMLs) [1] enable modelers to construct models using abstractions that are intuitive to the target domain. In addition, constraints coupled with the abstractions ensure constructed models do not violate the domain's semantics. Model interpreters then parse constructed models and generate concrete artifacts that are—at many times—error prone, tedious, and time-consuming to manually handcraft (*e.g.*, configuration files and source code). Examples of environments that facilitate creation of DSMLs include: Generic Modeling Environment (GME) [1], Microsoft DSL Tools [2], Eclipse Modeling Framework [3], and Generic Eclipse Modeling System (GEMS) [4].

Many domains have benefited from using DSMLs [5]–[9]. This can be attributed to DSMLs increasing the level-of-abstraction for the target domain. For example, model-based testing and experimentation (T&E) of distributed real-time and embedded (DRE) systems has seen benefits from using DSMLs [10]. This is because DRE systems possess characteristics that complicate their software lifecycle, such as design and composition, deployment and configuration, and heterogeneity and scale. DSMLs therefore have improved T&E capabilities by enabling DRE system testers (*i.e.*, the modelers) to evaluate system design choices on the target architecture during early phases of the software lifecycle instead of waiting until complete system integration time [10].

Although DSMLs improve T&E capabilities, testers rely heavily on *single instance* models where one model represents one test configuration (or experiment) [11]. If variations of the test is needed (*i.e*, a different configuration), then testers either (1) manually update the existing model to capture the new configuration, such as changing the value of an attribute, or (2) manually duplicate an existing model and capture the new configuration. This approach, however, is not cost-effective because testers either lose their original model or have to *manually* manage replicas of similar models.

The use of parameters is a well-known technique for increasing variability. For example, parameters in a function enable software developers to increase the scope (or variability) of acceptable values the function can handle. Likewise, parameters in C++ templates enable software developers to increase the set of acceptable data types for a given function/class (*i.e.*, increase variability). Most importantly, parameters help increase variability, but not at the expense of duplicating effort. It is therefore believed that supporting parameters in DSMLs can also increase variability for model-based T&E without the expense of duplicating modeling effort. Parameters in DSMLs will also reduce the number of single instance configuration models where the model is similar, but the configuration is *slightly* different.

This paper therefore describes a technique for adding support for parameters to DSMLs. The main contributions of this paper are as follows:

- It presents a method called called *parameterized attributes* for DSMLs, which is the *first* attempt at adding parameter support to DSMLs;
- It discusses the design alternatives for supporting parameters in DSMLs, including advantages and disadvantages of each design alternative;
- It showcases how parameterized attributes have been realized in an open-source tool named the GME Template Engine; and
- It evaluates the benefits and consequences of using supporting parameterized attributes in DSMLs.

Experience from using parameterized attributes show that testers are able to validate different configurations within the original model. This is opposed to outside the model where the DSML's semantics do not exist, or are enforced in an *ad hoc* manner. Moreover, parameterized attributes can decrease modeling effort once the original model (or design) is crafted.

Paper organization. The remainder of this paper is organized as follows: Section II discusses shortcomings of current techniques in context of a case study; Section III presents the details of parameterized attributes for DSMLs; Section IV evaluates using parameterized attributes in DSMLs; Section V discusses related work; and Section VI presents concluding remarks.

II. Motivating Case Study: Model-based T&E of the SLICE Scenario

The SLICE scenario is a representative enterprise DRE system from the domain of shipboard computing. The SLICE scenario has been used in other case studies, such as validating system execution modeling tools [10], validating emulation techniques [12], and locating valid deployment and configurations that achieve desired quality-of-service (QoS) requirements [13], such as end-to-end response time.

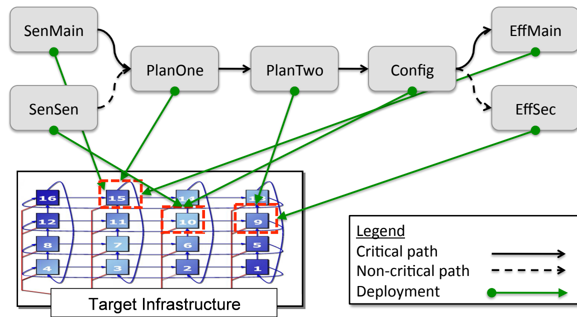

Fig. 1. High-level overview of the SLICE scenario [12].

To briefly reiterate, the SLICE scenario (shown in Figure 1) consists of 7 component instances (*i.e.*, the rectangular figures): `SensorMain`, `SensorSec`, `PlannerOne`, `PlannerTwo`, `ConfigOp`, `EffectorMain`, and `EffectorSec`. The line between each component instance represents a channel for inter-component communication, such as sending and event from one component to another using a messaging service. Finally, the SLICE scenario has a critical path of execution that flows from `SensorMain` to `EffectorMain` and must meet a specified deadline.

Validating the SLICE scenario's critical path of execution requires T&E in a realistic environment using many different deployments and configurations. To accomplish this feat, the SLICE testers used CUTS, which is a system execution modeling tool built atop a DSML (see Sidebar 1). Figure 2 illustrates a portion of a model for the SLICE scenario that was created using CUTS. As shown in this, a single value differentiates models (a)–(d). Depending on the number of configurations needed to evaluate the SLICE scenario's critical path of execution, SLICE testers must create duplicate models for each configuration—similar to Figure 2. Moreover, the effort required to duplicate such models increases in direct relation to the number of points-of-variability (*i.e.*, variable portions of the model where values can change to represent different configurations).

One common alternative to addressing this challenge is modifying artifacts after they have been generated from a model—if doing so is *even* possible. This approach, however, presents three (3) limitations:

1) **Limited reuse.** The approach is *ad hoc* and usually (re)invented by testers for each file type generated from a model. For example, testers typically create a new script to modify generated configuration files for each application domain.
2) **Limited persistence.** The changes occur outside of model and can easily be overwritten. For example, testers use a script to modify a generated configuration file. They then use to original model to (re)generate the original configuration, which *accidently* overwrites the modified configuration file.
3) **Limited validation support.** Changes outside of the model are not governed by the metamodel's semantics. This means invalid changes can go undetected without the proper domain knowledge. For example, a property in the configuration file has a range of [0, 10], but testers change the value to 15. This is an invalid value that can go undetected since the change occurred outside of the model.

Due to these limitations, it is hard for the SLICE testers to effectively use CUTS for validating different deployment and configurations. Moreover, this problem extends beyond the SLICE scenario and CUTS and applies to other projects and applications of DSMLs that must support some notion of variability at the model level (*i.e.*, generate similar models

Sidebar 1: Brief Overview of CUTS

CUTS [10] is a system execution modeling tool for conducting system integration test that validate (QoS), *e.g.*, end-to-end response time, latency, and scalability, properties on the target architecture during early phases of the software lifecycle. Testers use CUTS via the following steps:

1) Use DSMLs [1] to model behavior and workload at high-levels of abstraction;
2) Use generative programming techniques [14] to synthesize a complete test system from constructed models that conform to the target architecture; and
3) Use emulation techniques to execute the synthesized system on its target architecture and validate its QoS properties.

Testers can also replace emulated portions of the system with its real counterpart as its development is completed. Finally, CUTS supports validating QoS properties on several network communication architectures, such as: CIAO (www.dre.vanderbilt.edu/CIAO), OpenSplice (www.opensplice.org), RTI-DDS (www.rti.com), and TCP/IP.

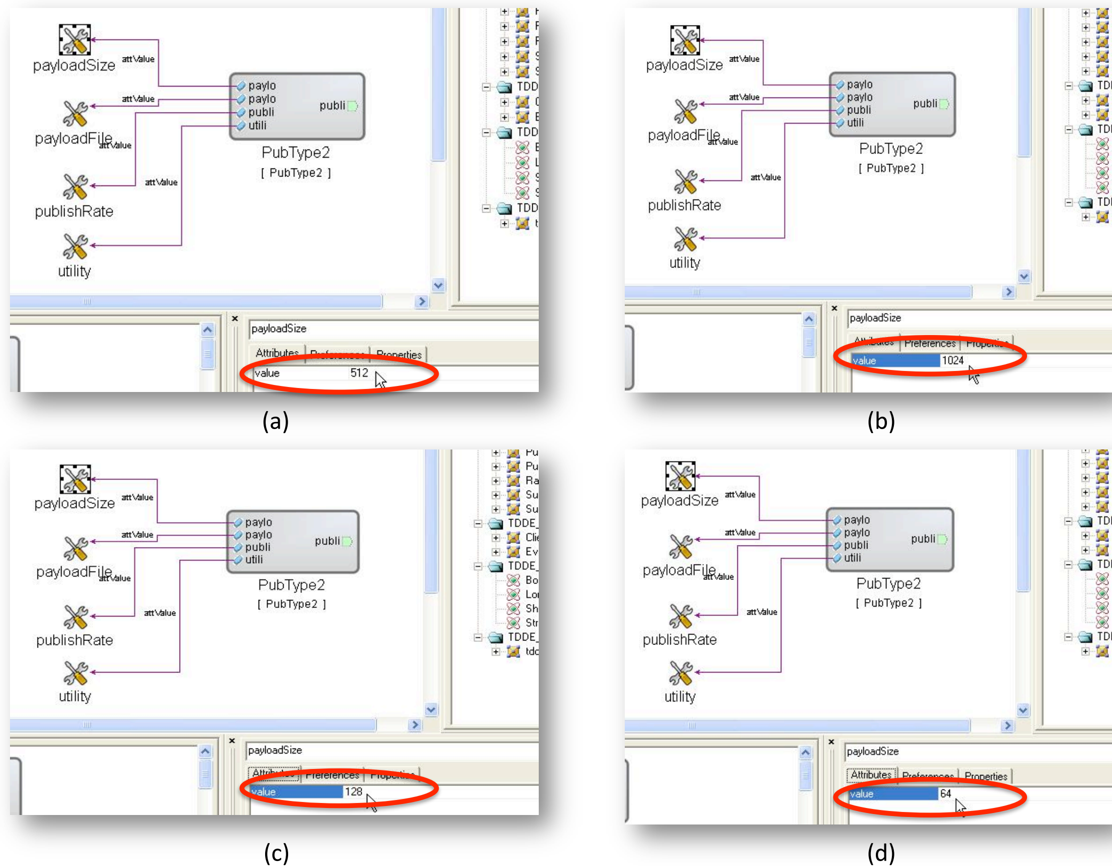

Fig. 2. Example of creating similar models for different configurations.

with different configurations without duplicating the modeling effort). This problem, however, has *extreme* applicability to model-based T&E via DSMLs because of the need to execute many similar tests. The remainder of this paper therefore discusses how parameterized attributes for DSMLs addresses these limitations.

III. Parameterized Attributes in DSMLs

This section discusses the details of parameterized attributes for DSMLs to address the limitations introduced in Section II. This section also illustrates how parameterized attributes are realized in an open-source tool for the GME named the *GME Template Engine*.

A. Evaluating Design Alternatives for Realizing Parameterized Attributes in DSMLs

There are different design alternatives for realizing parameterized attributes in DSML where each as both its advantages and disadvantages. In DSMLs, there are three different design alternatives:

- **Alternative 1: Realization at metamodel level.** The first level of abstraction is at the metamodel level. The advantage of realizing parameterized attributes at this level is physical elements used to construct a model can support direct parameterization by other physical model elements from the same model. The disadvantage of this approach is it can complicate the validation process for DSMLs because the semantics of the parameterized element must be learned at runtime. This can imply the semantics are not well-defined. Moreover, it can complicate the model interpretation process since model interpreters are tightly coupled to a DSML and also have well-defined semantics.
- **Alternative 2: Realization at model level.** The second level of abstraction is at the model level. The advantage of realizing parameterized attributes at this level is that the DSML's semantics remain well-defined. This is because parameters are used as placeholders for actual values and the model elements are static (or constant). The disadvantage of this approach is it requires external tool, such as local file or remote database, to capture parameter values for instantiating different single instance models.
- **Alternative 3: Realization at interpreter level.** The third level of abstraction is at the interpreter level using template-like languages. This approach has the same advantages as realizing parameterized attributes at the model level. The main disadvantage of this approach is it is *hard* to validate the interpreted models changes when it is instantiated with concrete values. This is because the parameters are not replaced directly in the model.

Instead, they are replaced outside of the model during the interpretation process. In order to validate the model under this approach, it is necessary for the interpreter to duplicate efforts already captured by the metamodel.

Because of the limitations introduced in Section II, Alternative 2 is selected as the design alternative. Alternative 1 was not selected because testers will not be able to use parameters to define different configurations. Instead, it should be used when defining different families of languages. Likewise, Alternative 3 makes it *hard* for testers to guarantee parameter values do not violate the metamodel's semantics.

Alternative 2 allows testers to define configurations in external sources that can be used to instantiate many different single instance models. Alternative 2 therefore is a more practical solution to addressing the 3 limitations introduced in Section II.

B. Defining Parameterized Attributes in DSMLs

Ideally, to reduce complexity associated with duplicate modeling effort to improve variability in model-based T&E when using DSMLs, testers need the ability to use *placeholders* that can represent the points-of-variability for different model configurations. The placeholders can then be replaced using user-defined values that correspond to different single instance models. These placeholders can be realized using template-like constructs that represent parameterized attributes—similar to how parameters are applied in other domains.

Based on this objective and the selected design alternative, a model $M = (A, N)$ is viewed as:

- a set A of attributes that capture scalar values of model element M; and
- a set N of child model elements that are contained in the current model.

Because N is a set of models, the following is true: $\forall n \in N | M_n = (A_n, N_n)$. Finally, the global set of all attributes Λ_M in model M, *i.e.*, the attributes of the current model and all its child models, is defined as:

$$\Lambda_M = \begin{cases} A_M & N_M = \emptyset \\ A_M \cup \Lambda_N & N_M \neq \emptyset \end{cases} \tag{1}$$

The global set of attributes Λ_M represents the points-of-variability that subject to parameterization using parameterized attributes at the model level. The set of parameterized attributes, however, is a subset of the global set of attributes (*i.e.*, $P \subset \Lambda_M$) because each $p \in P$ has a unique name. It is therefore possible for a parameterized attribute to appear more than once in a parameterized model.

Definition 3.1: A *parameterized model* is a model that contains one or more parameterized attributes.

Given a parameterized model, it must be instantiated before it can be used like a normal model. Instantiating a parameterized model requires substituting each parameterized attribute with a concrete value. These values are extracted from a external configuration located in persistent storage that stores name-value pairs where the name corresponds to a parameterized attribute, and the value is the actual value of that parameterized attribute.

C. Validating Parameterized Attributes and Configurations in DSMLs

By supporting parameterized attributes at the model level, the actual values of some attributes are not explicitly defined in the model. Instead, placeholders are used and the concrete values are extracted an external source. Since concrete values are not present in the parameterized model, it is not possible to validate parameterized models in their current state. Instead, each parameterized model must be instantiated using a configuration before it can undergo validation.

In some cases, different configurations may or may not be valid. To help determine if a configuration is valid, the following definitions are used when instantiating a parameterized model:

Definition 3.2: A configuration is **complete** iff for all parameterized attributes in the model, there exists a name-value pair in configuration where the name of the name-value pair equals the name of a parameterized attribute.

Definition 3.3: A configuration is **valid** iff the configuration is complete and the instantiated model is valid (*i.e.*, passes all its constraints).

D. Realizing Parameterized Attributes using GME Template Engine

The approach presented in Section III-B and Section III-C has been realized in a tool called *GME Template Engine*. GME Template Engine is implemented as a GME plugin, *i.e.*, a DSML-independent model interpreter. Modelers use GME Template Engine via the following steps (shown in Figure 3):

1) Augment an existing model with template-like parameters that represent portions of the model that can vary between different configurations (*i.e.*, the points-of-variability);
2) Create a text-based file that contains multiple configurations where each configuration has a value for each parameter in the target model; and
3) Execute GME Template Engine plugin by selecting the configuration file created in Step 2 and the model interpreter to execute after instantiating each single instance configuration.

After completing the steps above, GME Template Engine parses a text-based configuration file. For each *complete* configuration, CUTS Template Engine uses it to (1) instantiate the parameterized model, (2) execute the constraint checker on the instantiated model, and (3) interpret the instantiated model is valid.

Figure 4 illustrates an augmented a model (Step 1) using the SLICE scenario. As illustrated in this figure, parameterized attributes are strings enclosed by <% %> markers. Likewise, Listing 1 illustrates an example configuration for the SLICE scenario that corresponds to Figure 4.

As highlighted in Listing 1, each configuration has an unique name (defined by the modeler) and is a simple

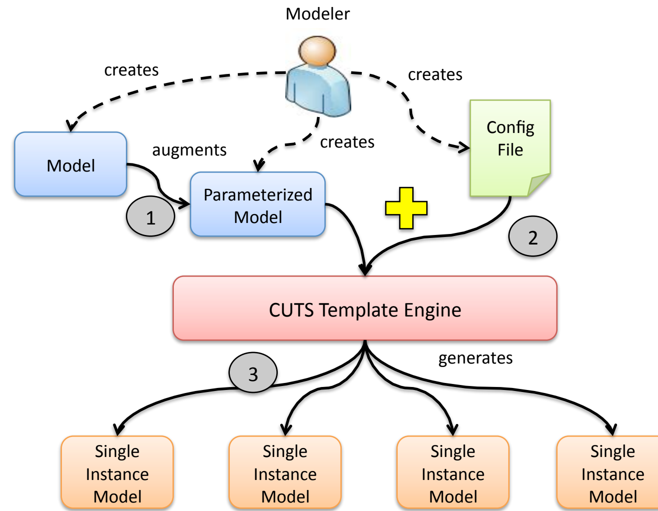

Fig. 3. Workflow for using GME Template Engine in GME on parameterized models.

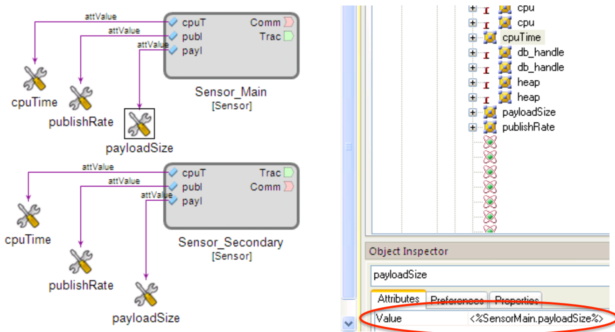

Fig. 4. Example of augmenting an existing GME model with parameterized attributes.

structure-like specification. Each line in the configuration is a name-value pair for a parameterized attribute that appears in the parameterized model.

```
config (HighPayload) {
  SensorMain.payloadSize=1024
  SensorMain.publishRate=34
  SensorSec.payloadSize=256
  SensorSec.publishRate=12
}
```

Listing 1. Example configuration file for GME Template Engine.

For example, the `SensorMain.payloadSize=1024` on line 2 in Listing 1 corresponds to the `<%SensorMain.payloadSize%>` parameterized attribute in Figure 4. GME Template Engine also supports built-in parameters, which do not appear in a user-defined configuration. Table I highlights several of the built-in parameters currently supported by CUTS Template Engine.

In addition to built-in parameters, GME Template Engine supports parameter values that are defined in terms of other parameter values. For example, GME Template Engine is able to parse the configuration value:

```
unique.name=<%config.name%>.Deployment
```

TABLE I
BUILT-IN PARAMETERS SUPPORTED BY GME TEMPLATE ENGINE.

Name	Description
config.name	Name of the configuration
config.size	Number of parameters in the configuration
self.modelPath	Full path of element in the model

for the `<%unique.name%>` parameterized attribute.

```
config (BaseConfig) {
  SensorSec.payloadSize=256
  SensorSec.publishRate=12
}

config (HighPayload) : BaseConfig {
  SensorMain.payloadSize=1024
  SensorMain.publishRate=34
}

config (LowPayload) : BaseConfig {
  SensorMain.payloadSize=128
  SensorMain.publishRate=15
}
```

Listing 2. Example configuration inheritance for GME Template Engine

Finally, GME Template Engine supports configuration inheritance, similar to conventional object-oriented programming languages such as C++ and Java. Listing 2 illustrates configuration inheritance in GME Template Engine. In configuration inheritance, the sub-configuration (*i.e.*, `HighPayload` or `LowPayload`) inherits the configuration values of its base configuration (*e.g.*, `BaseConfig` in Listing 2). In the case of duplicate parameter values in different base configurations, precedence of base configuration values increases from left to right. Moreover, sub-configuration values override base configuration values. By supporting these features, CUTS Template Engine provides testers with more expressive capabilities to define configurations for parameterized models.

IV. Applying Parameterized Attributes to the SLICE Scenario

This section evaluates applying parameterized attributes and GME Template Engine to the SLICE scenario.

A. Evaluating the SLICE Scenario

As explained in Section II, the SLICE scenario is a representative enterprise DRE system from the domain of shipboard computing environments. An important aspect of the SLICE scenario is validating its critical path of execution. This validation task requires T&E of the SLICE scenario in a realistic environment using many different configurations.

Since the SLICE testers are using model-based T&E realized by DSMLs, and more specifically CUTS, they must create many different—yet similar—models to conduct the necessary tests. To avoid this requirement, the SLICE testers use parameterized attributes and GME Template Engine. Moreover, their ultimate goal is using parameterized attributes to create parameterized models that capture the core intellect of similar tests, such as the system's structure, and place configuration values for similar tests in an external configuration file.

To leverage parameterized attributes for their model-based T&E purposes, SLICE testers perform the following steps, which are similar to the steps presented in Section III-D:

1) **Create initial model.** The SLICE testers first used CUTS to create an initial model of the SLICE scenario. The initial model was considered a single instance model, but it captured all the core intellect of the scenario, *e.g.*, system structure, component attributes, and system deployments.
2) **Augment initial model.** After the SLICE testers created their initial model, they manually identified points-of-variability in the model, *i.e.*, attributes that are modifiable. Once SLICE testers identified the points-of-variability, they converted them to parameterized attributes that were identified using `<% %>` template-like parameters. For each component, there were three parameterized attributes: `publishRate`, `serviceTime`, and `payloadSize`. This gave the SLICE testers 21 parameterized attributes within the SLICE scenario model.
3) **Define configurations.** The final task was defining different configurations where each configuration represented a different test. Each configuration was defined as a set of name-value pairs where the name of the each name-value pair matched the name of a parameterized attribute in the augmented SLICE scenario model. Each configuration was stored in text file on disk.

After completing the three steps above, SLICE testers used GME Template Engine to generate the different experiments for the SLICE scenario. In total, the SLICE testers used scripts to auto-generate 500 different configurations (*i.e.*, name-value pairs) for the different parameterized attributes in the parameterized model. This gave the SLICE testers 500 different experiments of the SLICE scenario, which was created from a single parameterized model, to execute and evaluate in a realistic environment.

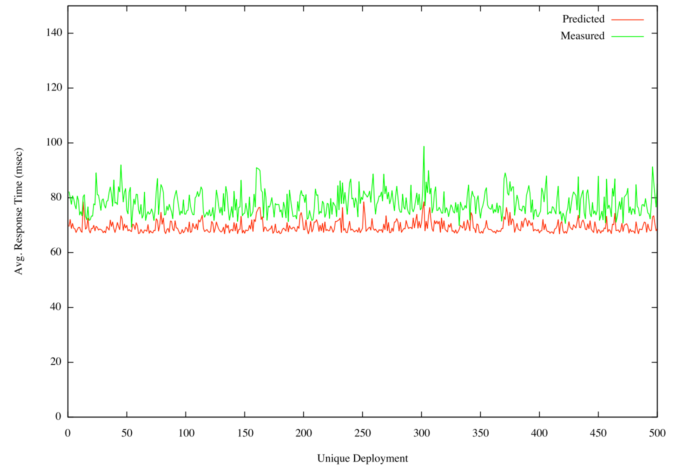

Fig. 5. Results for executing 500 different experiments of the SLICE scenario.

All experiments of the SLICE scenario were conducted in ISISlab (www.isislab.vanderbilt.edu), which is an integration testbed powered by Emulab [15] software. Figure 5 illustrates measuring the critical path of execution for the 500 different experiments (or deployments) discussed above. Although Figure 5 illustrates many test results for the SLICE scenario, the importance of Figure 5 is that the SLICE testers used 1 model (*i.e.*, a parameterized model) to generate 500 different experiments. Moreover, because the parameterized model was instantiated at the model level, SLICE testers were able to validate each configuration before generating artifacts from the instantiated model.

B. Consequences of the Parameterized Attribute Approach

Reduction in modeling effort. In parameterized models, each configuration, which corresponds to a single instance model, is represented as a set of name-value pairs in an external source, such as a local file or remote database. The number of name-value pairs that appear in configuration is *always* less than or equal to the number of parameters in a given model, which is *always* be less than or equal to the number of attributes defined in a given model. If each attribute in the target model was a point-of-variability, then it would have a corresponding name-value pair in the configuration, which is typically not the case. This therefore implies that the modeling time (or effort) to create different single instance models is less using parameterized attributes.

Table II highlights the steps SLICE testers (and modelers) must take when using conventional techniques versus using parameterized attributes to create different single instance configurations. As shown in this table, the number of steps for realizing many different single instance models using parameterized attributes is less than the conventional approach. Although Table II shows there is one less step, the step not present in parameterized models (*i.e.*, Step 5) is the one that

TABLE II
STEPS REQUIRED TO REALIZE SINGLE INSTANCE CONFIGURATIONS USING CONVENTIONAL TECHNIQUES VS. PARAMETERIZED ATTRIBUTES.

Step	Conventional DSML	Parameterized Attributes
1	Create model	Create model
2	Set model attributes	Set (non-)parameterized attributes
3	Run model interpreter	Define configuration(s)
4	Duplicate model	Run model interpreter
5	Manually repeat steps 2–5 until done	

depends on the number of single instance configurations. In the case of one single instance configuration, the number steps in conventional DSMLs and parameterized models is equal. Whenever there is more than one single instance model, parameterized attributes can drastically decrease required number of manual steps.

Initial cost associated with of parameterized models. Although parameterized models can decrease manual modeling effort, there is an initial cost associated with creating a parameterized model. As discussed in Section IV-A, SLICE testers derived the parameterized model from a single instance configuration model. This is the typical approach because, many times, what can be parameterized is unknown until the initial model is created and such a need arises.

$$C(M') = C(M_0) + C(P) \tag{2}$$

Based on this usage, Equation 2 shows the cost of creating a parameterized model $C(M')$ is equal to cost of creating the initial model $C(M_0)$ plus cost of defining the parameterized attributes $C(P)$.

Likewise, given $M'(P_i)$ is the instantiated model using parameters from configuration i, the cost of creating an instantiated model is equal to the cost of creating its parameters, as shown in Equation 3.

$$C(M'(P_i)) = C(P_i), i > 0 \tag{3}$$

The cost of defining the parameters for each instantiated model is also less or equal to the cost of creating actual model, as shown in Equation 4.

$$C(P_i) \leq C(M_i), i > 0 \tag{4}$$

Using Equation 3–4, Equation 5 shows cost of using parameterized models is less than or equal to cost of using traditional approaches.

$$C(M'(P_i)) \leq C(M_i), i > 0 \tag{5}$$

Whenever the cost of defining the actual parameters in a configuration is more than the cost of creating the actual model, parameterized models does not reduce cost. This can occur when dealing with small or simple models.

Generic attributes in metamodels. Parameterized attributes are a string-based placeholder for the concrete value. This implies that the target attribute must be of string type because most DSML environments use the attribute's type to enforce constraints on the GUI's input for the corresponding attribute. For example, if an attribute is of a integer type, then the DSML environment will (or should) enforce that only integers are input into the GUI for the corresponding attribute.

This requirement, however, can pose the following problems:

1) As stated before, an attribute that can be parameterized must be of string type. Even if only one instance of that attribute in the entire model is parameterized, the attribute's type in the metamodel must be generalized (*i.e.*, a string type) to support the need;
2) The DSML environment can no longer derive GUI constraints from the attribute's type. Likewise, the modeler can no longer rely on the GUI to enforce the attribute's type; ands
3) *All* constraints associated with the parameterized attribute type must be update to cast the string attribute value to the expected type.

These are considered *hidden cost* that both the DSML developer and the modeler must deal with, and are *hard* to quantify.

V. Related Work

This section compares work on parameterized attributes and CUTS Template Engine with other related work.

Aspect-oriented modeling techniques. C-SAW [16] is a MDE tool for supporting aspect-oriented modeling techniques, which are similar to aspect-oriented programming techniques [17], in DSMLs. C-SAW enables modelers to define cross-cutting concerns in models, such as model elements that determine the thread synchronization model for accessing variables or model elements that determine what values in a system should be logged at runtime, that are weaved into existing models. Parameterized attributes and CUTS Template Engine differ from C-SAW in that it focuses more on identifying points-of-variability in a model that can be replaced and validated using different external configurations; whereas, C-SAW focuses on extending existing models with new model elements based on cross-cutting concerns extracted from existing models. It, however, is believed that the CUTS Template Engine and C-SAW compliment each other in that they are two different (and non-competing) approaches for improving variability.

Feature-oriented modeling techniques. The Generic Eclipse Modeling System (GEMS) [18] is a graphical environment that supports the creation of DSMLs. One key feature of GEMS is it directly supports feature-modeling techniques [19], which is similar to feature-oriented programming [20]. Modelers use tools like GEMS and feature-modeling to define "features" that configurations should contain, which usually

is defined at the metamodel level. These features can then be used to construct configurations that meet their needs. Parameterized attributes and the CUTS Template Engine differ from GEMS and feature-modeling techniques in that it focuses on identifying points-of-variability that are defined statically in configurations; whereas GEMS derives such configurations based on the state of the existing model. Likewise, parameterized attributes functions at the model level instead of the metamodel level. Similar to C-SAW, however, the CUTS Template Engine and GEMS feature-modeling techniques can compliment each other in that they two different approaches for improving variability that can work together.

Programmatic techniques. Template libraries and frameworks, such as Google Templates [21], CodeSmith [22], and openArchitectureWare [23], can be used programmatically construct template-like models similar to parameterized models and the CUTS Template Engine. The main difference between the template library/framework approach and parameterized attributes is with the former instantiation of single instance configurations happens outside of the model. It is therefore hard to validate each configuration. With parameterized attributes, the instantiation of single instance configurations occurs within the model. By using parameterized attributes, it is therefore possible to leverage the DSML's semantics to validate different configurations.

Other domains that leverage parameterization. Parameterization is not a new concept and have been used in domains other than DSMLs. For example, parameterization has been leveraged in Information Systems to create business rules engines [24]. This is where concrete rules for a company (or system) are abstracted away from the systems workflow to improve system reuse, configurability, and variability. Likewise, parameterization has been leveraged in creating unit tests [25]. Instead of creating many different, yet similar, unit tests, an abstract unit test that captures commonality is created. Then testers use the parameterized unit tests to test concrete classes that are applicable to the generalized test. In either case, the use of parameterization reduces the amount of duplicated effort. The same benefits are also seen when using parameterized attributes for DSMLs.

VI. Concluding Remarks

This paper presented a method called *parameterized attributes* for DSMLs, which has been realized in an open-source tool named the *GME Template Engine*. Experience applying parameterized attributes to DSMLs a representative example resulted in the following lessons learned and future research directions:

- **Parameterized attributes in DSMLs enables validation of different configurations at design-time.** This is because the configurations are validated using the DSML's constraints. Further enhancements should include real-time feedback and auto-selection of parameter values when defining a configuration to help reduce occurrence of invalid configurations.
- **Dynamic configurations are needed to further improve model variability.** Currently, parameterized models consist of parameterized attributes that are simple and scalar values. To further improve T&E capabilities via DSMLs, testers need the ability to auto-generate elements in a parameterized model. Future work therefore will investigate techniques for support dynamic parameterized models to continue improving T&E capabilities.
- **Parameterized attributes are a step towards modeling patterns (or templates) using DSMLs.** When modelers use parameterized attributes, the parameterized model captures core intellect that remains constant across multiple models. Properties that have lesser impact are captured in a configuration. Parameterized attributes therefore help testers (and modelers) capture concept patterns, which increases the level-of-abstraction for T&E and improves reuse.

GME Template Engine is freely available in open-source format for download at the following location: game.cs.iupui.edu.

References

[1] Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer **34**(11) (2001) 44–51

[2] Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley (2007)

[3] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley, Reading, MA (2003)

[4] White, J., Schmidt, D.C., Nechypurenko, A., Wuchner, E.: Introduction to the Generic Eclipse Modeling System. Eclipse Magazine **7** (2007)

[5] Gao, W., Neema, E., Gray, J., Picone, J., Por, S., Musunuri, S., Mathews, J.: Hybrid Powertrain Design Using a Domain- Specific Modeling Environment. In: IEEE Conference on Vehicle Power and Propulsion. (September 2005) 423–429

[6] Kelly, S.: Improving Developer Productivity With Domain-Specific Modeling Languages. www.developerdotstar.com/mag/articles/domain_modeling_language.html

[7] Balasubramanian, K.: Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems. PhD thesis, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville (September 2007)

[8] Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons (2008)

[9] Hermans, F., Pinzger, M., Deursen, A.: Domain-Specific Languages in Practice: A User Study on the Success Factors. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2009) 423–437

[10] Hill, J.H., Slaby, J., Baker, S., Schmidt, D.C.: Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In: Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia (August 2006)

[11] Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press (2009)

[12] Hill, J.H.: An Architecture Independent Approach to Emulating Computation Intensive Workload for Early Integration Testing of Enterprise DRE Systems. In: Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications, Vilamoura, Algarve-Portugal (November 2009)

[13] Hill, J.H., Gokhale, A.: Towards Improving End-to-End Performance of Distributed Real-time and Embedded Systems using Baseline Profiles. Software Engineering Research, Management and Applications (SERA 08), Special Issue of Springer Journal of Studies in Computational Intelligence **150**(14) (August 2008) 43–57

[14] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, Massachusetts (2000)

[15] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proc. of the Fifth Symposium on Operating Systems Design and Implementation, Boston, MA, USENIX Association (December 2002) 255–270

[16] Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., Natarajan, B.: An Approach for Supporting Aspect-Oriented Domain Modeling. In: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03). (2003)

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). (June 1997) 220–242

[18] White, J., Schmidt, D.C.: Simplifying the Development of Product-line Customization Tools via Model Driven Development. In: Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines, Half Moon Bay, Jamaica (October 2005)

[19] White, J., Nechypurenko, A., Wuchner, E., Schmidt, D.C.: Automating Product-Line Variant Selection for Mobile Devices. In: Proceedings of the 11th Annual Software Product Line Conference (SPLC), Kyoto, Japan (September 2007)

[20] Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In Aksit, M., Matsuoka, S., eds.: ECOOP'97—Object-Oriented Programming, 11th European Conference. Volume 1241., Jyväskylä, Finland, Springer (9–13 1997) 419–443

[21] google-ctemplate: google-ctemplate. code.google.com/p/google-ctemplate (2007)

[22] Tools, C.: CodeSmith Tools. www.codesmithtools.com (2009)

[23] openArchitectureWare: openArchitectureWare. www.openarchitectureware.org (2007)

[24] Chisholm, M.: How to Build A Business Rules Engine: Extending Application Functionality Through Metadata Engineering. Morgan Kaufmann (2003)

[25] Tillmann, N., Schulte, W.: Parameterized Unit Tests. SIGSOFT Software Engineering Notes **30**(5) (2005) 253–262