

Measuring and Reducing Modeling Effort in Domain-specific Modeling Languages with Examples

James H. Hill
Indiana University-Purdue
University Indianapolis
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

Abstract—Domain-specific modeling languages (DSMLs) facilitate rapid and “correct-by-construction” realization of concepts for the target domain. Although DSMLs provide such benefits, there is implied (or hidden) modeling effort—in terms of user actions—associated with using a DSML that can negatively impact its effectiveness. It is therefore critical that DSML developers understand the meaning of modeling effort and how to reduce it so their DSML is of high quality.

This paper provides two contributions to research on developing DSMLs. First, the paper defines a metric for measuring model effort. Secondly, this paper discusses several techniques, with examples, reducing (or improving) modeling effort. The techniques discussed in the paper have been applied to an open-source DSML called the *Platform Independent Component Modeling Language (PICML)*, which is currently used in both academic and industry settings for designing and implementing large-scale distributed systems. Finally, results show that it is possible to reduce modeling effort without requiring user studies to analyze such concerns.

I. INTRODUCTION

Domain-specific modeling (DSM) [1], [2] is a model-driven engineering (MDE) [3] technique where custom graphical languages capture both the abstractions and semantics of a given domain. Domain-specific modeling languages (DSMLs) then provide modelers with intuitive abstractions, such as well-recognized images for each model element in the domain, that enables modelers to realize concepts governed by the DSML’s semantics (*i.e.*, the underlying metamodel). Examples of environments that facilitate the construction of DSMLs include: the Generic Modeling Environment (GME) [1], Microsoft DSL Tools [4], Eclipse Modeling Framework [5], and the Generic Eclipse Modeling System (GEMS) [6].

One of the benefits of DSMLs is rapid and “correct-by-construction” realization of concepts in the target domain. For example, a tedious and error-prone task when developing large-scale component-based distributed systems is crafting deployment descriptors [7]. These documents detail (a) the placement of a component, (b) the binary that contains a component, (c) the values of a component’s attributes (if any), and (d) the connections between component ports. Manually handcrafting this document is problematic because

(1) the document is usually XML-based and error-prone to write, and (2) it is hard to accurately cross-reference and ensure the semantics of key elements (such as connections between component ports). DSMLs therefore have had a major impact in this domain by providing intuitive abstractions for constructing models that represent concepts for this domain [8]. Moreover, model interpreters transform constructed models into concrete artifacts that were previously hard to write.

Although some benefits of DSMLs remove tedious and error-prone tasks, the complexity of a DSML, *i.e.*, its structure and functionality, can greatly impact its overall ability to realize concepts for its target domain. For example, one of the most common claims for DSMLs is that it raises the level-of-abstraction [3], [9], which helps realize concepts more rapidly. Likewise, its model interpreters can generate artifacts “faster” than a human can manually handcraft the same artifacts, such as complex XML-based deployment descriptors for large-scale component-based distributed systems. Although these are valid claims, if modelers have a hard time creating models, then it is hard for them to reap claimed benefits.

It is evident that there are two aspects to modeling effort that must be addressed when creating a DSML. The first aspect is the DSML’s ability to realize trivial and non-trivial concepts for the target domain. The second aspect is the DSML’s ability for modelers to rapidly create models that realize the concepts. The first aspect has been researched heavily [9]–[12]. Likewise, the second aspect has been researched [13]–[16], but existing research results are based on study groups and cognitive studies of graphical languages. Consequently, there exist no well-defined methodology for measure modeling effort—in terms of actions (or steps) the modeler must take to realize a concept—without the assistance of a study group. This makes it hard for the DSML developer to analyze modeling effort of a DSML before it is completely developed and ready to use.

This paper therefore provides contributions to research on improving DSML usability. More specifically, this paper’s main contribution is a methodology and metric for measuring modeling effort from the end-user’s point-of-view. This

paper also provides techniques, with examples, on how to reduce modeling effort, which can improve DSML quality. The techniques presented in this paper have been evaluated in an open-source DSML called the *Platform Independent Component Modeling Language (PICML)* [8], which is used in both industry and academic settings for designing and implementing component-based distributed systems. Results from applying the methodology to PICML show that it is possible for a DSML developer to measure modeling effort without requiring case studies with user-groups.

Paper organization. The remainder of this paper is organized as follows: Section II introduces PICML and presents examples where modeling effort is high; Section III discusses modeling effort in detail; Section IV presents examples for reducing modeling effort; Section V discusses lessons learned; Section VI compares this work with related work; and Section VII provides concluding remarks.

II. CASE STUDY: THE PLATFORM INDEPENDENT COMPONENT MODELING LANGUAGE (PICML)

The Platform Independent Component Modeling Language (PICML) is a DSML for designing and implementing component-based distributed systems. PICML is a large-scale DSML consisting of approximately 930 modeling elements and 130 constraints. It also contains 9 different model interpreters that transform models into concrete artifacts, such as source code, deployment descriptors, and interface definition language (IDL) files. Finally, PICML's model abstractions and constraints are based on the OMG's CORBA Component Model [17] and Deployment and Configuration [7] specification, however, it is not bound to the CORBA Component Model (*i.e.*, provides platform-independent abstractions and uses interpreters to realize platform-specific artifacts).

To-date, PICML has been used on many case studies from both academic and industry settings. For example, PICML has been used in research settings to assist with research and development of deployment and configuration tools [18], system execution modeling tools [19], and optimization techniques [8] for component-based distributed systems. Likewise, PICML has been used in industry settings to assist with developing large-scale component-based distributed systems from the domain of shipboard computing environments, avionic systems, and global communications systems.

Because PICML is based on the OMG's CORBA Component Model and Deployment and Configuration specification, it has many high-level goals. Such goals include modeling and ensuring the semantics of (1) component structure and functionality, (2) inter-component communication, (3) network structure (*e.g.*, hosts and LANs), and (4) component placement (*i.e.*, the mapping of components to physical hosts). Figure 1 shows an example PICML model of a component assembly where each rectangular box is

a component and the lines between components represent inter-component communication.

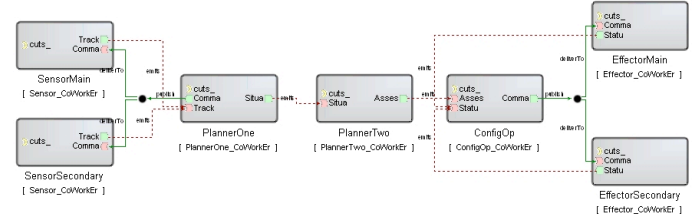


Figure 1. Example model of a component assembly in PICML.

Although PICML has many high-level goals, which are mapped to individual functionalities that facilitate designing and implementing component-based distributed systems, PICML has several functionalities that increase modeling effort. For example, it is hard for a modeler to know when a component is, and is not, deployed on a host. Likewise, it is hard for a modeler to model a complete component, *i.e.*, its structure and functionality, binaries, and implementation.

It can be argued that such complexities are the result of an improperly designed DSML. Although this is true in some cases, it is necessary to perform *domain analysis* [9], [20] when designing and implementing a DSML. This analysis determines what concepts should be captured by the DSML, and how to best represent them [13], [14]. Moreover, such analysis determines the DSML's granularity. Based on the domain analysis and desired granularity, the complexities mentioned above may, or may not, appear in a different DSML for modeling component-based distributed systems, or any DSML.

In either case, a DSML is governed by its metamodel. The metamodel of a DSML captures the abstractions and semantics of the target domain, and not modeling effort concerns. Accidentally and inherently, complexities associated with modeling effort therefore will be present in a DSML, such as PICML. It is therefore the responsibility of the DSML developer to design the DSML such that it does not possess high modeling effort. The remainder of this paper therefore discusses the notion of modeling effort in more detail and how to reduce modeling effort while using PICML to provide concrete examples.

III. DEFINING AND UNDERSTANDING MODELING EFFORT IN DSMLS

This section defines the concept of modeling effort in DSMLS. This section also discusses techniques for reducing modeling effort.

A. Defining Modeling Effort in DSMLS

In the context of this paper, *modeling effort* is defined as the number of steps it takes a modeler to complete a high-level modeling goal (or a task). For example, in PICML, an example high-level modeling goal is fully modeling a

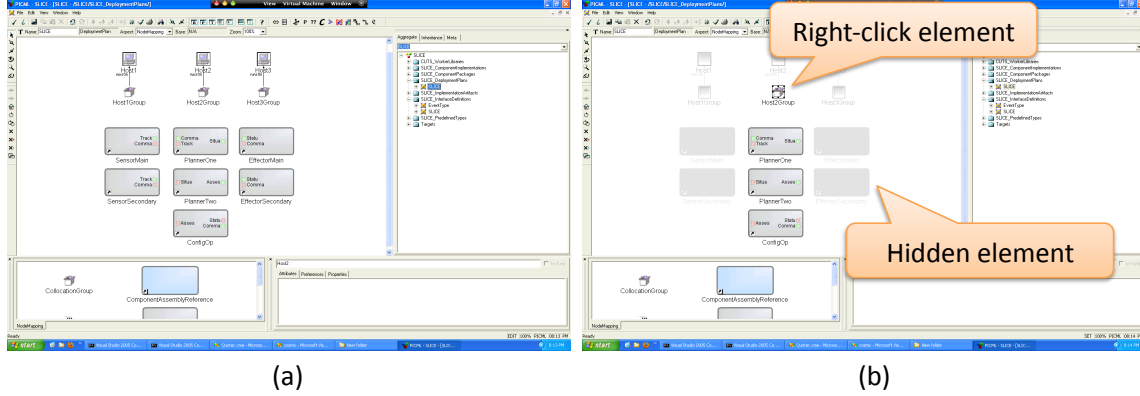


Figure 2. Example of a deployment model in PICML.

component’s implementation. This high-level goal consists of the following steps:

- 1) Modeling the component’s structure (*i.e.*, its interface and attributes);
- 2) Modeling the component’s binaries, which includes their names and locations; and
- 3) Associating the component’s implementation with the specified binaries.

Based on this definition of modeling effort, each task is comprised of many different modeling actions. Such modeling actions can include adding/removing an element, updating an existing element, and setting the attribute of a modeling element. Although this list of example modeling actions signify modification to a model, the set of modeling actions is not bound to this criteria. It can also include actions that do not modify the current model, such as visually checking the model’s state or clicking an element to learn its properties (or attribute values).

For example, Figure 2 shows a deployment model in PICML. In this example, not all of the components are deployed (or placed) on a host. At first glance, it is hard to determine what components are deployed, and which ones are not deployed as shown in Figure 2a. This is because the deployed and undeployed components visually look the same. In order to determine what components are not deployed, the modeler has to (1) right-click each node in the deployment¹ and (2) physically look at each component in the deployment model to determine if it is deployed or not deployed. Figure 2b illustrates how the components deployed on a host are highlighted once the modeler right-clicks one of the host model elements in the model.

In the example above, modeling effort is $n \times m$ where n is the number of components and m is the number of host elements in the model. This is because the modeler

¹PICML defines the `Host` element as a GME set model element. Another DSML design choice is to use a connection to show deployment, as done by White et al. [21]. Using a connection, however, can result in the model having too much clutter when dealing with large models.

must click (or select) each host in the deployment model and physically look at each component—whether or not it is of interest—contained in the deployment model.

If the DSML provided modeler’s with a graphical representation that showed a component’s deployment status, *e.g.* a different image when a component is deployed versus when it is not deployed, and did not require any other action outside of opening the modeling and looking, then modeling effort for this high-level goal would be n . Moreover, this would increase the DSML’s efficiency, which in-turn would improve modeler’s user experience.

B. Measuring Modeling Effort in DSMLs

As stated in Section III-A, modeling effort is defined as the number of actions required to complete a high-level modeling goal. The higher modeling effort is, the less efficient the DSML. The lower modeling effort is, the more efficient the DSML. To measure modeling effort, however, is it necessary to first understand the make-up of each action that is part of a given task.

In the software performance engineering [22], [23] domain, when calculating response time (or any measure) of a task that involves a human operator, such as a modeler, the task is divided into two disjoint parts: the computer and human part. The computer portion of the task is measurable, whereas the human portion is *hard* to measure. Because of this fact, the human portion of the task (or operation) is referred to as *think time* [22], [23] and represented as a variable Z that is not negligible.

Using such principles, it is possible to represent model effort in the same manner. This is because each task is completed by a human operator (*i.e.*, the modeler). With this in mind, the general equation for measuring modeling effort $M(T)$ is defined as:

$$M(T) = \sum_{t \in T} Z_t + \text{time}(t) \quad (1)$$

As illustrated in Equation 1, for each action t in task T , modeling effort is measured as the summation of the think

time for a given action Z_t and the time for the computer to complete its give portion of the task, *i.e.*, $time(t)$. $M(4)$ therefore is interpreted as, “the modeling effort of the task is 4 actions.” Likewise, $M(n)$ is interpreted as, “the modeling effort of the task is n actions.” Finally, constants cannot be reduced (or removed) from the analysis. This is because hidden think time is associated with each action. Therefore, $M(n+2) \neq M(n)$ because there are two actions with think time in the former that are not negligible.

To simply illustrate this definition, assume a model has a simple task of adding a new element to the model, which is $M(1)$. In order to complete this modeling goal, the modeler has to first think about what element needs to be added to the model². Once the modeler has decided what element to add, the modeler inserts the element. This is when the computer completes its portion of the action.

C. On Reducing Modeling Effort in DSMLs

Section III-B discussed how to measure modeling effort of a given task. As explained in that section, measuring modeling effort consists of summing the time for a computer to complete each task and the think time associated with each task. For example, if a high-level modeling goal had modeling effort of n , *i.e.*, consists of n actions, then its measure of modeling effort is defined by Equation 2:

$$M(n) = \sum_{i=1}^n Z_i + time(i) \quad (2)$$

where i is one of the n actions in the give high-level modeling goal.

To reduce modeling effort, first it is necessary to understand what portion of the measurement above is easy to reduce (or affect). When the two parts of the measurement are examined, it is *hard* to impact think time because it is variable. Moreover, the amount of think time associated with each action is dependent on the modelers experience—in most cases. Finally, to truly impact think time associated with a given action, it is necessary to do case studies and receive unbiased feedback on such concerns.

On the other hand, think time is associated with each action in a high-level modeling goal. Although it is hard, and almost impossible, to impact think time for a given action, if the action is completely removed from the task (*i.e.*, not completed manually by the modeler), then the associated think time is also removed from the equation. Because of this fact, reducing modeling effort can be achieved by removing actions completed by the modeler from a given high-level modeling goal. This in turn will reduce think time, but leave think time associated with the remaining actions.

For example, assume we reconsider the example of locating a component’s deployment status that was presented in

Section II. In PICML, the current modeling effort for this high-level modeling goal is $M(n \times m)$. This is because the modeler has to click through each node and look at each component in the model. If PICML provided a mechanism to view a component’s deployment status without clicking through each node, then modeling effort would be $M(n)$. This is because m actions were removed from this high-level modeling goal that was associated with n more actions. The modeler therefore need only look at the n components in the model, and deal with think time associated with looking at those n components in the deployment model.

Techniques for reducing modeling effort. Many DSML environments provide external artifacts that can be used to enhance a DSML’s functionality. If used properly within a DSML, such artifacts can also help reduce modeling effort. For example, the following is a non-exhaustive list of external artifacts provided by DSML environments that can be used to reduce modeling effort:

- **Model observers.** Model observers are artifacts that run in the background and observe modeling events. These events can include adding a new element, deleting an existing element, creating a connection between two elements, and changing the value of an attribute. When the model observer receives notification of an event that it has registered to receive, it reacts on the model by automating one or more actions that are completed manually by the modeler. In GME, model observers are called *add-ons*. DSML developers implement add-ons and distribute them with the DSML. As modelers complete modeling actions, the add-on receives notification of the modeling events. The DSML developer can then program the add-on to react to the events, which can help reduce modeling effort for given a modeling goal.
- **Model decorators.** Model decorators are artifacts that manage the visualization aspect of a model element. The goal of the model decorators is to provide different visual representations based on the model’s state. This therefore can reduce think time in model complexity, because model decorators can make elements more apparent depending on the modeling goal. In GME, model decorators are called *decorators*. DSML developers implement the decorators and assign them to a particular element in the DSML’s metamodel. When a modeler views the contents of a model and its child elements have an associated decorator, then the decorator will display the model element according to its programming.
- **Model solvers.** Model solvers are artifacts that complete parts of a model based on the current state of the model. In order to use a model solver, first the modeler completes a minimum portion of the model that is needed from the model solver to execute correctly.

²Even if the modelers knows what element he/she wants to add to the model, there is still a think time associated with completing the action.

The modeler then invokes the model solver and it completes the remaining portions of the model based on its programming. Model solvers are typically used to reduce complexity associated with solving modeling goals that are too hard to complete manually, such as solving the deployment strategy for a large-scale system consisting of hundreds of components and hosts based on component resource requirements and host resource availability.

In GME, model solvers are called *interpreters*. DSML developers manually implement interpreters to solve a particular problem. Modelers then model the necessary elements, and invoke the interpreter. The interpreter then completes the remaining portions of the model, thereby reducing modeling effort for that portion of the model.

Using the above external artifacts, it is possible to reduce modeling effort. This is because these artifacts try to reduce the number of model actions a modeler has to execute to complete a modeling goal, which in turn removes think time associated with the specific action. The external artifacts also operate on the existing DSML “as is.” This therefore implies it is not necessary to refactor the existing DSML’s metamodel to reduce modeling effort—given the DSML has the correct abstractions and semantics.

IV. EXAMPLES OF REDUCING MODEL USE COMPLEXITY IN PICML

This section presents examples of reducing modeling effort in the PICML, which was introduced in Section II. Each example is first introduced and then its modeling effort is analyzed. Lastly, an external artifact discussed in Section III-C is used to reduce the example’s modeling effort.

A. Defining a Unique Identifier for New Components

Problem description. In PICML, each component has a unique identifier (UUID) that distinguishes it from other `Component` model elements. This is necessary because the UUID is used in generated artifacts to cross-reference other components. For example, Figure 3 illustrates that the PICML metamodel requires all `Component` model elements to have a UUID.

When a modeler adds a new `Component` model element, or any element that requires an UUID, the modeler must manually define the identifier. More importantly, the modeler must ensure the identifier is unique when compared to all the other `Component` model elements that appear in the current model.

Modeling effort. In this example, there are three tasks that the modeler must complete in order to define a UUID for a component. First, the modeler has to define an identifier, which is $M(1)$. Second, the modeler must determine if the identifier is unique, which is $M(n)$. Finally, the modeler

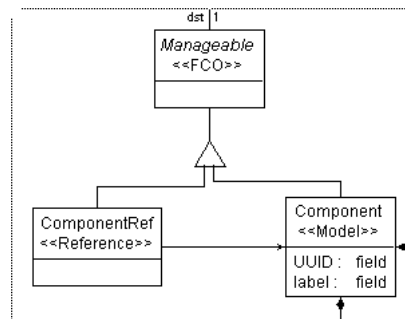


Figure 3. Unique identifier (UUID) attribute defined on the `Component` modeling element in PICML.

must set the value of the UUID attribute to that of the defined identifier³, which is also $M(1)$. Modeling effort of this example therefore is $M(n+2)$ where n is the number of elements that must be checked to determine if the identifier is unique. Again, the 2 remains in the complexity because there is associated think time with each modeling action, which is variable and not negligible.

Reducing modeling effort. To reduce modeling effort of this example, the *model observer* external artifact is used. More specifically, when a modeler adds a new component to the model, the GME add-on automatically generates an identifier and checks it for uniqueness against the other UUIDs defined in the model. If the identifier is unique, then the UUID attribute is set to the value of the identifier. Otherwise, the add-on repeats the process until a valid UUID is defined⁴.

Because of the add-on, defining a UUID for a new component now consists of one modeling action. This modeling action is adding a new component to the model. The add-on (or computer) then handles defining a UUID for the new component. Modeling effort of this particular goal is now $M(1)$, which is less than $M(n+2)$.

B. Defining the Value of a Component’s Attribute

Problem description. In PICML, components have attributes and these attributes have values. The values are used at deployment-time to configure each component. As shown in Figure 4, when defining the value of a component’s attribute a modeler must complete the following actions: (1) add a `Property` element to the model; (2) open the `Property` element; (3) add the correct `DataType` to the model, such as string or integer, where the type matches the type of the target attribute; (4) create a connection between

³This example makes the assumption that the identifier is found to be unique after the second step. If this were not true, then actions (1) and (2) repeat until a valid UUID is found.

⁴It is obvious that unique identifiers should be auto-generated, but the DSML developer must use an external artifact to achieve the necessary functionality. This is because such criteria is unknown to a general-purpose DSML environment. This example therefore illustrates why such a feature is necessary.

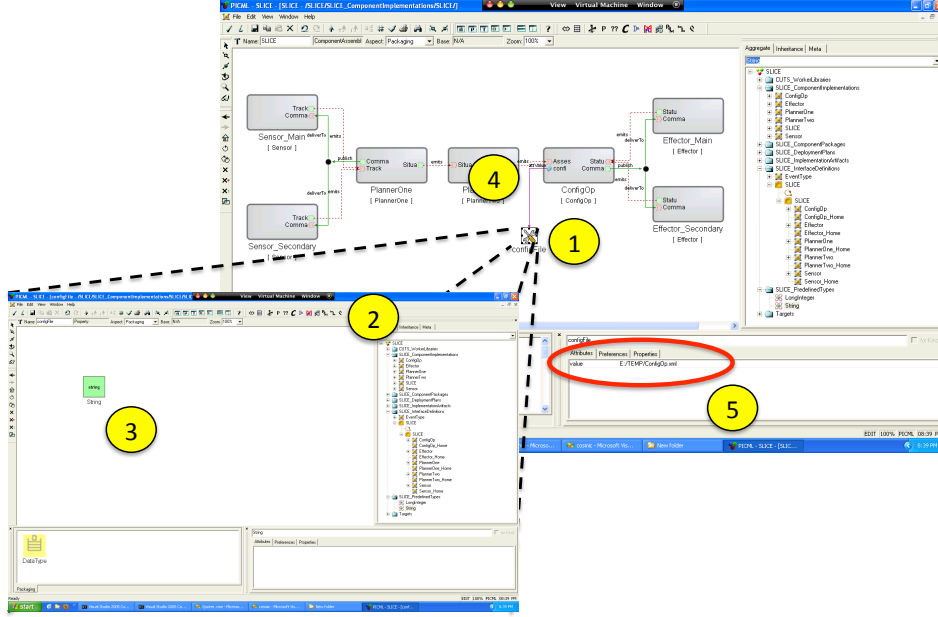


Figure 4. Example of defining the value of a component's attribute in PICML.

the target attribute and the `Property`; and (5) set the value of the attribute, which is a model attribute on the `Property` element.

Modeling effort. In this example, modeling effort is $M(5)$ because it is 5 single-step modeling actions. Although modeling effort for this example is constant, changing the data type of a component's attribute presents a more complex scenario. This is because all `Property` elements that connect with the attribute whose data type changed must also change their contained data types accordingly. Otherwise, the model is considered invalid because of data type mismatch. The modeling effort of this goal is therefore $M(n + 1)$, *i.e.*, changing the target attribute's data type and updating n elements to reflect the change. This goal, however, is more complex because the attributes that need to be changed (or updated) must be manually located, which typically involves a lot of think time and hidden (or accidental) actions not accounted for in the analysis above.

Reducing modeling effort. To reduce modeling effort of both goals in this example, PICML uses a *model observer*. More specifically, when defining the value of a component's attribute, the modeler (1) adds a `Property` element to the model, (2) connects the `Property` element to the target component's attribute, and (3) defines the value of the attribute using the model attribute on the `Property` element. When the modeler connects the target attribute to the `Property` element, the add-on automatically defines the `Property` data type (see Step 2–3 in the problem description above) based on the connected component's attribute. Modeling effort for this goal is therefore reduced to $M(3)$, which is less than $M(5)$.

When changing the data type of a component's attribute, the GME add-on also detects this modeling event. The add-on then locates all `Property` elements defined in the model that are connected to the attribute that changed its data type. Finally, the add-on changes the data type of located `Property` elements to match the data type of the connected component attribute. This therefore reduces modeling effort for this goal to $M(1)$, *i.e.*, changing the attribute's data type, which is less than $M(n + 1)$.

C. Determining a Component's Deployment Status

Problem description. In PICML, modelers create deployment models that determine what component instance to place on a given host in the run-time environment. As shown in Figure 2a, modelers create this model by adding component instances to GME sets, which represent an execution process on the target host. As shown in this Figure 2b, it is hard for a modeler to determine whether or not a component has been assigned to a host because all the component images (*i.e.*, the rounded-edge rectangles) look the same. In order to determine the deployment status of a component, *e.g.*, what component is not assigned to a host, the modeler has to iterate through each host (and its processes).

Modeling effort. In this example, modeling effort for determining a component instance's deployment status is $M(n \times m)$, as explained in Section III-A.

Reducing modeling effort. To reduce modeling effort of this example, PICML uses a *model decorator*. As shown in Figure 5, if the component has been deployed to host in the model, then it has the standard image. If a component has

not been deployed to a host, then it has an overlay image of an X. By using the decorator to highlight the deployment status of a component, modeling effort of this goal is now $M(n)$ because the modeler only has to look at the model to determine the necessary status of each component, or a single component.

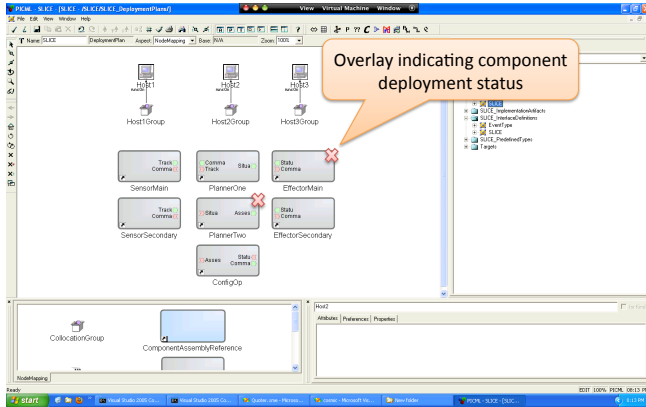


Figure 5. Using the decorator external artifact to highlight components that are not deployed, thereby reducing model complexity.

D. Modeling a Component's Implementation

Problem description. Modeling a component's implementation is considered one of the most complex modeling goals in PICML. This is because it requires three other high-level modeling goals. As shown in Figure 6, this goal requires (1) adding a new component to the model (see Section IV-A; (2) modeling the component implementation's artifacts (*i.e.*, the servant and executor binaries); and (3) associating the component implementation with its correct component type and component implementation artifacts. Moreover, each element added to the model has a UUID associated with it, and at least one model attribute that must be correctly defined, which is not shown in Figure 6.

Modeling effort. Given the steps above for modeling a component's implementation, where each key element has an associated UUID, modeling effort of this example is $M(4n+6)$ where $4n$ is for defining the unique identifier for each of the key element and 6 is for creating the four key elements and associating the component implementation with its correct implementation artifacts. This analysis, however, is not taking into account setting the values for the model attributes as explained in the problem description.

In this example, if defining the UUID is $M(1)$ (see Section IV-A), then its modeling effort is still $M(6)$. Similar to previous examples, $M(6)$ is high because there is a lot of think time associated with achieving this goal. Moreover, this think time increases for novice users, which accidentally increases modeling effort for this goal.

Reducing modeling effort. To reduce modeling effort associated with this goal, PICML uses a *model observer*.

More specifically, when a modeler adds a new component to the model, the GME add-on receives notification of the new component and displays a dialog to the modeler. As shown in Figure 7, the dialog contains the bare minimum information needed to auto-generate the required model elements for a component's implementation⁵.

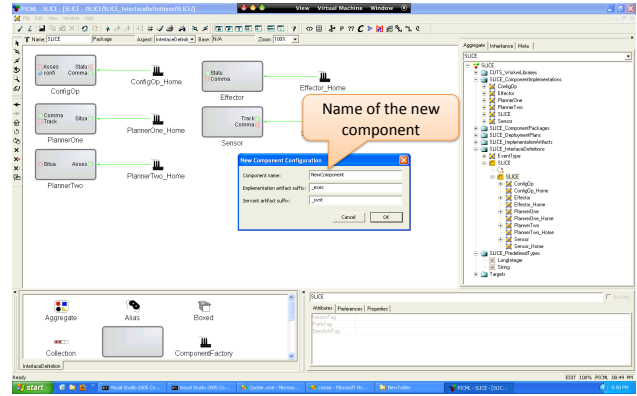


Figure 7. Dialog presented to the modeler that captures information needed to auto-generate a component's implementation.

The modeler then inputs the name of the component, changes the default values of the other text boxes (if applicable), and clicks the OK button. The add-on then automatically executes the actions that were once done manually by the modeler. Because of the PICML add-on, modeling effort of this goal is now $M(2)$, *i.e.*, adding a new component to the model and typing the name of the component.

V. DISCUSSION OF LESSONS LEARNED

Section IV discussed examples of high-level modeling goals in PICML with high modeling effort. Likewise, the modeling effort was reduced in each example using one of the techniques (or external artifacts) presented in Section III-C. Based on these experiences, and others not discussed in this paper, this section discusses the lessons learned from reducing modeling effort.

A. Model Observers Can Reduce Modeling Effort to a Constant

This is because the model observer automates many of the actions in a high-level modeling goal that were once completed manually by the modeler. As shown in Section IV-A, Section IV-B and Section IV-D, the model observer implemented for PICML was able to reduce modeling effort to either a constant, or a constant of lesser value.

Although the model observer reduced modeling effort of the examples above to a constant, this is not always

⁵The GME add-on does not manage the model elements once they have been created. This, however, can be another goal where modeling effort can be reduced.

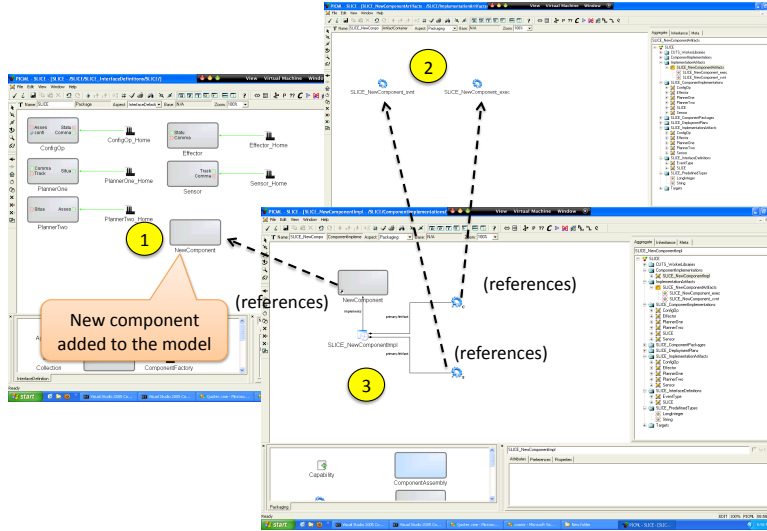


Figure 6. Modeling a component's implementation in PICML.

possible. For example, modeling effort of creating the unique identifier is reduced to $M(1)$ because it is possible to infer enough information about model to remove the manual actions. The same is true for modeling a component's default implementation, as presented in Section IV-D. It is possible to reduce its modeling effort because the values (or required attributes) of the auto-complete operations can be inferred based on (1) the current state of the model and/or (2) the name of the new component.

This implies that the model observer is most effective at reducing modeling effort when it is possible to auto-complete manual actions by inferring future actions based on the current state of the model. This also includes newly added elements from recently completed actions. If it is not possible to infer future actions, *i.e.*, there is no association between two actions in a given task, then the model observer is not effective. Moreover, it is *hard* to reduce modeling effort for such a scenario because it will require human intervention—thereby bringing the human (and think time) back into the equation.

B. Model Decorators Can Remove a Factor from Modeling Effort

This is because model decorators provide a mechanism for exposing *hidden* information. Instead of requiring a modeler to search for the information, which can be embedded somewhere in the model tree, the decorator can make it visible within the current model. As shown in Section IV-C, the decorator added to PICML for determining a component's deployment status exposes this information as an overlay on each component in the deployment model. This therefore removes the factor of searching each node to determine a component's deployment status.

Although model decorators can remove a factor from

modeling effort, it is *hard* for a model decorator to reduce modeling effort to a constant. This is because the decorator does not physically remove any *required* actions from the modeling task. Instead, the model decorator is removing extra (or unnecessary) actions from a modeling task. This implies that the model decorator is best used when modeling task has multiple factors influenced by hidden information that can be improved by presenting the modeler with an appropriate visualization—given the modeler understands the meaning of the model decorators visualization.

C. Feedback is Needed to Reduce Modeling Effort of User-specific Modeling Goals

Model observers, model decorators, and model solvers are techniques that can be used to reduce modeling effort. Using the definition and equation for measuring modeling effort, it is possible for a DSML developer to analyze their own DSML to (1) determine what high-level modeling goals have high modeling effort and (2) what techniques can be used to reduce such complexity.

This approach is very effective for the *common* high-level modeling goals. For example, modeling a component's implementation or checking a component's deployment status are *common* high-level modeling goals. It is therefore easy to improve modeling effort of such goals since it is how PICML is expected to be used by a modeler.

Some modeling goals, however, are not common. These modeling goals are referred to as *user-specific modeling goals* because the goals depend on how the user (or modeler) intends to use the DSML. In such cases, it is *hard* for a DSML developer to know what high-level goals—outside of the norm—have high modeling effort. This implies that DSML developers must receive feedback from modelers on what user-specific modeling goals have high modeling

effort. Once such feedback is received, DSML developers can leverage the methodology for measuring and reducing modeling effort (as discussed in the paper) to improve the DSML—given such improvements do not negatively impact the DSML’s common modeling goals.

VI. RELATED WORKS

Grammar usability and effectiveness has been heavily studied in prior works of literature. For example, Wand and Weber [13], [14] defined a set of ontological constructs to describe all real-world phenomena captured via a modeling grammar. Likewise, Green et al. [15] present a framework for measuring the usability of visual programming environments, such as DSMLs, based on cognitive dimensions, such as abstraction gradient, error-proneness, and hidden dependencies. Finally, Recker et al. [16] present a study that develops an instrument for measuring the user perception of ontological deficiencies based on the theory of ontological expressiveness. In these works, and others not mentioned, the results are based on using the visual grammar and end-user studies. The work presented in this paper also contributes to current research on understanding the effectiveness of graphical languages (or grammars) in that it defines a metric for measuring modeling effort. This work differs in that it does not require end-user studies to analyze modeling effort (*i.e.*, the DSML developer can conduct such analysis on their own) or base its analysis on traits/characteristics possessed by a graphical language.

Mernik et. al [9] discuss “how and when” to develop a domain-specific language (DSL). In this paper, the authors provide a general overview of five critical steps when developing a DSL namely: decision, analysis, design, implementation, and deployment. These steps are also critical when developing a DSML. This paper, however, provides more details on the analysis, design, and implementation phase of designing an DSML in relation to modeling effort and improving DSML quality. Mernik et. al also discuss the notion of *task automation*, which are generators that remove repetitive tasks done manually by DSL users. Similar to this concept, this paper illustrates how model observers can be used for task automation.

Model guidance [24], [25] techniques have been used to reduce modeling effort by pointing modelers in the right direction for completing a modeling goal. Likewise, White et. al [11] and Sagar et. al [26] have demonstrated how model intelligence can be used to complete portions of the model automatically for the modeler. This paper differs from earlier work in that earlier work looks at model complexity from the modeler’s ability to locate a valid solution in large solution space, such as creating a valid deployment model based on component resource requirements and host resource availability (*i.e.*, bin-packing [27]). This work, however, looks at modeling effort as the number of modeling

actions a modeler executes in order to complete a modeling goal.

Finally, Sprinkle [28] and Wu et al. [29] discuss techniques for analyzing modeling effort (or complexity). Sprinkle uses the model creation events and trajectories to measure complexity of exercise (*i.e.*, modeling effort). The higher the complexity of exercise, the more complex the DSML. Wu, however, extends this effort and incorporates more actions into measuring modeling effort, such as mouse clicks, keystroke input, and drag and drop. Similar to both Sprinkle and (more so) Wu, this work views those actions as measurable. This work, however, extends their notions of measuring modeling effort with *think time*. Moreover, this work discusses techniques for reducing modeling effort, and not only measuring it.

VII. CONCLUDING REMARKS

The design and functionality of a DSML can greatly impact modeling and overall DSML efficiency and effectiveness. This paper therefore defined modeling effort and discussed how to analyze it in the context of a representative DSML named the *Platform Independent Component Modeling Language (PICML)*. Likewise, examples that illustrate how to reduce modeling effort was presented. Based on lessons learned the following are directions for future research:

- **Compare and contrast with ontological expressiveness.** There has been a lot of existing research on the ontological expressiveness of grammars and determining if such a grammar is adequate for modeling concerns of the target domain. Such research, however, has been conducted based on case studies. Future research therefore is to conduct similar studies to determine if the techniques presented in this paper for measuring modeling effort produce similar outcomes.
- **Cognitive studies to improve model decorator effectiveness.** Model decorators can reduce think time associated with an action. The effectiveness of this ability, however, is highly dependent on how the modeler perceives the decorators visualizations. Future research therefore involves leverage existing research in cognitive studies of graphical languages to better (1) understand how to improve model decorator effectiveness and (2) understand how the different cognitive dimensions relate to modeling effort.
- **Patterns for reducing model complexity.** As similar problems related to modeling effort arise, patterns for reducing modeling effort will emerge. Future research therefore includes continuing to perform such analysis on DSMLs to identify such patterns. This will enable such patterns to be incorporated into general-purpose DSML environments so they are usable by all DSMLs.

PICML is available for download in open-source format from www.dre.vanderbilt.edu/cosmic.

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] Kelly, S., Tolvanen, J.P.: *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons (2008)
- [3] Schmidt, D.C.: Model-Driven Engineering. *IEEE Computer* **39**(2) (2006) 25–31
- [4] Cook, S., Jones, G., Kent, S., Wills, A.C.: *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley (2007)
- [5] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: *Eclipse Modeling Framework*. Addison-Wesley, Reading, MA (2003)
- [6] White, J., Schmidt, D.C., Nechypurenko, A., Wuchner, E.: Introduction to the Generic Eclipse Modeling System. *Eclipse Magazine* **7** (2007)
- [7] OMG: Deployment and Configuration of Component-based Distributed Applications, v4.0. Document formal/2006-04-02 edn. (April 2006)
- [8] 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)
- [9] Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-specific Languages. *ACM Computing Surveys* **37**(4) (2005) 316–344
- [10] France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: *2007 Future of Software Engineering*. (2007) 37–54
- [11] Nechypurenko, A., Wuchner, E., White, J., Schmidt, D.C.: Application of Aspect-based Modeling and Weaving for Complexity Reduction in Development of Automotive Distributed Realtime Embedded Systems. In: *Proceedings to the Sixth International Conference on Aspect-Oriented Software Development*, Vancouver, British Columbia (March 2007)
- [12] Denton, T., Jones, E., Srinivasan, S., Owens, K., Buskens, R.: NAOMI-An Experimental Platform for Multi-modeling. In: *Proceedings of MODELS*, Toulouse, France (October 2008) 143–157
- [13] Wand, Y., Weber, R.: An Ontological Model of an Information System. *IEEE Transactions on Software Engineering* **16**(11) (1990) 1282–1292
- [14] Wand, Y., Weber, R.: On the Ontological Expressiveness of Information Systems Analysis and Design Grammars. *Journal of Information Systems* **3**(4) (1993) 217–237
- [15] Green, T.R.G., Petre, M.: Usability Analysis of Visual Programming Environments: a ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing* **7** (1996) 131–174
- [16] Recker, J.C., Rosemann, M.: The Measurement of Perceived Ontological Deficiencies of Conceptual Modeling Grammars. *Knowledge and Data Engineering* **69**(5) (2010) 516–532
- [17] Object Management Group: CORBA Components v4.0. OMG Document formal/2006-04-01 edn. (April 2006)
- [18] Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DANCE: A QoS-enabled Component Deployment and Configuration Engine. In: *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, Grenoble, France (November 2005) 67–82
- [19] 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)
- [20] Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering* **5**(0) (January 1998) 143–168
- [21] Nechypurenko, A., White, J., Wuchner, E., Schmidt, D.C.: Applying Model Intelligence Frameworks for Deployment Problem in Real-time and Embedded Systems. In: *MARTES: Modeling and Analysis of Real-Time and Embedded Systems workshop*, Genova, Italy (October 2006)
- [22] Smith, C., Williams, L.: *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA (September 2001)
- [23] Menasce, D.A., Dowdy, L.W., Almeida, V.A.F.: *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA (2004)
- [24] Hessellund, A., Czarnecki, K., Wsowski, A.: Guided Development with Multiple Domain-Specific Languages. In: *ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems*. (2007)
- [25] White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A.: Model Intelligence: an Approach to Modeling Guidance. *Novtica* **8**(192) (March 2008)
- [26] Sen, S., Baudry, B., Vangheluwe, H.: Towards Domain-specific Model Editors with Automatic Model Completion. *Simulation* **86**(2) (2010) 109–126
- [27] Fernandez de la Vega, W., Lueker, G.: Bin packing can be solved within $1 + \varepsilon$ in linear time. *Combinatorica* **1**(4) (1981) 349–355
- [28] Sprinkle, J.: Analysis of a Metamodel to Estimate Complexity of Using a Domain-Specific Language. In: *Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM’10)*, Reno, NV (October 2010)
- [29] Wu, Y., Hernandez, F., Ortega, F., Clarke, P.J., France, R.: Analysis of a Metamodel to Estimate Complexity of Using a Domain-Specific Language. In: *Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM’10)*, Reno, NV (October 2010)