Modeling Interface Definition Language Extensions (IDL3+) using Domain-Specific Modeling Languages

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

***Abstract*—Model-driven engineering (MDE) of distributed real-time and embedded (DRE) systems built using distributed middleware technologies typically rely on interface definition language (IDL) to define interfaces and attributes of the system under development. Recent needs for using IDL to design and implement systems composed of heterogeneous communication architectures, however, has realized the limitations of IDL. To address these limitations, vendors have proposed several non-trivial extensions to IDL—also known as IDL3+. In order to leverage such extensions in the modeling domain, it is necessary to update existing tools (*e.g.*, domain-specific modeling languages) to support such extensions.**

This paper provides two contributions to MDE of DRE systems using domain-specific modeling languages (DSMLs). First, this paper highlights the technical challenges associated with modeling IDL3+. Secondly, this paper discusses how to overcome such challenges in the context of a representative DSML for modeling DRE systems designed and implemented using IDL3+. Experience gained from using DSMLs to model IDL3+ shows that DSML environments—as is—do not suffice and need improved application frameworks to support complex DSMLs, such as IDL3+.

I. Introduction

Model-driven engineering (MDE) [20] techniques, such as domain-specific modeling languages (DSMLs) [8], are increasingly being used to design and implement enterprise distributed real-time and embedded (DRE) systems, *e.g.*, shipboard computing environments, traffic management systems, and manufacturing and controls systems. One main benefits of using DSMLs to design and implement such systems is that DSMLs shield DRE system developers from both the inherent and accidental complexities of the domain. Moreover, its intuitive graphical abstractions and model interpreters can increase the level-of-abstraction and the improve processes that were originally tedious and error-prone to complete if done manually or in an *ad hoc* manner, such as manually crafting dense XML deployment descriptors or validating deployment models.

When designing and implementing enterprise DRE systems, DRE system developer typically begin with the interface definition language (IDL) to define the interfaces and attributes of the DRE system under development. The crafted IDL is then used to implement individual components (or objects) that constitute the building blocks of the final system. Because DRE system developers may not completely understand IDL, *e.g.*, its semantics and syntax, DSMLs have been used to shield DRE system developers from this complexity. Instead of manually handcrafting IDL, DRE system developers use a DSML to model the interface and attributes of the DRE system. DRE system developers then use model interpreters to automatically generate IDL from constructed models.

Enterprise DRE systems, however, are growing in both size and complexity [7]. Moreover, their needs and envisioned application domains (*i.e.*, deployment locations) is continuing to grow. The result of this demand is that it is no longer suitable to design and implement an enterprise DRE system using a single communication middleware, such as the Common Object Request Broker Architecture (CORBA) [12]–[14]. Instead, many enterprise systems are now being composed from heterogeneous architectures, such CORBA, Data Distribution Services (DDS) [9], [18], [19], and legacy architectures. The advantage of using different architectures is that DRE system developers can select the best suitable technology to address individual design challenges in the overall problem space.

Unfortunately, traditional IDL (currently IDL3 [12]) does not provide adequate support for constructing enterprise DRE systems composed from heterogeneous architectures. This is not to say that it is impossible to use IDL to construct heterogeneous architectures, but realizing heterogeneous architectures built from IDL specifications requires substantial amounts of time and effort. To address this limitation, vendors have proposed an extension named *IDL3+* [15]. The goal of IDL3+ is focus on generalizing the interactions between components (such as their communication architecture) so such details can be determined at deployment time (*i.e.*, when the system's components are being assembled).

Since DSMLs have been successfully used to model IDL (*i.e.*, IDL3) and generate IDL files, it is only natural to update existing DSMLs to support IDL3+. Unfortunately, unlike its predecessor, IDL3+ is not a trivial language. This paper therefore presents an approach for modeling

IDL3+, and experience gained from modeling IDL3+ using DSMLs. In particular, this paper shows how IDL3+ was realized in the *Platform Independent Component Model Language (PICML)* [1], [2], which is a DSML for modeling component-based enterprise DRE systems. The main contributions of this paper are as follows:

- It presents design alternatives—along with advantages and disadvantages—for constructing a metamodel for modeling IDL3+;
- It shows how to model template languages (a feature of IDL3+) using a DSML; and
- Its presents lessons learned for constructing a DSML that must rely heavily on dynamic semantics to construct and maintain valid models.

Finally, experience from updating PICML to support IDL3+ shows that although DSML environments provide means to support complex DSMLs, without a comprehensive application framework it is *hard* to realize complex DSMLs that are intuitive for the modeler to use effectively.

Paper organization. The remainder of this paper is organized as follows: Section II provides details of IDL3+ and PICML; Section III discusses the technical challenges and solutions for updating PICML to support IDL3+; Section IV illustrates examples of IDL3+ in PICML; Section V discusses related works; and Section VI provides concluding remarks.

II. The Platform Independent Component Modeling Language and IDL3+

The Platform Independent Component Modeling Language (PICML) is a DSML for developing component-based distributed systems. PICML is designed and implemented in the Generic Modeling Environment [8]. PICML's metamodel is based on different OMG (www.omg.org) specifications for developing distributed systems, such as the CORBA Component Model [11] and the deployment and configuration specification [10].

One of PICML's main functionalities is modeling a distributed system's interfaces and attributes. This is accomplished by using graphical modeling elements that represent CORBA's IDL constructs. As stated in Section I, DRE system developers then use model interpreters to generate valid IDL from constructed models.

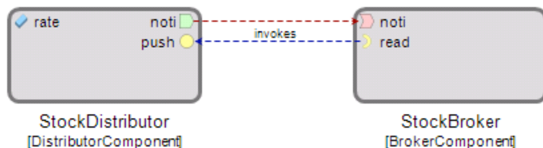

Figure 1. Assembly view component-based DRE system in PICML.

The modeled IDL also serves as building blocks for other modeling aspects in PICML. For example, Figure 1 highlights the assembly view of a component-based DRE system in PICML. The assembly view captures the component instances and their communication connections, such as event source/sink and facet/receptacle connections. As shown in this figure, there are two component instances (*i.e.*, `StockBroker` and `StockDistributor`) that are instances of a component model in IDL (*i.e.*, `DistributorComponent` and `BrokerComponent`, respectively).

Current implementations of distributed middleware architectures and their associated modeling tools assume that DRE systems are homogeneous. For example, the connections between `StockBroker` and `StockDistributor` are assumed to use the same technology. There, however, is growing need to developing DRE systems composed of many different distributed middleware architectures. To address this need, as explained in Section I, vendors have proposed a new specification called IDL3+. Listing 1 highlights an example of IDL3+.

```
1  module Typed <typename T>
2  {
3    struct Pair
4    {
5      T first;
6      T second;
7    };
8
9    interface A { };
10
11   interface B { };
12
13   porttype ComboPort
14   {
15     uses A a_port;
16     provides B b_port;
17   };
18 };
19
20 ::Typed <unsigned long> ULong_Typed;
21
22 struct Stock
23 {
24   ::ULong_Typed::Pair price;
25 };
26
27 connector F
28 {
29   port ComboPort p1;
30   mirrorport ComboPort p2;
31 };
```

Listing 1. Example of the proposed IDL3+.

As shown in Listing 1, IDL3+'s main extensions to IDL3 are parameterized modules (line 1), extended ports (line 13), template module instances (line 20), and connectors (line 27). Parameterized modules have semantics similar to templates in C++. Extended ports are a collection of facet/receptacle connections that must *always* be connected together. Template module instances are an instantiation of a template module with concrete values. Finally, connectors are abstractions that act as a gateway between traditional CORBA components and a different network architecture/middleware, such as RTI-DDS [19] and OpenSplice DDS [18].

Updating PICML to handle extended ports and connectors in IDL3+ is a trivial process. This is because extended ports and connectors can map directly to modeling elements used to construct PICML's metamodel. Moreover, their modeling semantics are similar to other model elements in PICML, such as facets/receptacles and components, respectively.

Updating PICML to handle template modules and template module instances, however, is not a trivial process. This is because elements in the instantiated parameterized module (line 20) can be used at later points in IDL (line 24). It is therefore critical that updates to PICML for supporting IDL3+ take into account such usage, and do so in a effective manner. The remainder of this paper therefore discusses how PICML was updated to support IDL3+, with emphasis on handling template modules.

III. A METAMODEL FOR IDL3+

This section discusses the design and implementation of a metamodel for IDL3+. This section also discusses the advantages and disadvantages of several design choices considered when extending PICML to support IDL3+.

A. Design Alternatives for Modeling Template Modules

As stated in Section II, constructing a model for template modules and their instances is not a trivial process. The main reason is because elements inside a template module instance can be referenced by other elements in the model, such as an attribute, in/out parameter, and facet/receptacle. In order to model a template instance, it is necessary to first model template modules.

The following are two design alternatives considered for modeling template modules in IDL3+:

- **Inheritance-based methodology.** The *inheritance-based* metamodel design uses the "is-a" methodology [5] to define template modules. As shown in Figure 2, the template module (`TemplateModule`) inherits its semantics from a standard module (`Module`). Likewise, the template module element contains one or more template parameters (`TemplateParameter`). The template parameter element is abstract in that it can be any template parameter supported by IDL3+ (not pictured), such as generic types, exceptions, and interfaces.
  The advantage of this approach is the metamodel directly correlates with how a template model is viewed conceptually. For example, a template model "is-a" standard module with parameters. In addition, the template module inherits the standard modules semantics, which includes its attributes and child model elements (*e.g.*, structures, components, and other modules).
  The disadvantage of this approach is that it is *hard* to convert between a standard module and a template module. For example, if a DRE system developer want to convert a standard module to a template module, they have to create a template module element and copy all the elements in the standard module to the template module. Although this disadvantage may seem like little effort, it does not scale well with large and complex models. The same holds true when converting a template module to a standard module.

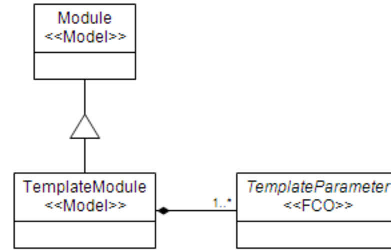

Figure 2. Illustration of the inheritance-based metamodel for template modules in IDL3+.

- **State-based methodology.** The *state-based* methodology uses containment to define template modules. As shown in Figure 3, the standard module and template module are one in the same. The difference between a standard module and a template module is that the template module has one or more template parameters. This is signified by the cardinality on the template parameter element.

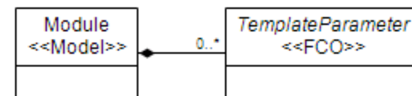

Figure 3. Illustration of the state-based metamodel for template modules in IDL3+.

  The advantage of this approach is that it is easier to convert between a standard module and a template module since they are one in the same element. This approach also directly correlates to the "is-a" terminology, but not literally in terms of object-relations (*i.e.*, there is not a separate model element for a standard module and a template module).
  The main disadvantage of this approach is that DSML developers cannot rely on separate objects to distinguish between a standard module and template module. Instead, the DSML developer must rely on external capabilities (*e.g.*, model decorators, which are domain-specific components that can be used to create custom graphics for model elements above and beyond the standard icon).

Based on the advantages and disadvantages of the two design alternatives above, the state-based methodology was selected to model template modules in PICML. This design method was selected because it requires less effort to convert between standard and template modules.

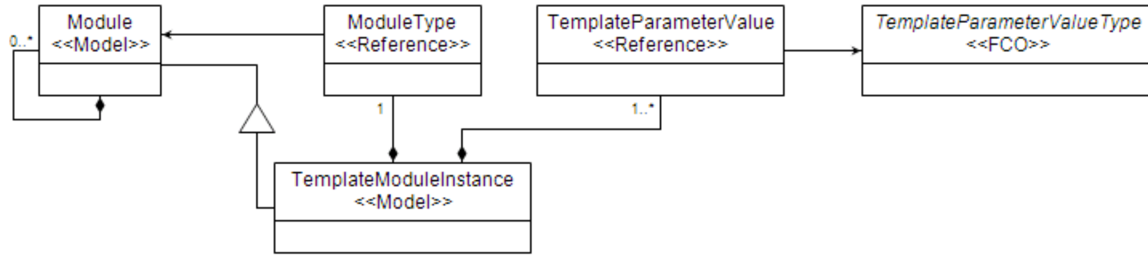

Figure 5. Key elements of the metamodel for modeling template module instance elements.

Figure 4. Example of a standard module (left) and template module (right) in PICML.

Figure 4 illustrates an example template module in PICML. Since PICML uses a state-based design for template modules, by default the standard and template module have the same icons. To distinguish between a standard and template module, PICML uses a decorator. If the module contains one or more template parameters the decorator draws an icon that represents a template module as shown by the right icon in Figure 4. Otherwise, the module has the image on the left of Figure 4.

B. Metamodel for Modeling Template Module Instances

A template module instance in IDL3+ is an instantiation of an existing template module. When the template module is instantiated, the elements in the template module can be referenced by other elements outside of the template module instance. Based on this understanding of a template module instance, Figure 5 shows the key elements on the updated PICML metamodel modeling template instances.

As shown in Figure 5, a template module instance (`TemplateModuleInstance`) is a subclass of a `Module`. This design was selected because it allows a template module instance to contain the same elements as a standard module. It is therefore possible to reference concrete elements in the template package instance, which is inline with its expectations.

The template module also contains a reference to the target template module that it is instantiating. Finally, the template module instance contains a set of parameter values. Each parameter value elements is the value for its corresponding parameter in the template module.

C. Modeling Template Modules and Their Instances

The previous sections discussed the metamodel for template modules and template module instances. The previous sections also introduced how both template modules and template module instances are used in the extended version of PICML. Although there is a metamodel for modeling both template modules and their instantiations, using template module instances is not a trivial process. This is because the metamodel above only describes what a template model instance contains. It does not take into account how the elements in the template module instance are created (or should be created).

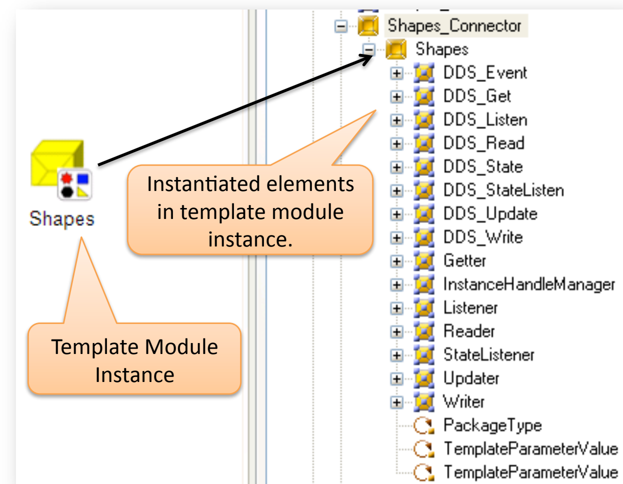

Figure 6. Snapshot of an example illustrating the contents of a template module instance in PICML.

For example, Figure 6 shows an example of a template module instance in PICML. As shown in this example, the template module instance contains a large number of elements, which is based on the template module being instantiated. When the template module instance shown in Figure 6 is instantiated, the "concrete" elements must be made available for usage based on the metamodel described in Figure 5. Unfortunately, this does not happen automatically because the modeling environment does not understand this domain-specific requirement.

To address this challenge problem, PICML relies on domain-specific add-ons in GME. GME add-ons are components that listen for modeling events, and react on the

model based on its programming. For example, an add-on can receive an element creation event and set the newly create element's attribute(s) to a predefined value. Likewise, the add-on can create required elements when a specific element is created.

The following is a list of different add-ons implemented in PICML to support correct usage of template modules and template module instances:

- **Template Module Instance Builder.** The template module instance builder is responsible for populating a template module instance with elements from the corresponding template module it is instantiating. When a modeler inserts a template model instance element, this add-on assists the modeler in selecting the target template module to instantiate and each of the parameter values in the template module instance.
  Once the modeler has selected the target template module to instantiate, and its corresponding template parameter values, this add-on creates a duplicate copy of all elements in the template module in the template module instance. In addition, when the template module instance builder encounters an element that is a template parameter, it is replaced with its corresponding template parameter value. Finally, this add-on sets each duplicate elements (or copies) to read-only. This prevents the modeler from accidently deleting elements in the template module instance.
- **Parameter Listener.** The parameter listener add-on is responsible for observing changes to template parameters and template parameter values. If the parameters of template module change (*i.e.*, either added or removed), then this add-on is responsible for making the corresponding update to all template module instance elements in the model.
  The parameter listener add-on is also responsible for observing changes to the template module instances parameter values. For example, if one of the parameter values change, then the parameter listener will update all elements to reference the new template parameter value. This is accomplished by using the Observer [5] software design pattern.
  When the template module instance is initially populated, this add-on also keeps track of what elements use what parameter value in the template module instance. When the value of the template parameter changes, this add-on iterates over the set of elements associated with that template parameter, and makes the necessary update.
- **Module Change Listener.** The module change listener is responsible for observing changes to elements in modules that are contained in a template module. Such changes can include adding new elements to a template module, deleting existing elements, moving elements to a new location, and changing an elements attributes. When the elements in a template module change, this add-on performs the same modification to all template module instances that are instantiated from the corresponding template module.
  The module change listener add-on can also be viewed as a *model instance emulator* for GME. This is because a model instance is an element that is instantiated from another element in GME. Whenever a change occurs to the archetype element, the same change occurs to the model instance. The module change listener add-on therefore provides the same functionality as a GME model instance. The add-on is used instead of a GME model instance because functionality needed to handle module changes is more domain-specific than GME model instance elements.

D. Modeling Extended Ports

Extended ports in IDL3+ are a collection of object ports, *i.e.*, facets and receptacles, that are connected at the same time. Because PICML already has support for modeling facets and receptacles via components, updating PICML to handle extended ports is a trivial process. It is therefore included in this paper for completeness of updating PICML to support IDL3+.

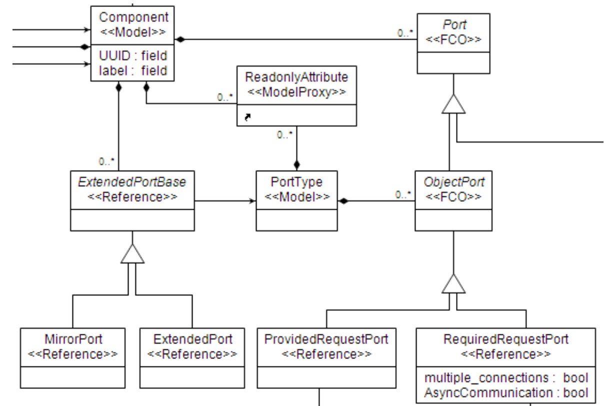

Figure 7. Metamodel for modeling extended ports in PICML.

Figure 7 illustrates the portion of PICML's metamodel for extended ports. As shown in this figure, an extended port, either `ExtendedPort` or `MirrorPort`, contains references a port type (`PortType`). The port type element contains one or more object ports, *i.e.*, facet (`ProvidedRequestPort`) and receptacle (`RequiredRequestPort`) elements. Finally, one or more extended port element is contained in a component or connector element (see next section).

E. Modeling Connectors

The last major extension IDL3+ adds to IDL is connectors. Connectors, in essence, are a gateway between

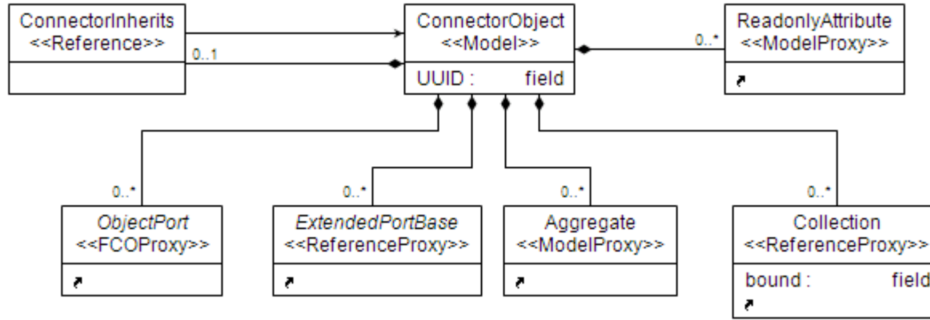

Figure 8. Metamodel for modeling connectors in PICML.

traditional CORBA components and different distributed middleware architectures. For example, the connector can be used to enable CORBA Component Model (CCM) components to communicate with DDS ports. Likewise, connectors can be used to facilitate communication with a proprietary communication architecture.

Connectors also provide ports, both standard and extended ports, and have attributes. Because of the connector's design, their structure and functionality resemble regular components. Figure 8 therefore shows the metamodel for modeling connectors in PICML. As shown in this figure, connectors are model elements that contain zero or more ports (`ObjectPort`) and extended ports (`ExtendedPortBase`). In addition, connectors contain zero or more attributes, as explained above. Finally, as shown in Figure 8, connectors can contain structure (`Aggregate`) and sequence (`Collection`) declarations, similar to traditional CORBA components.

IV. Using IDL3+ to Model and Deploy Enterprise DRE Systems

Using the extended version of PICML, it is possible for DRE system developers to model DRE system that want to leverage IDL3+. The previous section, however, only discussed concerns that related to using IDL3+ for specifying a systems interfaces and attributes. To truly leverage IDL3+, it is necessary for DRE system developers to construct assembly models, which are instances of components and connectors and the connections between components.

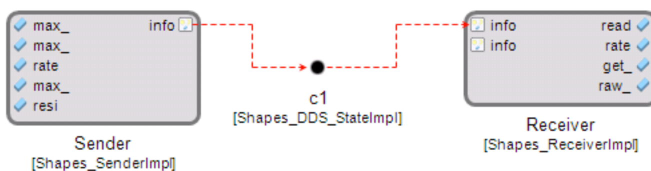

Figure 9. Example of a component assembly that uses IDL3+ constructs.

Figure 9 shows an example of a PICML assembly model that supports IDL3+. As shown in this example, each rectangular boxes represents a component instance. The component instance abstractions already existed in PICML before it was extended to support IDL3+. The black filled circles in Figure 9 are connector instances (*i.e.*, a model instance of a connector element from Section III-E). This abstraction was added to PICML's component assemblies to support IDL3+.

The method for creating a connection between a component's port and a connector is different from the method for creating a connection between ports on different components (see Figure 1 in Section II). When creating a connection between the port of a component instance and a connector instance, the connection goes from the component port's directly to the target connector. The name of the connection is then used to determine what port on the connector to connect with. This method was selected to not let the connector element overpower the modeling palette since it can have a large number of ports. If all ports are visualizable, then the connector will be larger than many component instances.

It can be hard for DRE system developers to know the exact name of the port on the connector to connect with. To handle this challenge, PICML implements an add-on that assists with setting the connections name. If there is only one port of the target type exposed by the connector, then the add-on automatically selects that name. If there is more than one port of the expected port type, then the add-on displays a dialog that contains a list of possible names. The DRE system developer then selects that correct name and the add-on sets the name of the connection accordingly.

Once DRE system developers construct the assembly view of the system from its component and connector types, they specify a deployment model. The deployment model is then used to generate deployment and configuration (D&C) information. In the case of PICML, it is used to generate D&C information that conforms to OMG's D&C specification.

The deployment model with PICML did not have to change to support IDL3+. The reason is because connector instances are not explicitly deployed to a host in the target domain. Instead, the model interpreter must infer the de-

ployment location based on connection between a connector fragment, which is the portion of the connector deployed on a host, and a component. The deployment plan generator therefore was updated to deploy connector fragments on the same host as the connect component instance. Each connector fragment then acted the a gateway between its connected component and the target technology.

V. Related Works

To the best of the author's knowledge, this is the first modeling tool that has been updated to support IDL3+. The reason being is that IDL3+ is a new specification and is still be adopted by vendors. Because there are no tools that currently support modeling IDL3+, this work is compared against tools from other domains that support modeling template specifications. In particular, tools that support modeling UML are compared against this work.

There are several UML tools that exist—many which are used to generate Java source code. With the recent adoption generics in Java, many UML-based tools are starting to support template-like constructs. For example, Papyrus UML [17] and Rational Software Modeler [6] are two UML tools that support modeling templates in class specifications—similar to IDL3+ in PICML.

The IDL3+ extensions to PICML, however, are different from Papyrus UML in that the instantiation of the template parameters requires more modeling effort [16] than PICML's approach. The IDL3+ extensions to PICML are more closely related to the template support in Rational Software Modeler. This is because when all template parameters are selected, the template instantiation is automatically created. Moreover, the instantiated element is available for use.

Finally, Bruck [3] has shown how to define Java generics using UML and transform the UML model to an Ecore [4] specification. In order to support this mapping, the Ecore profile has to be extended. Similar to the PICML metamodel to support IDL3+, the extended Ecore profile contains the notion of a template parameter. The IDL3+ extensions to the PICML metamodel extend this work by adding template parameter values and instantiated elements. This allows the modeler to use concrete elements that result from the template instantiation.

VI. Concluding Remarks

This paper discussed the design and implementation of a metamodel for modeling IDL3+. In particular, this paper illustrated how the *Platform Independent Component Modeling Language (PICML)* was updated to support IDL3+. As shown throughout this paper, design choices for a DSML's metamodel is critical to guaranteeing the DSML is intuitive to use. Moreover, design choices should be made so that the DSML's usage requires no more than the expected modeling effort. Otherwise, modelers will have a hard time using the DSML to realize domain-specific concepts.

Throughout the process of updating PICML to support IDL3+, it was learned that generic modeling environments do not provide—out-of-the-box—the necessary constructs to realize complex DSMLs. For example, add-ons were needed to help synchronize template modules and template module instance elements. It is therefore the responsibility of the DSML developer to understand this limitation in order to keep the DSML easy to use.

Unfortunately, without adequate application frameworks for generic modeling environments, such as GME, DSML developers will constantly reinvent logic that underpins domain-specific additions realized via add-ons and decorators. It is therefore critical that such environments begin providing application frameworks that capture common patterns for domain-specific extensions so they are available for all DSML developers to capitalize on.

Acknowledgements

The author would like to acknowledge Mark Haymen and John Prestel from Northrop Grumman Corporation. Their feedback on design choices for updating PICML to support IDL3+ was invaluable during the process. In addition, the authors would like to acknowledge Johnny Willemsen from Remedy IT for his assistance in understanding IDL3+. Finally, the author would like to acknowledge Will Otte and Jeff Parsons from Vanderbilt University for their feedback on generating D&C information for IDL3+ that is compliant to OMG's D&C specification.

The updated version of PICML that supports IDL3+ is freely available for download in open-source format from the following location: www.dre.vanderbilt.edu/cosmic.

References

[1] K. Balasubramanian. *Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Sept. 2007.

[2] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society.

[3] J. Bruck. Defining Generics with UML Templates. www.eclipse.org/articles/article.php?file=Article-Defining-Generics-wit%h-UML-Templates/index.html.

[4] Eclipse Modeling. Eclipse Modeling Framework Project (EMF). www.eclipse.org/modeling/emf/?project=emf.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[6] IBM. Rational Software Modeler. www-01.ibm.com/software/awdtools/modeler/swmodeler.

[7] S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, June 2006.

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

[9] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.0 edition, Mar. 2003.

[10] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.

[11] Object Management Group. *CORBA Components v4.0*, OMG Document formal/2006-04-01 edition, Apr. 2006.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces*, OMG Document formal/2008-01-04 edition, Jan. 2008.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 2: CORBA Interoperability*, OMG Document formal/2008-01-07 edition, Jan. 2008.

[14] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 edition, Jan. 2008.

[15] Object Management Group. *DDS for Lightweight CCM (DDS4CCM)*, ptc/2009-10-25 edition, February 2009.

[16] Papyrus UML. How to Use Templates in UML Models. www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?P=1%44&L=EN#7.

[17] Papyrus UML. Papyrus UML. www.papyrusuml.org.

[18] Prism Technologies. OpenSplice Data Distribution Service. www.prismtechnologies.com/, 2006.

[19] Real-time Innovations. NDDS: The Real-time Publish-Subscribe Middleware. www.rti.com/products/ndds/ndwp0899.pdf, 1999.

[20] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.