Domain Hierarchies: a Basic Theoretical Framework for Integrating Software Domains

Pierre Kelsen and Qin Ma
Laboratory for Advanced Software Systems, University of Luxembourg
6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg
Email: {Pierre.Kelsen, Qin.Ma}@uni.lu

***Abstract*—We present a new approach to executable modeling that borrows from executable UML the notion of domains and bridges and couches them in a formal abstract framework based on the novel concept of a domain hierarchy. The framework is independent of the language used for representing structure and behavior of domains and bridges. By plugging in a declarative executable modeling language with a formal semantics for representing both structure and behavior, we instantiate the abstract framework into a concrete framework that shares with executable UML the benefits of a high-level separation of the platform-independent model into domains and bridges while providing a formal and declarative description of the underlying models.**

I. Introduction

Complex software applications deal with many different subject matters. For example, a Web Application for electronic banking tackles: business objects relevant for the banking application; the graphical web-based user interface; security policies, to name just a few. Ideally a model-driven approach for designing software applications would build on high-level subject matter representations as reusable assets and add additional plumbing to ensure the overall system conforms to the user requirements. In the context of software development (as opposed to ontology development for instance [1]) the relevant subject matters are often executable, that is, they have a behavior associated with them. Therefore we need to integrate both the structure and behavior of the individual models to realize a working system.

Executable UML (xUML) [6], [7], an approach within the OMG Model-Driven Architecture initiative, provides an approach in line with the ideas outlined above: it structures a platform-independent model of an application into a highly decoupled system of domains, representing separate subject matters, and bridges, providing the glue between separate domains. At a lower level of abstraction structure and behavior of domains are described using UML and an imperative action language. While the high-level partitioning of subject matters into domains offers the potential for large-scale reuse, the particular choice of languages for representing structure and behavior entails at least two drawbacks: because UML does not have a formal semantics, the resulting models are not easily amenable to formal analysis; furthermore the imperative nature of the action language clashes with the declarative nature of the structural models.

In the present paper we strive to use the main advantages of the xUML method - namely the partition of platform-independent models according to subject matter using domains and bridges - while avoiding the disadvantages related to the concrete languages used to specify structural and behavioral aspects. We achieve this by defining an abstract framework built on domains and bridges that is independent of the particular choice of languages used to represent these concepts. We introduce the new notion of a domain hierarchy, which is essentially a collection of domains that are glued together to form a more complex domain. We formally define bridges to be the connectors between separate domains; bridges not only connect structural features of underlying domains but they also link the behaviors of these domains into a coherent whole. We show how to instantiate the abstract framework into a concrete one by choosing a declarative language named EP [3], [5] that has a formal semantics and is executable. We validate our approach by applying it to a small case study dealing with a document management system. For full details, see [4].

II. Domain Hierarchies

We generalize the high-level approach of xUML based on domains and bridges by redefining these notions formally within an abstract framework. We assume that all domains and bridges are expressed within the same *language*. Abstractly, this language contains two kinds of elements: those expressing concepts and those expressing relationships between concepts. We call instances of the former *entities*, and instances of the latter *links*. Each domain or bridge is thus made up of a number of entities and links. While a domain is self-contained, i.e. both the source and target entities of its links are included in the domain itself, a bridge is not. A bridge is specified over a set of domains with at least one external link such that its target entity belongs to some participant domain. Following the definition, the union of a bridge and its participant domains represents a domain, which can thus participate in another level of domain integration. In this way, we obtain the concept of a domain hierarchy.

Definition 1 (Domain Hierarchy)**.** *A domain hierarchy is a directed acyclic graph whose nodes are domains and bridges such that:*

1) *all sink nodes are domains;*
2) *there is a unique source node, which is a domain, called the root of the domain hierarchy;*

3) *each bridge node has as successors all the nodes that represent its domain participants;*
4) *each domain node has either no successors, or it has as successor a single bridge representing a domain that is derived by integrating the bridge with its participants.*

Moreover, in order to support executable modeling of software systems, the language for representing both domains and bridges in the domain hierarchy needs to satisfy some additional requirements: 1) Both structural and behavioral specification are supported. Note that operations alone do not imply executability. Dynamic semantics of the operations needs to be expressed as well in the language; 2) Both the syntax and the semantics of the language must be expressed with a precise mathematical notation; 3) A mechanism similar to inversion of control used in the context of components[2] is offered, which allows external behavior to be weaved into an operation of a domain without it being aware of this.

III. An Example Language and a Case Study

We valid our approach by instantiating the abstract domain hierarchy framework with an example modeling language named EP, and applying the result to a simple document management system case study.

EP is a research language developed by our team with a formal semantics [3], [5]. The essence of EP can be summarized by its two key concepts of *properties* and *events*, where properties express system states and events modify system states. The dynamics of events are specified in terms of *event edges*: an event can modify a property by an *impact edge* to compute a new value of the impacted property; an event can trigger other events via *push edges* and *pull edges*, for which execution then proceeds in a similar way. Moreover, inversion of control is achieved by the combination of pull and push edges: a bridge event owns a pull edge with the target event belonging to some domain. When the target event in the domain is triggered, because of "pulling", the bridge event will also be triggered. Then using another push edge, this bridge event can then trigger further events in another domain, which in total realizes the propagation of execution from one domain to another. Note that neither the pull edges nor the push edges owned by the bridge event pollute the domains that it hooks up to since these edges are not part of these domains.

As a case study, we model a document management system (DMS) using a domain hierarchy instantiated with EP. The domain hierarchy of DMS is sketched in Fig. 1. We start from three domain definitions at the bottom: Gui encapsulates: the graphical user interface facilities; Document the document management services; and Log the logging services. Domains are self-contained (with no outgoing dependencies) and designed to be general and reusable. We then define bridge DocumentGui to customize Gui and bridge DocumentLog to customize Log for the purpose of this particular application. The outgoing links from bridges to domains depict the dependency of the bridges on their participant domains. Finally, a third bridge DMS is specified to glue all the building blocks together to achieve an executable system.

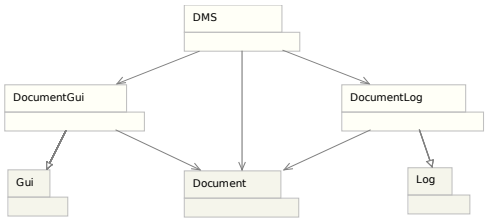

Fig. 1. Document management system: the domain hierarchy.

Note that not only structural bridging such as implementing an interface from a domain with information from another domain, but also execution propagation is realized in these bridges. In Fig. 2, following the control flow from left to right as indicated by the arrows, where solid arrowheads correspond to push edges, and white ones to pull edges, the *event tree* illustrates how execution of OK action in BookEditWindow of Gui domain eventually updates the corresponding edited book with new contents.

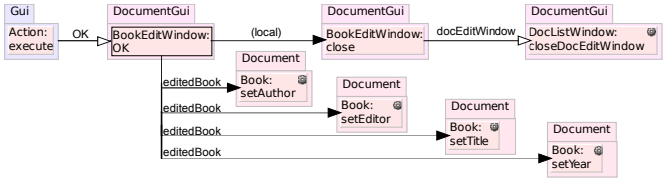

Fig. 2. Event propagation via pull and push edges anchored at bridges.

IV. Conclusion and Future Work

We have carried out a theoretical investigation of some basic concepts - domains, bridges and domain hierarchies - that allow one to view a complex application as a nested collection of domains and bridges. By applying these concepts to a concrete example we have demonstrated that this approach allows one in principle to build abstract executable models of applications from domains. Future work includes carrying out a more realistic case study and defining a methodology for building domain hierarchies.

References

[1] Matthias Bräuer and Henrik Lochmann. Towards semantic integration of multiple domain-specic languages using ontological foundations. In *the Proceedings of ATEM 2007*, 2007.

[2] Martin Fowler. Inversion of control containers and the dependency injection pattern, 2004. http://martinfowler.com/articles/injection.html.

[3] Pierre Kelsen and Qin Ma. A formal definition of the EP language. Technical Report TR-LASSY-08-03, University of Luxembourg, May 2008. http://democles.lassy.uni.lu/documentation/TR_LASSY_08_03.pdf.

[4] Pierre Kelsen and Qin Ma. A language for domain hierarchies with applications to the domain integration problem. Technical Report TR-LASSY-08-05, University of Luxembourg, 2008. http://democles.lassy.uni.lu/documentation/TR_LASSY_08_05.pdf.

[5] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *the Proceedings of MoDELS 2008*, LNCS 5301, pages 690–704, 2008.

[6] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, 2002.

[7] Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.