A Language for Domain Hierarchies with Applications to the Domain Integration Problem

Pierre Kelsen and Qin Ma
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg

TR-LASSY-08-05

Abstract

Executable UML (xUML), an approach within the OMG Model-Driven Architecture initiative, 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 into subject matters 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 this paper we present a new approach that borrows from xUML 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 xUML 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.

1 Introduction

Complex software applications deal with many different subject matters. As an example consider a Web Application for electronic banking. Such an application has a business domain dealing with the business objects relevant for a banking application, a graphical user interface domain representing concepts related to the graphical user interfaces of web applications, a security domain that expresses the security policies, to name just a few.

Ideally a model-driven approach for designing software applications would build on these high-level domain 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 domains 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 domains to realize a working system.

Executable UML (xUML) [11, 16] provides an approach in line with the ideas outlined above: it builds platform-independent models partitioned according to subject matter into domains which are self-contained and independent of other domains; these domains are then integrated into a whole using bridges, which are essentially connectors between domains. The high-level approach of xUML is attractive since these domains have the potential to provide high-level reusable components.

The structure of xUML models is expressed using UML class diagrams while the behavior is given using UML state charts and an action language. The use of UML implies that the resulting models do not have a formal semantics. This limits the possibility of formally analyzing xUML models and thus jeopardizes their use in the context of safety-critical applications. Action languages used for describing part of the behavior in xUML models are of an imperative nature which clashes with the declarative nature of the structural description. Furthermore, while the action semantics have been standardized by the OMG, the actual concrete syntax is not part of the standardization. This brings about the prospect of a plethora of action languages being used for describing systems, further complicating the task of understanding and sharing the underlying 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 declar-

ative language named EP [7, 8] 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.

In this paper we do not consider the problem of mapping or transforming platform-independent model to concrete platforms (such mappings are part of xUML). Realizing such mappings for our approach will be the subject of future work.

This paper is structured as follows: Section 2 builds an abstract framework by giving formal definitions of domains, bridges and domain hierarchies and proving some basic properties of these concepts. In Section 3 we describe the requirements that a language representing domains and bridges must satisfy and we present in Section 4 a declarative executable language, the EP language, as an example of a language that satisfies these requirements. We validate our approach in Section 5 where we build the domain hierarchy for a simple document management system. We discuss related work in Section 6, and finally the concluding remarks and future work in Section 7.

2 From Domains to Domain Hierarchies

As explained in the introduction we may view a software application as being composed of a number of subject matters or domains that need to be composed in order to produce an executable system. We refer to this problem as the **domain integration problem**. In order to tackle this problem we generalize the high-level approach of xUML based on domains and bridges by redefining these notions formally within an abstract framework.

We shall first assume that all domain models are expressed using the same *language*. We shall also assume that this language contains a number of elements that express abstract concepts (e.g, class or property) and elements that express relationships between these concepts (e.g., association or inheritance). We use the term *entity* to denote an instance of an element defined in the language which expresses a concept, and the term *link* to denote an instance of an element that expresses a relationship between concepts. Each domain is thus made up of a number of entities and links. In the following discussion we view a link as an ordered pair of entities. For a link (e_1, e_2) we call e_1 the *source* (entity) of the link and e_2 the *target* (entity) of the link.

To deal with multiple domains, we define a *universe of entities* U, which is essentially a set of all entities of interest. We now embark on setting up the basic definitions dealing with domains and bridges.

Definition 1 (Fragment)**.** *A fragment is a pair $F = (V, R)$ where V is a set of entities (i.e., $V \subseteq U$), and R is a set of links such that $R \subseteq V \times U$, that is, each link in R has a source entity in V but not necessarily a target entity in V.*

Definition 2 (Fragment Union)**.** *The union of two fragments $F_1 = (V_1, R_1)$ and $F_2 = (V_2, R_2)$ is the fragment $F = (V_1 \cup V_2, R_1 \cup R_2)$.*

Definition 3 (External Link)**.** *An external link in a fragment* (V, R) *is a link* $(e_1, e_2) \in R$ *such that* $e_2 \in U - V$*.*

Definition 4 (Domain)**.** *A domain is a fragment with no external links.*

This definition is the formal counterpart of the informal definition used in xUML. It captures the fact that each domain is self-contained.

Theorem 1. *The union of two domains is a domain.*

Proof. Follows immediately from the definitions. □

Definition 5 (Bridge)**.** *A bridge over a set of domains* $\{D_1, \ldots, D_k\}$ *is a fragment with at least one external link such that the target of every external link in this fragment belongs to some* $D_i, 1 \leq i \leq k$*.*

For this definition we have implicitly assumed that bridges are expressed using the same language as domains: they only differ in the existence of external links. In xUML on the other hand, bridges and domains are expressed using two different languages. In our view using a single language for domains and bridges should simplify the design of systems.

Theorem 2. *If* B *is a bridge over domains* $\{D_1, \ldots, D_k\}$*, then the fragment* $B \cup (\bigcup_{i \in \{1 \ldots k\}} D_i)$ *is a domain.*

Proof. By the previous theorem the fragment $\bigcup_{i \in \{1 \ldots k\}} D_i$ is a domain. Since all external links of B have their target in $\bigcup_{i \in \{1 \ldots k\}} D_i$, the claim of the theorem follows. □

Definition 6 (Bridge of Domain)**.** *Let* B *be a bridge over domains* $\{D_1, \ldots, D_k\}$ *and let* D *denote the domain* $B \cup (\bigcup_{i \in \{1 \ldots k\}} D_i)$*. Then we say that* B *is a bridge of domain* D*.*

Definition 7 (Domain Hierarchy)**.** *A domain hierarchy is a directed acyclic graph whose nodes are fragments such that:*

1. *all sink nodes (i.e., nodes with no outgoing edges) are domains;*
2. *there is a unique source node (i.e., a node without incoming edges), which is a domain, called the root of the domain hierarchy;*
3. *each node that is a bridge has as successors all the nodes that represent the domains over which the bridge is defined;*
4. *each node that is a domain has either no successors, or it has as successor a single bridge of this domain.*

Figure 1 shows a meta-model for domain hierarchies; any domain hierarchy can be viewed as an instance of this model. Moreover, we introduce a more compact representation of domain hierarchies called bridge graphs.

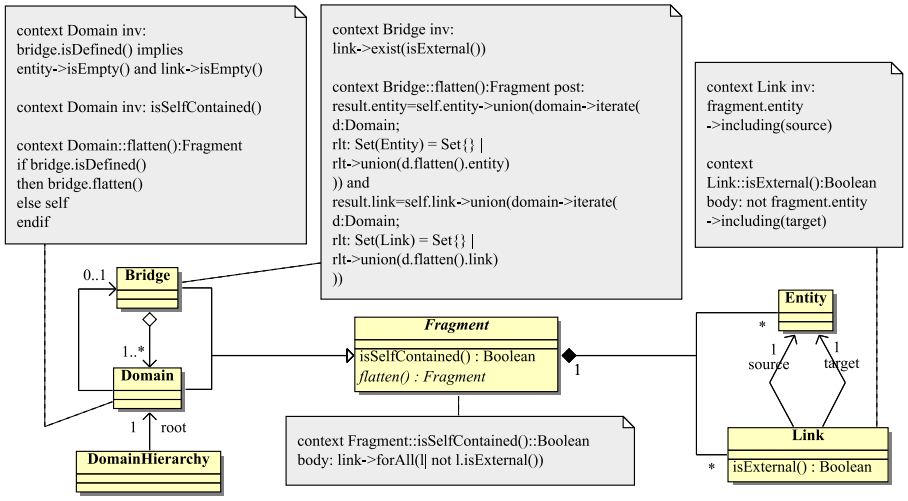


Figure 1: Domain Hierarchy meta-model abstract syntax

Definition 8 (Bridge Graph)**.** *The bridge graph for a domain hierarchy is obtained by collapsing each edge in the domain hierarchy graph that leads from a domain node to a bridge node into the bridge node.*

As an example, Figure 3 shows the bridge graph for the domain hierarchy of the document management system; it is described in more detail in Section 5.

3 Realizing Domain Hierarchies

In the last section we assumed that there exists a single language for representing both domains and bridges in the domain hierarchy. In this section we discuss the requirements that such a language should satisfy in order to allow the description of executable software systems. In the next section we will then present a concrete example language that does indeed fulfill these requirements .

1. Behavioral Modeling: For the system to be executable, the language should allow not only the structure to be defined but also the behavior. That is, one should be able to express the state of a system as well as operations that modify this state. It is not sufficient to have concepts that represent these operations, but the semantics of the operations (i.e., their effect on a system) needs to be expressed as well in the language. If this requirement is satisfied then the domain hierarchy representing the system will indeed be executable.

2. Formal Definition: It is important that structure and behavior are defined formally, i.e. the abstract syntax, the static semantics and the dynamic

semantics must be expressed with a precise mathematical notation. Such a formal definition is essential for tool interoperability and for proving properties of the resulting system.

3. Inversion of Control: A domain may offer an operation that may modify the state but the effect of the operation cannot be fully specified within the domain, for the simple reason that it may affect other domains that this domain is not aware of. As an example consider an *activate()* operation for an abstract button defined in the gui domain: the effect of this operation depends on the concrete application, i.e., this operation could for instance add or remove a document or a member in a library system. We thus need a mechanism that allows external behavior to be weaved into an operation without the domain containing the operation being aware of this. Such a mechanism would be similar to inversion of control used in the context of frameworks or components (see [5]).

4 An Example Language

In this section we present a modeling language with a formal semantics named EP; it was developed by our research team [7, 8] and will be used as an example language to represent domains and bridges.

The essence of EP is summarized as the meta-model in Figure 2. An EP system consists of a set of *models*, each of which consists of a set of *properties* and a set of *events* to cover both structural and behavioral characteristics. *Interfaces* can also be defined; they specify only the type information of properties and/or events, but without implementation details. Interfaces can extend other interfaces, and models can implement interfaces. This will build up a hierarchy of inheritance. Nominal sub-typing is followed. Namely, if A inherits from B, by either extending or implementing it, then for any places where an instance of type B is needed, any instance of type A will also work.

There are two types of properties in the system: *local properties* express the state of the system; *query properties* express side-effect free functions that are evaluated over the state of the system. The state of the system can be modified using events. An event can modify a local property via an *impact edge* carrying an expression that computes the new value of the impacted property. Expressions in EP models are formulated using the OCL expression language [14, 9].

Since an event may affect multiple properties in multiple models we can define event edges that will represent which other events will be executed when the current event is triggered. There are two types of *event edges*: *push edges* and *pull edges*. An event may have a number of such edges associated with itself. Both types of event edges have a target event and a link property, whose value indicates on which instance the target event will be triggered. Pull edges and push edges differ fundamentally in the way in which the flow of control is transferred: a pull edge means that the event that owns this edge will be

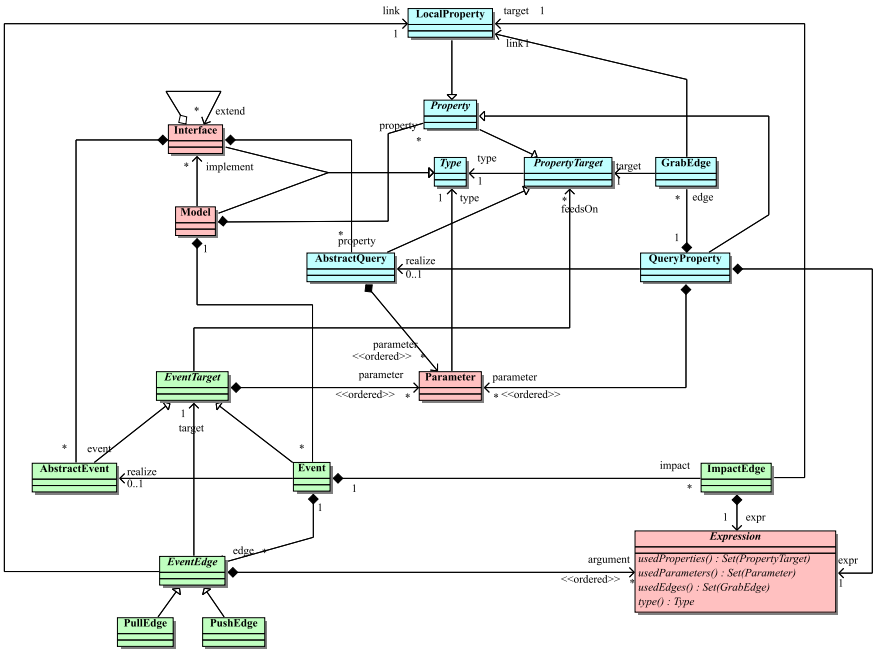
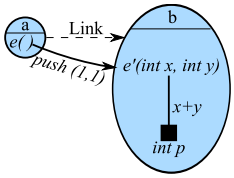


Figure 2: EP meta-model abstract syntax

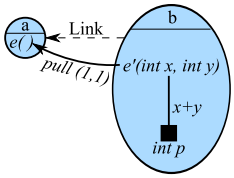
triggered when the target event is triggered. On the other hand a push edge works in the opposite direction: when the event owning this edge is executed it triggers the target event. So for pull edges flow of execution is from the target event to this event (owning the event edge) while for push edges flow of execution is from this event to the target event. Events may have parameters; expressions on the event edges provide the values for the parameters when the system executes.

To really understand the dynamic semantics of events, it helps to look at a concrete example with model instances that exist at run-time.



We consider the situation with two instances "a" and "b". Instance "a" (i.e., the corresponding model) has an event "e" with no parameters, and instance "b" has an event "e'" that takes two integer parameters "x, y". The solid arrow denotes a push edge between events and the dashed arrow labeled "Link" specifies the link of the edge which is basically a local property of "a" that points to "b" as its value. The square-headed arrow between event "e'" and local property "p" on instance "b" denotes an impact edge, whose associated expression is "$x + y$", i.e. the sum of the two parameters of the event. In a nutshell, this example depicts the following event propagation and local property modification: execution of "e"

on "a" would trigger execution of "e'" on "b" with the constant arguments 1 and 1, which would in turn change the value of "p" on instance "b" to the sum of the two arguments of "e'", i.e. 2.



The same scenario can also be captured by replacing the push edge with a pull edge. However, this time, the link between the two instances becomes a local property of instance "b" that points to "a" as its value. We see that in both situations, the execution at runtime will start from "e" on "a" and flow to "e'" on "b". However, with the push edge, it is instance "a" that initiates the propagation as in "a" pushing the execution into "b"; while with the pull edge, it is instance "b" that initiates the propagation as in "b" pulling the execution from "a".

Discussion: We first remark that EP conforms to the view of a language being composed of abstract concepts and relationships between concepts as seen from the domain hierarchy framework in Section 2. More specifically, EP has elements representing concepts (shown as boxes in Figure 2) such as *Model*, *Type*, *Property* and *Event*, as well as elements that represent relationships between concepts (represented by lines in Figure 2). Relationships are either associations or inheritances between concepts - for example the association named *implement* between *Model* and *Interface*, and the inheritance between *Model* and *Type*.

We now verify that EP does indeed satisfy the requirements stated in Section 3.

1. Behavioral Modeling: The behavior of an EP-system is defined via its events. The effect of the event is fully specified by the event edges (both pull and push edges) and impact edges. More precisely, if an event e is triggered, then it will modify the local properties impacted on this instance (on which the event is triggered) and set them to values specified by the expressions attached to the impact edges; it will also trigger those events on those instances that are specified by event edges (see the informal description above). Execution proceeds similarly for the events triggered in this way.

2. Formal Definition: The abstract syntax, the static semantics and the dynamic semantics are defined formally in [7, 8] using two different approaches: EBNF, type systems and operational semantics on one hand, and Alloy on the other hand.

3. Inversion of Control: Inversion of control is achieved using pull edges: the event owning the pull edge resides in the bridge while its target event belongs to some domain. When the target event in the domain is triggered, the bridge event will then also be triggered. Using push edges this bridge event can then trigger further events in other domains (or bridges). We note that neither the pull edges nor the push edges owned by events in the bridge pollute the domains that they hook up to since these edges are not part of these domains.

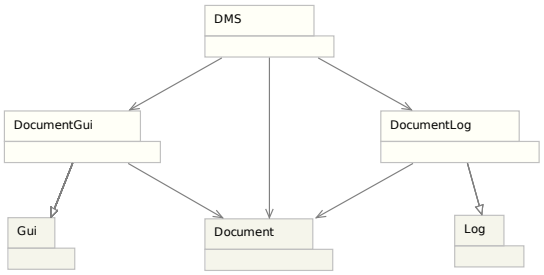


Figure 3: Document management system: the bridge graph of the domain hierarchy.

5 Case Study

In this section we model a concrete example system - a simple document management system (DMS) - using a domain hierarchy based on the EP language. We first describe this example system in more detail.

5.1 The Document Management System

The document management system provides services for managing a collection of documents, including: specifying types of documents maintained in the system, e.g. books and videos; adding new documents into the system or deleting documents; viewing the contents of existing documents; and editing the contents of documents in the system. For usability, all such services should be accessible via a graphical user interface. Moreover, any operation on the system should be properly logged (for the sake of future bug fixing or statistical studies).

5.2 The Domain hierarchy

The domain hierarchy of the document management system is presented in Figure 3 in terms of a bridge graph. We start with three domain definitions at the bottom of the graph: Gui, Document, and Log, each of which respectively encapsulates: the graphical user interface facilities, the document management services, and the logging services. Domains are self-contained (with no outgoing dependencies) and designed to be general and reusable. As a consequence, to fit in a particular application context, bridges are needed to customize general subject matters. In this example, we define bridge DocumentGui to customize Gui and bridge DocumentLog to customize Log. According to Theorem 2, a bridge together with the set of domains over which it is defined is a domain, and the outgoing links from bridges to domains depict the dependency of the bridges on the domains. Finally, a third bridge DMS is specified to glue all the building blocks together to achieve an executable system.

5.3 The Domains

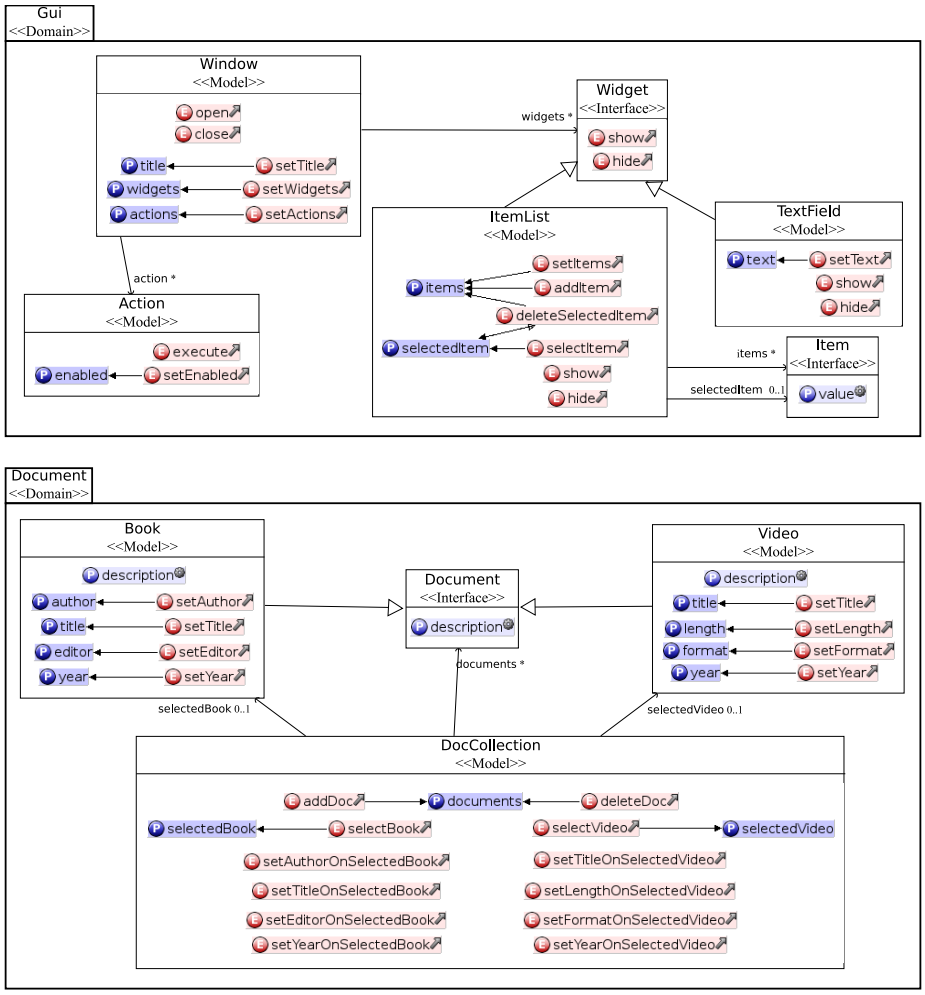


Figure 4: Domain Gui and domain Document for the document management system.

Let us consider the Gui and Document domains in more detail. Depicted by the diagrams in Figure 4, the Gui domain (represented by the folder-shaped rectangle on top) encapsulates concepts typically found in a graphical user interface. These include windows, actions, text fields etc., all implemented as EP models (represented by two-part boxes) with properties (represented by blue icons starting with "P"). Moreover, related behaviors on these concepts such as opening a window, executing an action, or setting the text of a text field, are also implemented as events of the models (represented by red icons starting with

"E"). Similarly, the Document domain characterizes concepts and behaviors relevant for document management. We remark that abstract entities and links (as defined in Section 2) are all concretized by instances of EP constructs (as defined in Figure 2). For lack of space, we omit the details of the Log domain, which is defined in a similar manner.

5.4 The Bridges

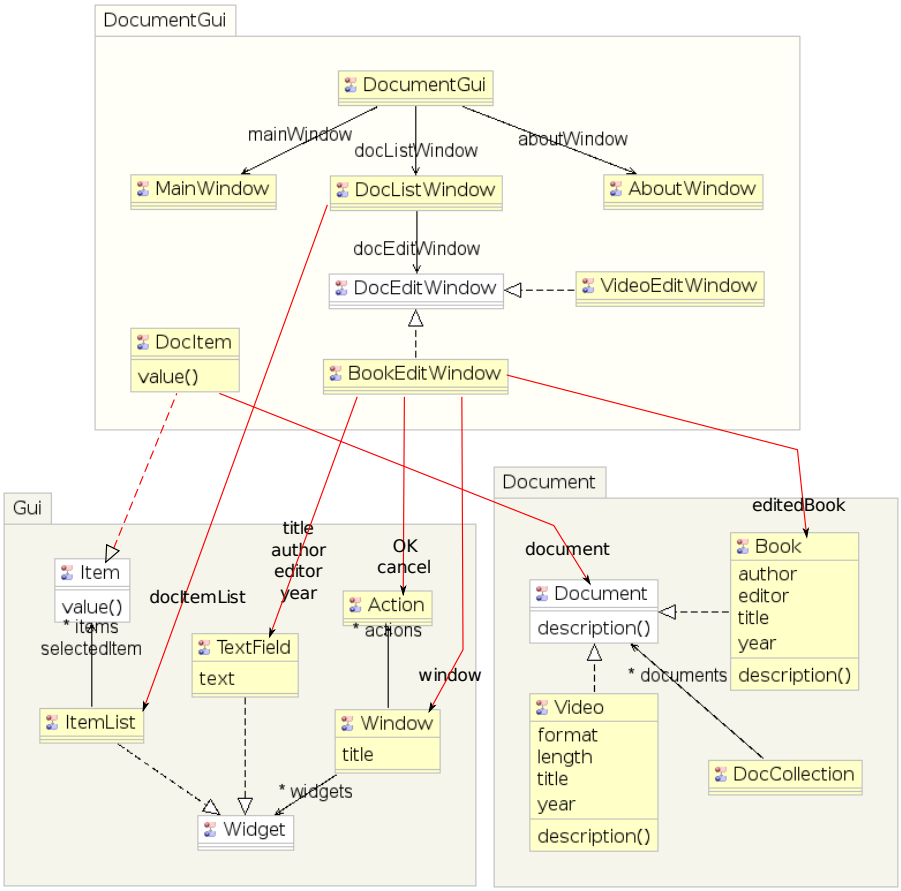


Figure 5: Bridge DocumentGui and its participating domains for the document management system.

As an example of a bridge specification, let us take a closer look at the bridge DocumentGui in Figure 5. It is built upon two domains: Gui and Document, mainly for the purpose of customizing the general Gui domain into an appropriate graphical user interface for the document management system. As

discussed above, a bridge should hook into both the structural and behavioral elements of the underlying domains. In the following, we elaborate on the DocumentGui bridge from both a structural and behavioral point of view. The notation "Domain::Element" gives the fully qualified reference to the "Element" defined in "Domain".

Structural bridging: As shown in Figure 5, the graphical user interface of the document system consists of a set of "windows": MainWindow, DocListWindow, BookEditWindow, etc. Each such window, implemented as an EP model in the bridge, is designed as a selected aggregation of the facilities provided by the underlying domains. Take BookEditWindow for example. All window relevant features, such as opening or closing, are delegated to a property named window that points to Window in the Gui domain as its type. It also possesses a property editedBook of type Document::Book to hold the book that is currently being edited; four Gui::TextField typed properties, namely title, author, editor, and year to represent the content of the edited book; and two actions OK and cancel both realized as an instance of Gui::Action.

In addition, bridges are also the place where abstract interfaces in domains are implemented. DocItem is such an example. It implements the Item interface from domain Gui. The purpose is evident: items listed in the DocListWindow are not just any items but those adapted to also contain information of the corresponding documents represented by the items in the graphical interface. As a consequence, each DocItem has a property document of type Document::Document.

All external links are presented as arrows crossing boxes in the diagrams. For this particular example, links are reified into two types of relations: inheritance (represented by dashed arrows) and typing (shown as solid arrows). Note that all arrows go from bridges to domains, but never in the other direction.

Behavioral propagation: Operations on the system are triggered by executing actions in the windows. For example, if the OK action in BookEditWindow is executed, the corresponding edited book will be updated with the new contents given by the four text fields, and the window itself will be closed.

The propagation of event execution from one domain (e.g. Gui) to another (e.g. Document) is best illustrated with the help of an *event tree*. Figure 6 shows the event tree corresponding to the OK action on BookEditWindow. Control flow propagation is from left to right, following the arrows. Solid arrowheads correspond to push edges, and white arrowheads represent pull edges. Note that because the direction of the arrows indicates flow of control, pull edges are actually drawn in the opposite direction (compared to the last figure in Section 4), namely, from the target event to the owner event. Events that impact properties, and thus modify the system state, are marked with a cog wheel. Looking at the root and leaves of the event tree is sufficient to determine the behavior of triggering the OK action: it results in state changes in the Document domain, namely setting the author, title, editor, and year fields of the edited book. Meanwhile, the BookEditWindow is closed which amounts to setting the docEditWindow property of DocListWindow to null. Note that such behavior propagation does not pollute the participating domains because all the

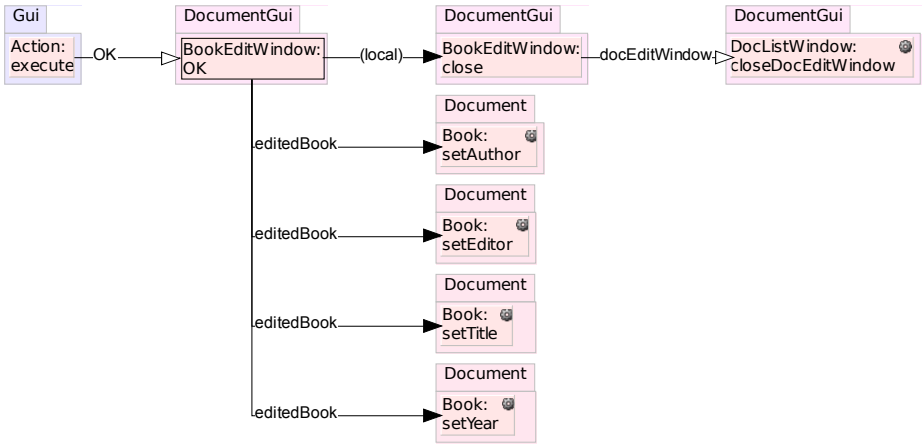


Figure 6: An example of event propagation via pull and push edges anchored at bridges.

necessary information for realizing relaying is anchored only at the bridges.

6 Related Work

We view research in three areas most relevant to our work on the domain integration problem, namely, aspect-oriented modeling, component-based modeling, and meta-modeling.

Aspect-oriented modeling (AOM): The use of aspect-oriented techniques in modeling allows system designers to encapsulate cross-cutting concerns into *aspect models*, separated from the *base model* of a system, and build the complete system model by composing both the aspect and base models together. Existing AOM approaches can be considered as either *specific*, focusing only on the composition of models specified in one specific modeling notation such as class diagrams [3] or sequence diagrams [10], or *generic*, being independent of the modeling language that is used to specify the aspect and base models (but requiring the language to have a well-defined meta-model).

Our work is more related to generic AOM approaches because the abstract domain hierarchy framework does not prescribe a particular language. We will now discuss three generic tool-based approaches, namely Kompose, MATA and Geko.

Kompose [4] provides a generic framework for automatic model composition based on composition directives defined by Reddy et al. [17]. Kompose only deals with structural composition and thus is not as expressive as our approach. By contrast, the MATA tool [18] and the Geko weaver [12] support behavioral compositions as well. MATA and Geko both rely on graph pattern matching for specifying aspect compositions. This adds an additional level of complexity

compared to our approach since we express the domains and their composition using a single language.

Component-based modeling: Component-based techniques build systems by glueing prefabricated components. This is usually done at the code level as opposed to our model-driven approach. A few component-based approaches are used at the modeling level. For instance a UML component [15] encapsulates a part of a system into a module with well-defined interfaces. Components are then wired together through their interfaces by *connectors*. Generally UML components are not fully executable because only certain aspects of their behavior are expressed (using sequence diagrams or state charts for instance).

Meta-modeling : Several languages have been defined in which we could express domains: prominent examples are MOF [13], Ecore [2] and KM3 [6]. Unfortunately few of these languages enable the modeling of behavior. Among those is Kermeta, a meta-language that is the result of weaving an action language into the EMOF meta-language. Kermeta differs from the EP-language that we propose in several respects: (1) it lacks a full formal semantics; (2) Inversion of control is not directly supported; (3) behavioral modeling is based on an imperative action language that contrasts the more declarative nature of our EP-language.

7 Conclusion

The domain integration problem is at the heart of model-driven software development because it offers the promise of making domain models reusable assets on which concrete applications can be built. 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. Of course a more realistic case study is needed to further validate our approach.

Several open questions remain for future work.

- What language should be used for modeling domains and bridges? Although we have advocated the use of the EP language in this paper, it is by no means clear at this point whether this is really the best choice. In particular the fact that EP is not aligned with more common (meta-)languages such as MOF or Ecore may be seen as a drawback. In this context we plan to investigate whether EP can be aligned with some commonly used (meta-)language, at least as far as structural features are concerned.

- How does one build a domain hierarchy? Although we have applied the notion of domain hierarchies to a concrete case study, we are still far from having a systematic method for building such a hierarchy. Such a method would require the identification of the relevant basic domains and

the design of suitable bridge models. For this we probably would need to first do a more thorough investigation of the different types of bridges that may be needed to build complex applications.

- How do we map the abstract system to the concrete platform? In model-driven development the mapping to the concrete platform is typically achieved using model transformations. Based on the findings of this paper it would be worthwhile to study whether domain hierarchies can be applied to this mapping problem as well. Indeed one could view a particular platform as a domain. By building a domain model for the platform using the common language, one may be able to map behavior at the abstract level to the platform using the bridging mechanisms described in this paper.

References

[1] M. Bräuer and H. Lochmann. Towards semantic integration of multiple domain-specic languages using ontological foundations. In *the Proceedings of 4th International Workshop on Language Engineering (ATEM 2007)*, Oct 2007.

[2] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[3] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[4] F. Fleurey, B. Baudry, R. B. France, and S. Ghosh. A generic approach for automatic model composition. In *the Proceedings of 11th Aspect-Oriented Modeling Workshop*, volume LNCS 5002, pages 7–15, 2007.

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

[6] F. Jouault and J. Bézivin. KM3: a dsl for metamodel specification. In *the Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185, 2006.

[7] P. Kelsen and Q. Ma. A formal definition of the EP language. Technical Report TR-LASSY-08-03, Laboratory for Advanced Software Systems, University of Luxembourg, May 2008. http://democles.lassy.uni.lu/documentation/TR_LASSY_08_03.pdf.

[8] P. Kelsen and Q. Ma. A lightweight approach for defining the formal semantics of a modeling language. In *the Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume LNCS 5301, pages 690–704, 2008.

[9] P. Kelsen, E. Pulvermueller, and C. Glodt. Specifying executable platform-independent models using OCL. In *the Proceedings of Workshop Ocl4All: Modelling Systems with OCL*, ECEASST 2008(9), 2007.

[10] J. Klein, F. Fleurey, and J.-M. Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on Aspect-Oriented Software Development*, LNCS 4620:167–199, 2007.

[11] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By Ivar Jacoboson.

[12] B. Morin, J. Klein, O. Barais, and J.-M. Jézéquel. A generic weaver for supporting product lines. In *the Proceedings of Early Aspects Workshop at ICSE 2008*, 2008.

[13] OMG. Meta Object Facility (MOF) core specification version 2.0, 2006. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[14] OMG. Object Constraint Language version 2.0, May 2006.

[15] OMG. Unified modeling language superstructure specification, version 2.1.2, November 2007.

[16] C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, New York, NY, USA, 2004.

[17] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development I*, pages 75–105, 2006.

[18] J. Whittle and P. K. Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In *the Proceedings of 11th Aspect-Oriented Modeling Workshop*, volume LNCS 5002, pages 16–27, 2007.