Combining Models with Code: a Tale of Two Languages

Qin Ma[‡], Sam Schmit[*], Christian Glodt[†] and Pierre Kelsen[†]
[*]Amazon EU Services, 5 Rue Plaetis, 2338 Luxembourg
Email: sam@schm.it
[†]University of Luxembourg, 6, Rue Coudenhove-Kalergi, 1359 Luxembourg
Email: firstname.lastname@uni.lu
[‡]Public Research Centre Henri Tudor, 29 Avenue J-F Kennedy, 1855 Luxembourg
Email: qin.ma@tudor.lu

Abstract—**In the pure model-driven view of software engineering, models are the sole artifacts to be created and maintained and executable source code is entirely generated from the models. However, due to the variety of modern platforms and the complexity of capturing them correctly in models, this vision has not yet been fully realized. In this paper, we propose an approach that allows combining high-level models with low-level code into an executable system. The approach is based on two modeling languages, one presenting a common abstraction of modeling and programming languages, and the other allowing to express the bridge between the model and code. We illustrate our approach using a running example of an invoicing system for which the business logic requirements are captured by an executable model and the requirements on the graphical user interface are directly mocked up using a GUI designer tool that generates Java code.**

I. Introduction

In the pure model-driven view of software engineering [1] a system is represented by a set of models from which the full code can be generated. However, due to the variety of modern platforms and the complexity of capturing them in models, this vision has not yet been fully realized. As a consequence, researchers start to explore alternative solutions where models and code co-exist to represent a system [2]. A main concern of such solutions is the techniques to integrate models with code to obtain a fully functioning system.

We explore in this paper an approach to this integration problem based on two new modeling languages. More specifically, the following particular scenario will be considered where the business logic requirements of a system are captured abstractly by a high-level executable model and the requirements on the graphical user interface are directly mocked up using a visual GUI designer tool that generates Java code. Our approach results in a set of models from which code can be generated which, together with the GUI code, constitutes a fully functioning system. Development and maintenance of the models and code can be realized in a globalized setting, with members of a geographically distributed team each being in charge of a respective part and collaboratively contributing to the construction of the overall system.

Model Driven Architecture (MDA) shares a common vision with us by advocating the separation of Platform Independent Models (PIMs) that document business functionality of a system from technology-specific code [3]. Following MDA, such a vision is achieved by augmenting a PIM with a Platform Model (PM) and mapping them to a Platform Specific Model (PSM). However, the practicability of MDA is hindered by the complexity of real platforms which makes it rather difficult to ensure the correctness of PMs. Moreover, the MDA approach also differs from ours in the sense that it attempts to model the entire system in a platform-independent manner while we only capture real platform independent information (e.g., the business logic) in models while leaving platform dependent information (e.g., the graphical user interface) in code and combine them together to reach an entire system.

Integrating artifacts from heterogeneous sources is an active field of research. Various techniques have been proposed to integrate heterogeneous artifacts, being software components (e.g., [4], [5]), web services (e.g., [6]), enterprise applications (e.g., [7]), software models (e.g. [8], [9], [10]), and languages (e.g., [11], [12], [13]), just to name a few. The combined set of integration techniques available in the literature covers different levels of abstraction. However, a single technique that bridges artifacts from two different levels of abstraction, i.e., models and code in our case, is still - to the best of our knowledge - under-researched.

The rest of the paper is organized as follows: in Sect. II we present the overall approach; in Sect. III we introduce our bridging mechanism and we illustrate it using a running example in Sect. IV; finally, we discuss the advantages and limitations of our work and present ideas for future work in Sect. V. The work presented in this paper is based on [14].

II. Approach Overview

Figure 1 shows the vision of our approach. Suppose a globalized setting of the development and maintenance team. A member (e.g., in China) gathers requirements on the business logic of the target system and captures them in the form of a model. Another member (e.g., in Australia) elicits requirements on the graphical user interface of the target system and designs them in a GUI designer which generates GUI code. Wrapper models are automatically generated on both sites to represent abstractions of the business logic model and the GUI code respectively. These wrapper models provide input to a third member (located e.g., in Germany) who keeps track of the requirements on the connection behavior between the business logic and the user interface, and specifies them in a bridge model. No hand-written glue code is necessary in

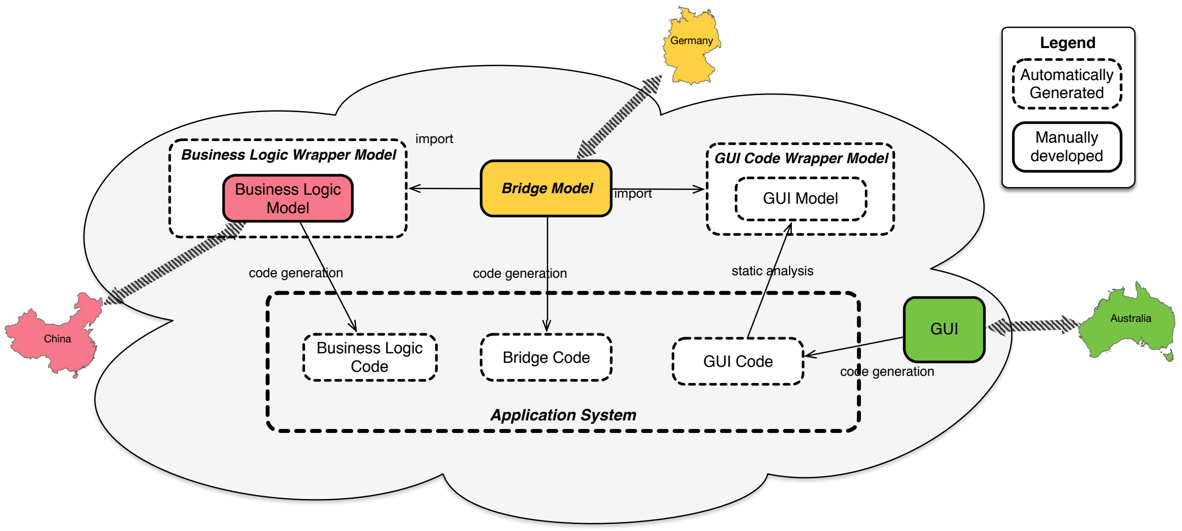


Fig. 1. Approach: overview

our approach in order to obtain a fully functioning application system. Instead, the business logic model and the bridge model will have the property that we can generate code from them which, together with the existing GUI code, constitutes a fully functioning software system.

The main challenge to realize our approach was the elaboration of the Wrapper and Bridge Languages in which the wrapper models and the bridge model are expressed. In essence, the Wrapper Language presents a common abstraction of object-oriented languages that support the event-driven paradigm, e.g., Java, UML, by generalizing notions that are relevant for integration and exposing them in an uniform interface. The uniform interface is then used by the Bridge Language to create bridge models between wrapper models. A bridge model specifies a set of connections, each of which links a source to a set of targets, to capture event propagation from one side (e.g., the business logic) to the other side (e.g., the GUI) and vice-versa.

The application of our approach requires adapting the candidate languages to be integrated. Briefly, it amounts to simply identifying concepts in the candidate languages which are relevant for expressing the integration bridges and categorizing them according to the notions exposed in the uniform interface. The adaptation is done at the language level and applies to all the artifacts written in the language.

We present our approach in more detail in the next sections using a running example. In this example, an executable modeling language called EP [15], [16], [17] is used to represent the business logic of the software system, and the graphical user interface of the system is directly prototyped in a GUI editor [1] which generates Java code. We illustrate how the two languages, i.e., EP and Java, are adapted using the Wrapper Language and how connections can be specified in the Bridge Language to integrate the business logic model and the GUI Java code.

[1]https://www.eclipse.org/windowbuilder/

III. The Wrapper Language and the Bridge Language

The realization of our approach relies on the elaboration of two new modeling languages: the Wrapper Language provides an uniform abstraction of source languages (and subsequently artifacts expressed in them) to be integrated, and the Bridge Language works with such an uniform abstraction and defines integration connections. Figure 2 shows the metamodels of the two languages.

We assume that the source languages used to express the models and code are object-oriented, support the event-driven paradigm, and follow the command-query separation principle. Why? On one hand, object-oriented modeling and programming are dominant in the world of software development nowadays. On the other hand, event-driven architecture is a prevalent integration pattern, in which integrated artifacts interact by announcing and responding to occurrences of events. Such a loose integration style is an excellent fit for software engineering in a distributed and globalized context where artifacts, developed and maintained at different sites, should have as little knowledge of and dependency on one another as possible to foster scalability, flexibility, and adaptability. Finally, the command-query separation (CQS) principle coined by Bertrand Meyer in one of the most influential OO books "Object Oriented Software Construction" [18] advocates the division of methods into two categories: queries that return a result and do not change the state of the system (i.e., side-effect free); and commands that mutate the state of a system but do not return a value. Such a clean separation simplifies software systems hence adds another layer of support for scalability, flexibility and adaptability.

Starting from this assumption, we identify for the Wrapper Language (as depicted in the upper part of Figure 2) the following concepts to be included in the uniform abstraction: Class, Property, Operation, and Event. A class is a template for creating objects. Three kinds of members of a class are relevant: properties (being either data fields or queries) allow to navigate the object graph of a system; operations describe

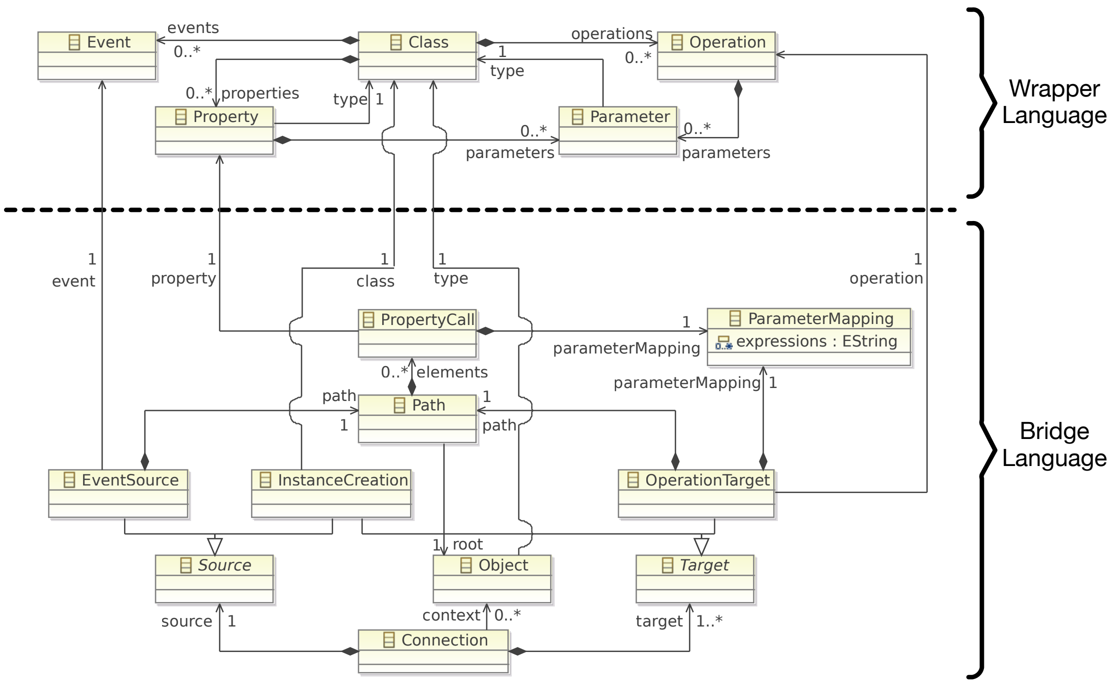


Fig. 2. The Wrapper Language (upper) and the Bridge Language (lower)

behavior of an object of the class that modifies the state of a system; and events signal the occurrence of something to an object of the class.

Events occurring on one side of the integration can be monitored and reacted to by actions on the other side of the integration to effect state change. Such a propagation pattern is supported by the Bridge Language (as depicted in the lower part of Figure 2) in the form of a Connection that monitors the occurrence of a *Source* and triggers several *Target*s in response. A connection is defined in the context of the system state to which it applies. In object-oriented systems, such a state is often captured by an object graph that states how many objects exist and how they are linked. We let connections maintain access to all the root objects in the graph. Two types of sources (given as two subclasses of *Source*) are considered: EventSource denotes events occurring on objects, e.g., an operation is executed on an object or the value of a property of an object is updated; and InstanceCreation denotes creation of new objects of a class. Similarly, we also consider two types of targets: apart from InstanceCreation, the second subclass OperationTarget denotes operation executions on objects to effect state change. Note that for EventSource (resp. OperationTarget), in addition to the identified Event (resp. the Operation), it is also required to designate the object on which the event occurs (resp. on which the operation is executed). Such an object is located by a Path starting from a root object, followed by a sequence of PropertyCalls.

Two types of ends on each side together give rise to four types of connections: (1) Event-Operation Connection are useful to capture scenarios such as a button-click on the GUI side triggering the change-of-state of an object on the business logic side; (2) Instance-Operation Connection are useful to capture scenarios such as the creation of a new object on the business logic side triggering the update of a list on the GUI side to display a new entry corresponding to the new object; (3) Event-Instance Connections are useful to capture scenarios such as the value-update of a property of an object on the business logic side triggering the creation of a new message dialog to be shown on the GUI side; and (4) Instance-Instance Connections are useful to capture scenarios such as the creation of the root business logic object upon system initialization triggering the creation of the main window on the GUI side.

IV. Illustrative Example: an Invoicing System

We illustrate our approach by applying it to the case study of an invoicing system proposed by Henri Habrias. We include in the following an excerpt of its informal description. Formal specifications of this case study can be found in [19].

The subject is to invoice orders. To invoice is to change the state of an order (to change it from the state "pending" to "invoiced"). On an order, we have one and only one reference to an ordered product of a certain quantity. The quantity can be different from other orders. The same reference can be ordered on several different orders. The state of the order will be changed into "invoiced" if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.

The application of our approach involves the following four steps: (1) adaptation of source languages; (2) generation of wrapper models; (3) specification of bridge model; and (4) code generation. We demonstrate these steps in the following sections.

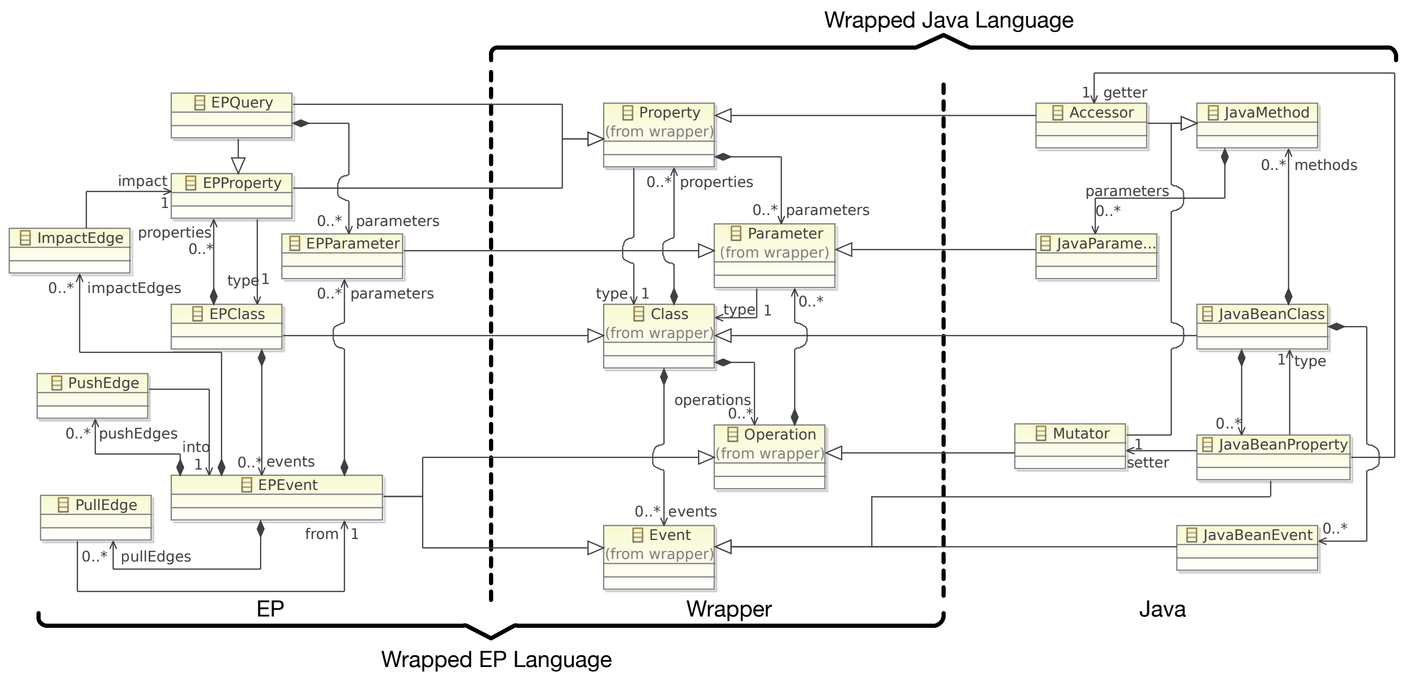


Fig. 3. Wrapped EP Language (left) and Wrapped Java Language (right)

A. Step 1: Source Language Adaptation

This step entails wrapping the two source languages to reveal a uniform appearance that is aligned with the abstraction captured in the Wrapper Language. Figure 3 summarizes the adaptation of both the two languages.

The first source language is EP (as depicted on the left side of Figure 3), which is used to model the business logic of the invoicing system. EP is object-oriented (where objects are created from EPClasses), supports the event-driven paradigm (with the notion of EPEvent), and follows the CQS principle (where an EPQuery is side-effect free, and an EPEvent modifies the values of an EPProperty via its ImpactEdges). Moreover, it offers many other advantages. Firstly, EP allows to express both the structure and behavior of a system. Secondly, EP has a small size compared to other executable modeling languages such as fUML [20]. Thirdly, a code generator exists for EP to generate Java code from EP models, which, together with the code generated from the wrapper and bridge models in Step 4, and the GUI code generated by WindowBuilder, constitutes the complete collection of source code of the invoicing system.

Thanks to the proper alignment to the Wrapper language, the adaptation of EP is straightforward. Without surprise, an EPClass is a *Class*; both an EPProperty and an EPQuery can act as a Property to retrieve objects from an object graph; and finally, an EPEvent plays the dual roles as an Operation and an Event.

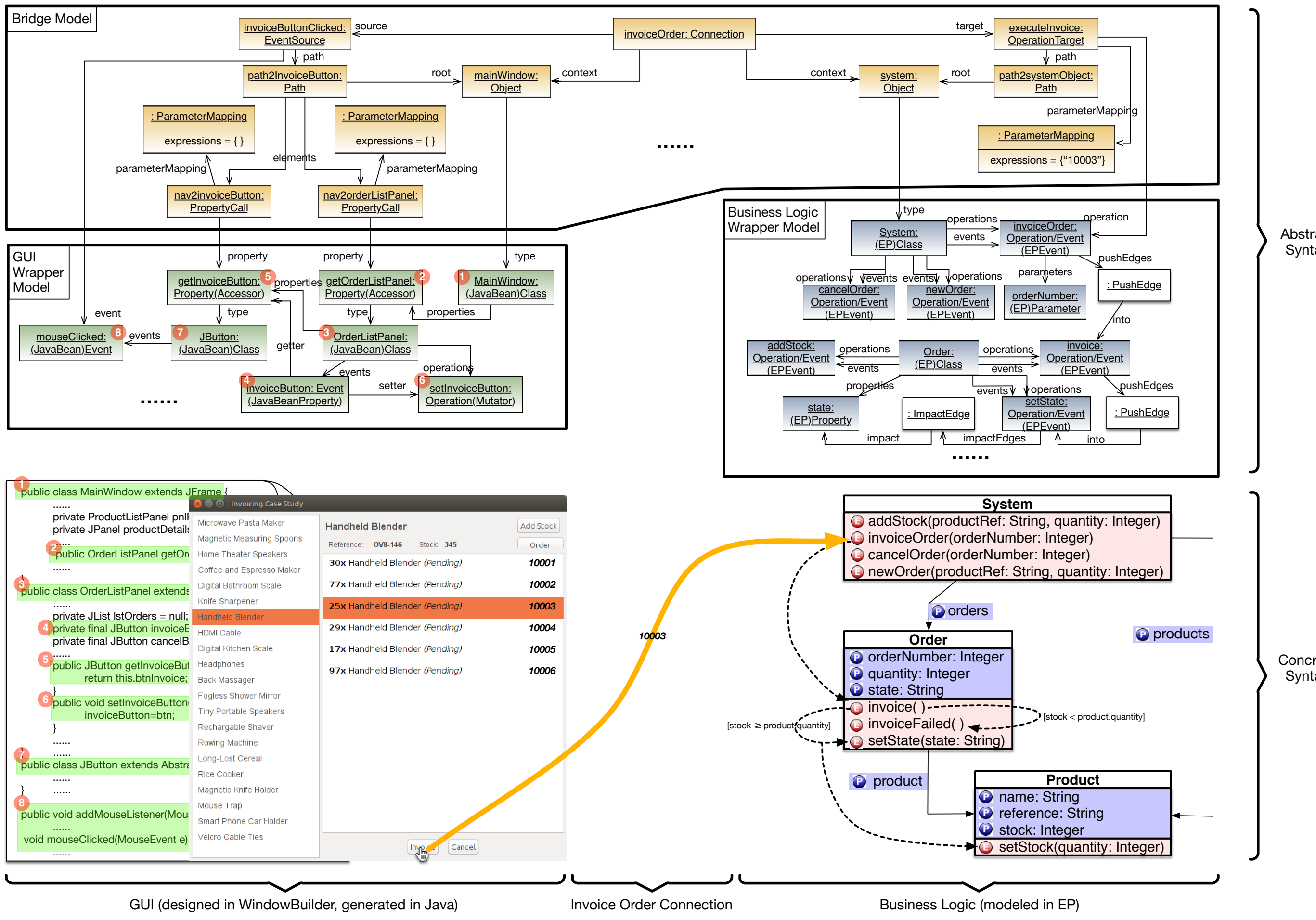
The second source language is Java, or more precisely the part of Java that follows the JavaBean convention [21] (as depicted on the right side of Figure 3). The GUI designer generates GUI code in this format. Java is object-oriented and through the JavaBean convention, it also supports event-driven programming. However, the QCS principle is not natively followed by Java. Analysis techniques such as [22] need to be exploited to distinguish query methods (Accessor) from state changing ones (Mutator). Moreover, according to the JavaBean convention, all fields should be associated with a getter and a setter method.

The adaptation of Java is also intuitive. A JavaBeanClass is a *Class*; a JavaBeanEvent is an Event; a JavaBeanProperty is also an event because following the JavaBean convention, a JavaBean property fires a "PropertyChangeEvent" when its value changes; an Accessor acts as a kind of query Property; and finally, a Mutator denotes an Operation whose execution effects state change. Note that there is no need to let JavaBeanProperty be a subclass of Property any more because of the existence of an associated getter method of the property, following the JavaBean convention.

B. Step 2: Wrapper Models

Figure 4 shows how the invoicing system should be realized by applying our approach. The business logic is modeled in EP (bottom right). The dashed arrows connecting EP events visualizes how events are propagated through an EP model. Event propagation can be guarded with firing conditions. For example, the occurrence of the invoice event of an order either triggers simultaneously the setState event of the order and the setStock event of the product being ordered if the stock is sufficient, or triggers the invoiceFailed event of the order otherwise.

The wrapper model on the business logic side (middle right of Figure 4) is simply the EP model itself, but with all the connectable instances identified. An instance is connectable if it can be type converted to an instance of a concept of the Wrapper Language, following the subclass relations specified in Figure 3. The set of connectable instances constitutes an



```
1 public class MainWindow extends JFrame {
      ......
      private ProductListPanel pnlP
      private JPanel productDetails
      ......
  2   public OrderListPanel getOr
      ......
  }
3 public class OrderListPanel extends
      ......
      private JList lstOrders = null;
  4   private final JButton invoiceB
      private final JButton cancelB
      ......
  5   public JButton getInvoiceBut
          return this.btnInvoice;
      }
  6   public void setInvoiceButton
          invoiceButton=btn;
      }
      ......
      ......
7 public class JButton extends Abstra
      ......
  }   ......
8 public void addMouseListener(Mou
      ......
  void mouseClicked(MouseEvent e)
      ......
```

Microwave Pasta Maker	
Magnetic Measuring Spoons	
Home Theater Speakers	
Coffee and Espresso Maker	
Digital Bathroom Scale	
Knife Sharpener	
Handheld Blender	
HDMI Cable	
Digital Kitchen Scale	
Headphones	
Back Massager	
Fogless Shower Mirror	
Tiny Portable Speakers	
Rechargable Shaver	
Rowing Machine	
Long-Lost Cereal	
Rice Cooker	
Magnetic Knife Holder	
Mouse Trap	
Smart Phone Car Holder	
Velcro Cable Ties	

Handheld Blender

Reference: OV8-146 Stock: 345

30x Handheld Blender *(Pending)*	*10001*
77x Handheld Blender *(Pending)*	*10002*
25x Handheld Blender *(Pending)*	*10003*
29x Handheld Blender *(Pending)*	*10004*
17x Handheld Blender *(Pending)*	*10005*
97x Handheld Blender *(Pending)*	*10006*

Fig. 4. Apply our approach to the Invoicing System, one connection illustrated.

interface that the EP model exposes to the connections to be specified in the bridge model (in Step 3). On the contrary, instances that are not connectable (with white background) are not accessible.

The GUI is designed in WindowBuilder and generated in Java (bottom left of Figure 4). The wrapper model on the GUI side (middle left) needs a bit of extra work, because the source is code. Firstly, a static analysis is performed on the Java code to detect instances of classes characterized in the Java metamodel (right side of Figure 3) and to construct a corresponding Java model. Briefly, for each class in the code, the analyzer creates a JavaBeanClass instance in the model; for each field, a JavaBeanProperty instance; for each method, either a Mutator or an Accessor instance depending on the nature of the method; and finally, for each pair of an event listener and a handler method, a JavaBeanEvent instance. We mark an instance of the generated Java model and the corresponding code snippet with the same number in Figure 4 to demonstrate the correlation. Secondly, the generated Java model is wrapped up to expose all the connectable instances in it, similarly to the EP model.

C. Step 3: Bridge Model

The business logic side and the GUI side are now ready to be integrated. For the case study, the bridge model consists of 15 connections [14]. For lack of space, we here only present one of the connections in detail, which implements the function of invoicing an order. This connection listens to the clicking of the "invoice" button on the GUI side and reacts accordingly by changing the state of the currently selected order on the business logic side. The modeler works at the concrete syntax level. With the tool support, it simply amounts to choosing the source from the GUI side and connecting it to the target from the business logic side. Meanwhile, the number of the order to be invoiced (which is the one that is currently selected in GUI) is extracted and passed on as a parameter to trigger the target operation. The corresponding bridge model in abstract syntax format (top of Figure 4) is generated by the tool. It interfaces with the two wrapper models generated in the previous step. The source end designates the mouseClicked event of the "invoice" button (located by the path mainWindow. orderListPanel.invoiceButton), and the target end designates executing the invoiceOrder EP event (which is an instance of Operation) of the system object.

D. Step 4: Code Generation

Three code generators are involved (see diagram in Figure 1): the first one comes from EP and generates code from the business logic model; the second one comes from WindowBuilder and generates code from the GUI designed in

WindowBuilder; the third one is implemented in this work and generates code from the bridge model (which imports the two wrapper models). Putting the generated code from the three sources together constitutes the invoicing system.

Both the EP code generator and the WindowBuilder code generator take Java as the target platform. As a consequence, our generator also produces Java code. More specifically, the target platform of the third code generator is Java plus aspect-oriented programming (AOP) support [2], due to the problem that Java does not natively create event notifications for object creation. Therefore, to implement the second type of connection sources, i.e., InstanceCreation defined in the Bridge Language in Figure 2, we exploit AOP to introduce the missing notification behavior. For all classes that offer instance creation as a source of a connection in the bridge model, aspect code is also generated together with the code generated for the class. The aspect intercepts object creation of the corresponding class and inject advice code to trigger the targets of all the connections that have instance creation of this class as source, following a dispatcher pattern.

V. Conclusion and Future Work

Interest in Model Driven Software Development (MDSD) has been rising over the last few years, and its peak is not reached yet. Using models to create software systems allows to raise the level of abstraction. We propose a mechanism in which a bridge is created between abstract models and concrete code. We find such an approach to have advantages over a purely model-driven scenario because platform dependent information (e.g., graphical user interfaces) is often better captured using platform specific notations than abstract models. With the help of platform specific tools such as a visual GUI designer and code generators, our approach requires no hand-written code during the development of a software system. Such a result is nicely aligned with the coding-free vision of MDSD and meanwhile overcomes the inability of abstract models when confronted with platform specificities. Moreover, our approach also surpasses hand-coding solutions especially when multiple platforms are targeted.

In its current state, we see the following directions to further improve our work. Firstly, tooling needs to improve to support the entire vision illustrated in Figure 4. As a proof of concept, it suffices to simply rely on the editors generated in the Eclipse Modeling Framework from the Ecore models of the Wrapper and the Bridge languages. However, a well designed concrete syntax and tool support will further boost the applicability of our approach and maximize its benefit. Secondly, a more thorough validation needs to be done using additional case studies to cover different combinations of source modeling and programming languages apart from EP and Java. Example of candidate languages include UML, C#, C++, etc. Thirdly, the Wrapper and Bridge languages need formal semantics to enable mathematical proofs of properties of our approach such as the correctness of the code generator.

References

[1] A. Brown, J. Conallen, and D. Tropeano, "Introduction: Models, modeling, and model-driven architecture (mda)," in *Model-Driven Software Development*. Springer, 2005, pp. 1–16.

[2] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[3] OMG, "MDA Guide, Version 1.0.1."

[4] G. Gössler and J. Sifakis, "Composition for component-based modeling," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 161–183, 2005.

[5] F. Cao, B. Bryant, R. Raje, M. Auguston, A. Olson, and C. Burt, "Component specification and wrapper/glue code generation with two-level grammar using domain specific knowledge," in *the Proceedings of 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM'02)*, vol. LNCS 2495, 2002, pp. 103–107.

[6] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall Professional Technical Reference, 2005.

[7] M. Böhm, J. Bittner, D. Habich, W. Lehner, and U. Wloka, "Model-driven generation of dynamic adapters for integration platforms," in *the Proceedings of 1st International Workshop on Model Driven Interoperability for Sustainable Information Systems (MDISIS)*, 2008.

[8] J. Bézivin, S. Bouzitouna, M. Del Fabro, M. P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige, "A canonical scheme for model composition," in *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006)*, vol. LNCS 4066, 2006, pp. 346–360.

[9] P. Kelsen and Q. Ma, "A modular model composition technique," in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, (FASE 2010)*, vol. LNCS 6013, 2010, pp. 173–187.

[10] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel, "A survey on aspect-oriented modeling approaches," E188 - Institut für Softwaretechnik und Interaktive Systeme; Technische Universität Wien, Tech. Rep., 2007. [Online]. Available: http://publik.tuwien.ac.at/files/pub-inf_4920.pdf

[11] H. Kühn, F. Bayer, S. Junginger, and D. Karagiannis, "Enterprise model integration," in *Pceedings of the 4th International Conference on E-Commerce and Web Technologies (EC-Web 2003)*, vol. LNCS 2738, 2003, pp. 379–392.

[12] S. Zivkovic, H. Kuhn, and D. Karagiannis, "Facilitate modelling using method integration: An approach using mappings and integration rules," in *Proceedings of the 15th European Conference on Information Systems (ECIS 2007)*, 2007, pp. 2038–2049.

[13] N. Choi, I.-Y. Song, and H. Han, "A survey on ontology mapping," *ACM SIGMOD Record*, vol. 35, no. 3, pp. 34–41, 2006.

[14] S. Schmit-van Werweke, "A Bridging Mechanism for Adapting Abstract Models to Platforms," University of Luxembourg, Tech. Rep. TR-LASSY-12-10, September 2012, http://democles.lassy.uni.lu/documentation/bridging-tr.pdf.

[15] P. Kelsen and Q. Ma, "A lightweight approach for defining the formal semantics of a modeling language," in *the Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08)*, vol. LNCS 5301, 2008, pp. 690–704.

[16] P. Kelsen and Q. Ma, "A formal definition of the EP language," University of Luxembourg, Tech. Rep. TR-LASSY-08-03, 2008.

[17] P. Kelsen, "A declarative executable model for object-based systems based on functional decomposition," in *the Proceedings of First International Conference on Software and Data Technologies (ICSOFT 2006)*, 2006, pp. 63–71.

[18] B. Meyer, Ed., *Object-Oriented Software Construction Second Edition*. Prentice Hall, 1997.

[19] M. Frappier and H. Habrias, Eds., *Software specification methods: an overview using a case study*, ser. Formal Approaches to Computing and Information Technology (FACIT). Springer, 2001.

[20] OMG, "Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0," February 2011.

[21] SUN Microsystems, "JavaBeans, Version 1.01-A," 1997.

[22] A. Rountev, "Precise identification of side-effect-free methods in java," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 82–91.

[2]http://www.eclipse.org/aspectj/