Model-Driven Development: Where Does the Code Come From?

Insights Learned From a Case Study

Jicheng Fu
Computer Science Department
University of Central Oklahoma
Edmond, OK, USA
jfu@uco.edu

Wei Hao
Computer Science Department
Northern Kentucky University
Highland Heights, KY, USA
haow1@nku.edu

Farokh B. Bastani, and I-Ling Yen
Computer Science Department
University of Texas at Dallas
Richardson, TX, USA
{bastani, ilyen}@utdallas.edu

***Abstract*—Model-driven development (MDD) drastically changes the traditional view of software modeling, which no longer serves merely as documentation that will be put aside at a certain point during the development. Instead, MDD has made models an integral part of the development process. As a result, software designers and developers can focus on high-level problem solving instead of low-level implementation details. However, the current research focus is on model transformations and overlooks the importance of code generation, which includes the generation of infrastructural code (the static aspects of the system) and business code (the behavioral aspects of the system). In this paper, we first analyze the root cause about why existing MDD approaches are only good at generating the infrastructural code. Then, we propose a comprehensive approach that considers functional, dynamic, and object modeling. This approach is able to generate both infrastructural and business code. Finally, we present a case study to evaluate the proposed approach. Through this case study, we identify some insights on automated code generation in MDD. Our results demonstrate that it is not only likely, but also possible to fully automate the code generation process in MDD.**

Keywords: Model-Driven Development (MDD), Model-Driven Architecture (MDA), AI Planning, Component-Based Software Development (CBSD), Code Pattern

I. INTRODUCTION

Model-driven development (MDD) is an emerging software development approach that aims to bridge the semantic gap between the problem domain and solution domain. Specifically, models are not only used to construct high-level specifications, but are also essential artifacts of the development process. This is a big improvement over the traditional software development processes, in which the software design and development are disconnected. For example, during the software analysis and design phase, use cases, interaction diagrams, class diagrams, and other UML diagrams are constantly used to model the problems to be solved. However, these artifacts are only understandable by people, not by computers. When the coding phase starts, these diagrams are quickly put aside because very few people want to go back and change the design documents to make them consistent with the software development. The design documents gradually lose their values as the development proceeds. This tendency also highlights other serious problems, namely, maintenance and documentation problems. As the design documents are always out of date, developers who did not participate in the initial development have a hard time to understand the system through the available documents. Therefore, it is also difficult for them to maintain the system well.

MDD overcomes the aforementioned problem through two key themes, namely, raising the level of abstraction of specifications and raising the level of automation [14]. For example, model-driven architecture (MDA) [17], the industry standard for MDD defined by OMG, classifies models as platform independent models (PIM), platform specific models (PSM), and code. PIMs are used to construct high-level specifications that are closer to the problem domains, but are independent of any implementation platforms. Then, automatic transformations are performed to transform PIMs into PSMs, which are in turn transformed into the code. MDA enables designers and developers to keep their focus on the high-level PIMs, i.e., the problem solving itself. The technical details will be handled through model transformation from PIMs to PSMs, and then to code.

The current research focus is on model transformations [5][15]. When it comes to code, most of the works would state using code generation tools to generate the code [5][15]. It seems that code generation has been mature enough in MDD. However, the majority of existing works are only good at generating the infrastructural code (i.e., the stub/skeleton of the code), not the business code (i.e., the implementation of the business logics) [16]. Theoretically, this limitation is not surprising because of the following two reasons. (1) Due to raising the level of abstraction of specification, the low-level implementation details are left out from the modeling languages [14]. (2) UML 2.0 is the de facto standard for MDD. Classes, objects, and/or components are the major artifacts for high-level modeling. These artifacts belong to object modeling, which only reflects the static aspect of the system [1].

To bridge the semantic gap between the high-level models and low-level business logic details, Selic [14] suggested "heterogeneous models", in which "fragments in detail-level languages are directly embedded in the *appropriate* parts of the model". Although this approach seems practical, it remains unclear where the fragments are embedded, i.e., in PIM or PSM. It is also unclear whether it is possible to automate this process. Sharing similar ideas, Frame Oriented Programming (FOP) [13], a template-based approach, is used to generate both infrastructural and business code automatically. The code generation is done through reusing existing frames, which are fragments written in detail-level languages. However, the reuse is achieved through name matching, i.e., the names in the specification are used to match the names of frames. Such matching strategy is fragile, i.e., it may miss the semantic meaning. It is possible that the matching will fail to find the frame that has the correct semantic content, but with a different name. It is also possible that it may match a frame with the same name, but has a different semantic meaning.

Obviously, the link between the high-level models and the low-level business code is missing. We believe that artificial intelligence techniques are mandatory for intelligently locating the missing details and assembling these details into the appropriate places. In this paper, we present a comprehensive approach for system modeling including functional, dynamic, and object modeling to establish the missing link. AI planning is used to connect the different aspects of modeling so that the level of automation is raised and both infrastructural and business code can be automatically generated. Furthermore, we use a case study system to examine the proposed approach and summarize the insights learned from the case study.

The rest of the paper is organized as follows. Section II proposes a comprehensive MDD-based software development approach. Section III evaluates the proposed approach through a case study. In Section IV, we present the insights learned from the case study. Finally, we discuss the related works in Section V and conclude the paper in Section VI.

II. A Comprehensive Software Development Approach

In this section, we propose a comprehensive approach that considers different aspects of system modeling, namely, functional models, object models, and dynamic models. Above all, we show how to use AI planning to connect different models together so that design and development can be automated.

A. AI Planning

We introduce the definitions and notations in AI planning that will be used in the rest of this paper.

Definition 1. An AI planning domain is a 4-tuple $\Sigma = (P, S, A, \gamma)$, where P is a finite set of propositions; $S \subseteq 2^P$ is a finite set of states; A is a finite set of actions; and $\gamma : S \times A \rightarrow 2^S$ is the state-transition function.

Definition 2. An action in AI planning is a pair $\langle pre, eff \rangle$ consisting of precondition and effect.

Definition 3. A planning problem is a triple $\langle s_0, g, \Sigma \rangle$, where s_0 is the initial state, g is the goal state, and Σ is the planning domain.

From Definition 3, we can see that the specification of a planning problem is declarative, i.e., focusing on what to do, instead of how to do it. Given a planning problem $\langle s_0, g, \Sigma \rangle$, the AI planner is responsible for finding a plan of actions that lead the system from the initial state s_0 to the goal state g.

Due to the declarative nature, the use of AI planning to support MDD is not intrusive and, therefore, can be loosely coupled with MDD.

B. Functional Modeling

Functional models are used to depict the functionality of the system from the user's point of view [1]. Such models are directly built upon the problem domains and, hence, are understandable by users. UML use case diagrams are commonly used for functional modeling. Use cases are independent of any implementation details and, thus, can be treated as platform independent models (PIMs).

However, use cases by themselves may be interpreted in many different ways. To avoid ambiguity, the semantic meaning should be formally specified. The popular way to specify semantics is to use pre-condition and effect $\langle R, E \rangle$, which is declarative and naturally fits in high-level modeling.

C. Connecting Functional and Dynamic Modeling

Since use cases only present the outside view of the system, the internal behaviors of the system should be captured by using dynamic modeling. For a particular use case, a sequence diagram and/or an activity diagram is usually constructed to illustrate the internal behaviors. The current practice typically builds dynamic models manually by designers. In this paper, we propose using AI planning to facilitate the construction of activity and sequence diagrams based on the semantics of use cases.

If we look at the way of formally specifying the semantics of a use case (i.e., $\langle R, E \rangle$) and the definition of a planning problem (i.e., $\langle s_0, g, \Sigma \rangle$ in Definition 3), we can see that it is possible to transform the semantics of a use case into an AI planning problem. Specifically, we can transform the precondition R into the initial state s_0 and transform the effect E into the goal g. Then, what is left is to define the planning domain Σ.

To locate the planning domain Σ, the focus should be put on the platform specific models (PSM). From Section I, we can see that no matter whether it uses heterogeneous models [14] or code templates [13], the assumption is that we can find platform specific models to reuse. In fact, such kind of reuse is possible. For example, component-based software development (CBSD) techniques [18] are concerned with reusing existing software components to build larger applications at a lower cost and risk and in less time. Previously verified or tested components serve as building blocks to construct reliable and dependable application systems. As another example, semantic Web services [10] are loosely coupled with each other and can be easily reused.

Figure 1 shows how AI planning can be used to connect functional and dynamic modeling. In the first step, the specifications of use cases are transformed into planning problems. In the second step, AI planner works on the planning domain, which is an abstraction of PSMs, to find a plan. Finally, the generated plan is represented as a UML sequence diagram or activity diagram. It needs to be emphasized that the generated plan is based on PSMs and the designers may need to perform some abstractions so that the dynamic models will represent PIMs. In other words, the last step may require human's involvement.

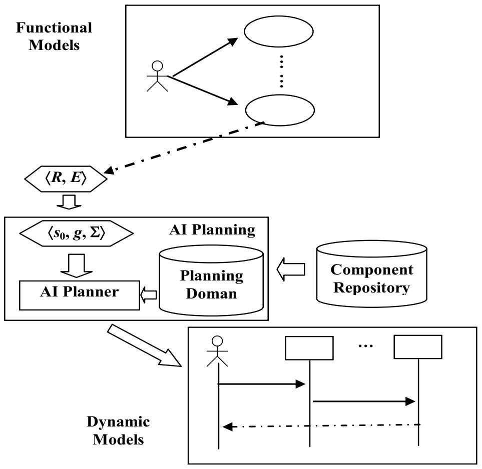


Figure 1. Using AI Planning to Connect Functional and Dynamic Modeling

D. An Advanced Program Generation Framework

Based on the functional and dynamic models, we can start to build the object models. UML class diagrams are commonly used for object modeling. As discussed in Section I, object models only represent the static aspects of the system. This is the reason why existing MDD approaches are only good at generating the infrastructural code (i.e., only include the definitions of classes and declarations of operations without implementations of the operations).

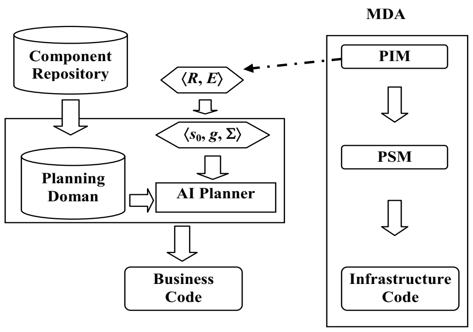


Figure 2. A Generic Program Generation Framework

In [3], we proposed a conceptual framework for automated code generation including AI planning, MDA, and Component-based software development (CBSD). Here, we generalize the framework, which is shown in Figure 2. MDA and AI planning are loosely coupled through specification transformations. The principle of such coupling comes from the fact that we can perform functional decomposition so that the high-level use cases can be decomposed into lower level use cases. Such functional decomposition stops when the use cases represent the operations defined in classes. Therefore, the semantics of the operations are also formalized as $\langle R, E\rangle$, i.e., the precondition and effect.

The planning domain is abstracted over the component repository. The implication is that existing components need some kind of formal specification. For example, the ontology language OWL-S [8] models a Web service as a 4-tuple, namely, inputs, outputs, precondition, and effects. Similarly, we proposed the concept of code patterns [2][7], which captures the typical usages of components and defines their possible calling sequences. The semantics of a code pattern is also specified with preconditions and effects.

From Figure 2, we can see that code is classified into two categories, namely, infrastructural code and business code. MDA is responsible for generating the infrastructural code through model transformation, i.e., from PIM to PSM and, then, from PSM to code. On the other hand, the AI planning based subsystem is responsible for generating the business code. Specifically, this is achieved through automatically generating the glue code needed to meet a given specification by assembling the system from existing components.

E. An Implementation of the Framework

We implemented the generic program generation framework with some specialized techniques. Specifically, we use IBM Rational Rose [12] for the MDA platform. Strictly speaking, Rational Rose is not designed for MDA. However, with its excellent modeling and code generation capability, it at least possesses the basic features of MDA.

For the reuse of existing components, we use code patterns [2][7], a component-based software development (CBSD) technique. The definition of code pattern is as follows.

Definition 4 (Code Pattern). A code pattern *cp* is a named functional unit that captures the typical structure and composition of a set of components. *cp* is represented by a triple *cp* = (*i*, *b*, *c*), where *cp* is the pattern name, *i* is the interface, *b* is the body, and *c* is a pair of precondition and effect $\{R, E\}$. The functionality of a pattern *cp* can be represented as $\{R\}cp\{E\}$.

The pattern body *b* is a code template that captures the typical way of using components. For example, we need two lines of code in Java to locate a Web service with the service stub, which is shown as follows.

```
?Service    locator = new  ?ServiceLocator();
?    locservice = locator.get?();
```

Here, "?" represents the service name. In the first step, the service locator is created and in the second step, an instance of the service is obtained. The use of code patterns can significantly reduce the chance of repeatedly writing similar code segments.

The pattern interface *i* defines all the parameters including the input parameters P_{in} that are used to instantiate the pattern body b and the output parameters P_{out} that are used to return the computation results.

Besides the formal semantic specification, code patterns have another major difference from a code template, which is the use of pattern operators. There are three pattern operators, namely, concatenate, splice, and reverse, defined over code patterns to facilitate glue code generation [2].

For AI planning, we use our own AI planner, FIP [4]. FIP is three orders of magnitude faster than other state-of-the-art AI planners and, hence, is well suited for automated code generation.

III. CASE STUDY

In this section, we assess the proposed comprehensive MDD approach through a Web-based E-Government inspection system. Assume that a city government has already developed a backend application system. The central business of the application system is related to inspections of housing foundations, fire alarms, and so on. Inspectors can create, reschedule, and cancel inspections. They can also make itineraries on a specific day to conduct inspections that are scheduled on that day. All these functionalities are supported by different Web services in the application system. Now, the city government wants to develop a Web-based system to provide a bridge between the inspectors and the backend application system. Figure 3 shows the overall picture of the whole system.

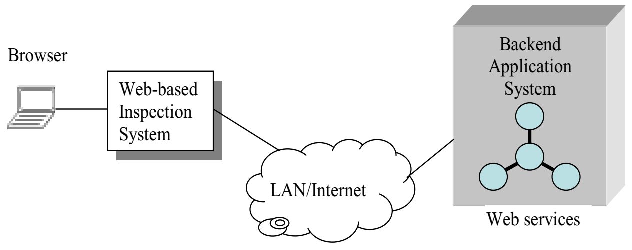


Figure 3. Overall System Structure

The major Web services involved in the system are listed in Table I.

TABLE I. MAJOR WEB SERVICES INVOLVED IN THE SYSTEM

Service	Comment
Authenticate	Require users to provide user names and passwords and check their validity
Create Inspection	Create a new inspection
Reschedule Inspection	Change the inspection date/time.
Cancel Inspection	Remove an inspection from the system
Locate Inspection	Find a particular inspection
Search Inspections	List a set of inspections according to some search criteria, e.g., date, location, etc.
GIS	Given a set of locations, return an itinerary that is time and fuel efficient
Fax	Fax itinerary or inspections to users
Email	Email itinerary or inspections to users
Callback	Call users to notify them about the itinerary or inspections
Reserve a car	The government provides a car for inspectors to carry out the inspections.
Car rental	If the internal cars are all scheduled on a specific day, the inspector can rent a car instead.
Reserve Taxi	The inspector can also reserve a taxi for transportation.
Credit card payment	Related to car rental or taxi reservation

A. *Web-based E-Government Inspection System*

The goal of this case study is to develop a Web-based inspection system. Here, the backend application system (consisting of all the Web services) is outside of the Web-based inspection system. The inspection system consists of four subsystems, namely, inspection management, itinerary, notification, and transportation. Figure 4 illustrates the top-level system structure.

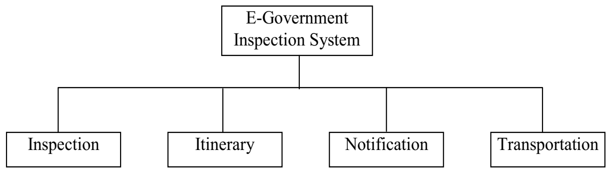


Figure 4. Top Level System Structure

The inspection management subsystem enables the inspectors to create, reschedule, cancel, and query inspections. This subsystem provides the main business functionalities and interacts with other subsystems to provide convenient services to inspectors. The itinerary subsystem provides the GIS (Geographic Information System) service. An inspector usually has to inspect several locations in one day. The inspector inputs the specific date for inspections and uses the search inspections service to locate the set of inspections scheduled on that day. Then, the GIS service provides an itinerary that should be efficient in terms of both the time and the distance traveled.

The notification subsystem enables inspectors to fax, email, or callback their inspection query results or itinerary details to themselves for their own record. After the inspection itinerary is prepared, the inspector needs to reserve a car. Normally, the government provides the cars to inspectors. In case all the cars are currently being used, the inspector can choose to rent a car or reserve a taxi to carry out the inspections. Inspectors can pay the related fees with the credit card payment service.

In addition, the user authentication is mandatory, i.e., a user is required to provide his/her user name and password to continue to operate the system.

B. *Component Repository.*

In this case study, we map Web services into code patterns. In industry, the typical way of invoking Web services is shown in Figure 5. The clients use client stubs to invoke Web services. The functionality of each Web service is delegated to the client stub, which is stored locally with the client programs. To users, it seems that the Web service itself is stored locally and the invocation of the Web service is just like a regular function call. Then, the client stub wraps up the request as a SOAP message and sends it to the server stub.

Therefore, the typical ways of invoking a Web service through client stubs can be modeled as code patterns. Typically, the capability of a Web service is modeled with a 4-tuple, namely, ⟨inputs, outputs, preconditions, effects⟩, i.e., ⟨I, O, P, E⟩ [8]. We have formulated a direct mapping from a Web service to a code pattern. Specifically, I is mapped to the input parameters P_{in} defined in the code pattern interface i; O is mapped to the output parameters P_{out}; P is mapped to the precondition R; and E is mapped to the effect E. For code patterns, there is an extra field, which is the pattern body b. The typical ways of using the client stub can be put into the pattern template to facilitate code generation. Therefore, a Web service is mapped perfectly into a code pattern.

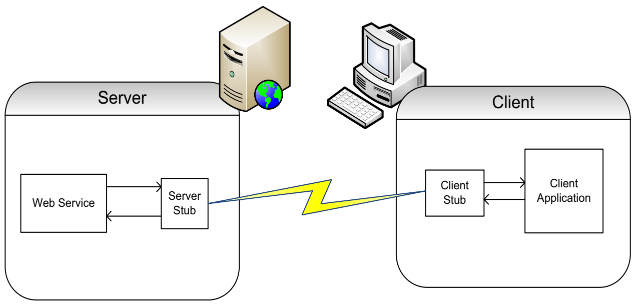


Figure 5. Invocation of Web Service

C. *High-Level System Design*

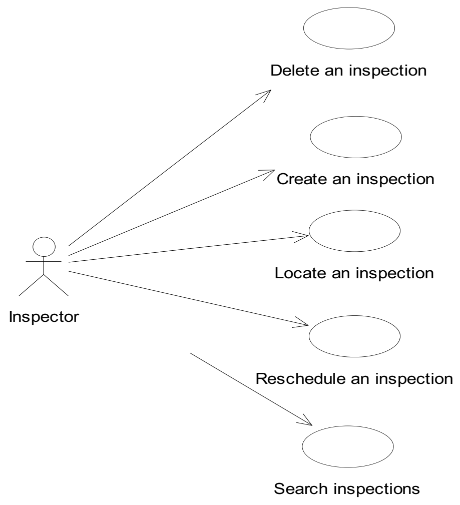


Figure 6. The Use Cases for the Inspection Management Subsystem

Use cases are the first tangible things that stakeholders interact with. Due to space limitation, we only show the use case diagram for the inspection management subsystem (in Figure 6). The use case diagram includes five use cases, namely, create an inspection, reschedule an inspection, delete an inspection, locate an inspection, and search a set of inspections according to a certain criteria.

1) Specifying semantics of use cases. As discussed in Section II.B, preconditions and effects are imposed on a use case to specify the conditions under which the use case can be applied and what effects the use case is expected to generate. For example, the precondition for the use cases of "Locate an inspection" is the availability of the user name, password, and the confirmation number of the inspection. The effect is that the inspection is located or does not exist.

2) Deriving dynamic models. As illustrated in Figure 1, the specifications of use cases are transformed into AI planning problems. Then, the AI planner works on the planing domain (which is derived from the component repository) to look for plans for these planning problems. The plans can be interpreted in many different ways, such as UML activity diagrams or UML sequence diagrams. For example, Figure 7 shows the activity diagrams for use cases of "Locate an inspection" and "Delete an inspection". Figure 8 shows the sequence diagram for the use case of "Delete an inspection".

The translation of the generated plans into the corresponding dynamic models (e.g., activity diagrams and sequence diagrams) was manually done in this case study. However, it is possible to automate this process in the future study.

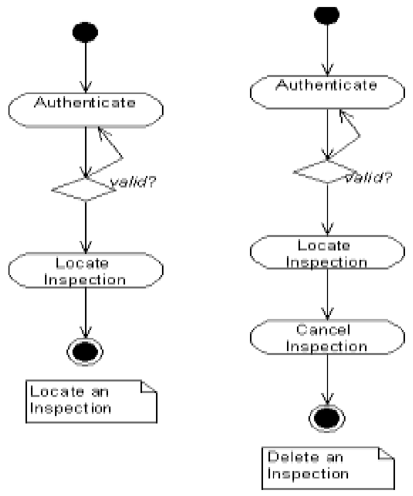


Figure 7. Activity Diagrams for Use Cases of "Locate an inspection" and "Delete an inspection".

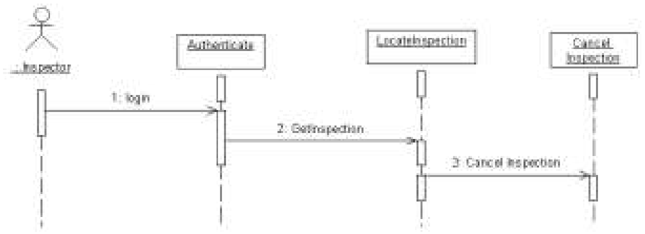


Figure 8. Sequence Diagram for "Delete an inspection"

3) Building the object models. Based on the functional and dynamic models, we built the object models. Especially, a sequence diagram depicts a flow of events consisting of objects participating in a use case. The flow gives us a clear picture about the business logics of the Web-based inspection system. They enable us to deepen the understanding of the system and provide insights to design the correct PIM. In addition, the dynamic models help us identify objects that will be used in the object modeling. For example, the flow of events in Figure 8 leads to the following PIM design shown in Figure 9. The model depicted as the square shape in Figure 9 represents the client side Web page and the models depicted as the gear shape represent server side or dynamic Web pages. It should be noted that these models for Web pages are independent of any implementations. For example, the server pages (i.e., depicted as the shape of gear) can be implemented with JSP (Java Server Page), ASP (Active Server Page), or PHP.

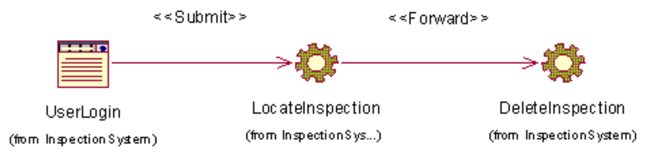


Figure 9. PIM for "Locate an inspection" and "Delete an inspection"

Combining all such pieces together, we are able to generalize the design of PIMs and their relationship. Figure 10 shows the major part of the PIMs that is designed for the E-Government system. The relationships among different PIMs are also defined.

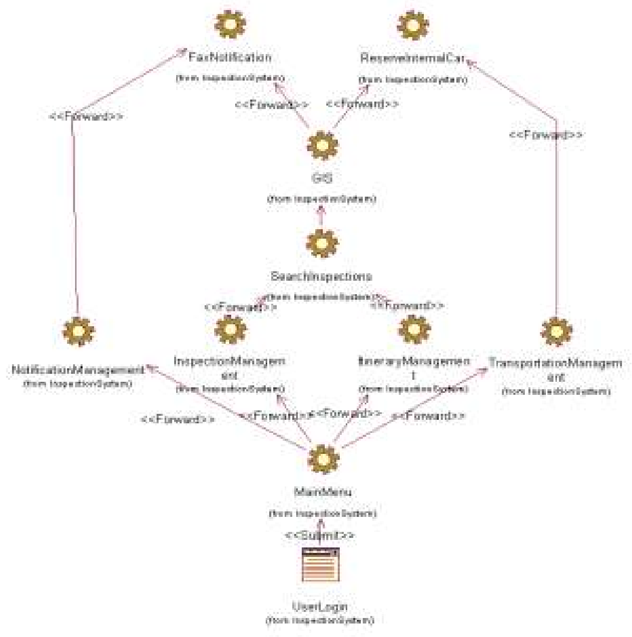


Figure 10. Major Parts of PIMs

D. Code Generation

As discussed in Section I, we classify code as infrastructural code and business code. To generate the infrastructural code, we rely on Rational Rose's code generation capability. Both client-side static HTML Web pages and the skeletons of server-side dynamic server pages are automatically generated. This is the static aspects of the E-Government inspection system because Rational Rose cannot generate business code for the server pages.

To generate business code, we need to specify the semantics of PIMs so that the specifications can be transformed into AI planning problems. As discussed in Section II.D, preconditions and effects are used to specify the semantics of the PIMs. Since the models are mostly obtained from the plans returned by the AI planner, their semantics can be found from the corresponding actions in the plans, i.e., use the preconditions and effects of these actions in the corresponding models.

After the semantics of PIMs are formally specified, we transform the specifications into AI planning problems. Then, we run the AI planners to generate plans, based on which the business code is synthesized over the component repository (see Section III.B).

In summary, AI planning plays two critical roles in the proposed MDD-based approach. First, AI planning connects the functional modeling with dynamic modeling in the high-level design. Second, AI planning-based synthesis can generate business code based on the semantics of object models.

IV. Discussion

In this section, we first discuss the insights learned from the case study. Then, we identify a few limitations of the study.

A. Insights

Through the case study, we can see that it is possible to automatically generate the complete system. To ensure this, the following requirements must be satisfied.

1) Functional and dynamic modeling must be integrated into object modeling. Object modeling alone is insufficient. This is the major reason why existing MDD approaches are not good at generating business code. Object modeling only represents the static aspects of the system. We must integrate functional and dynamic modeling into object modeling so that both business code and infrastructural code can be automatically generated.

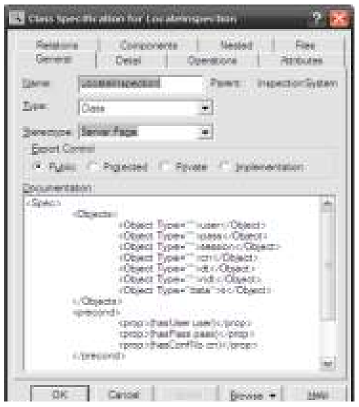
2) The techniques imposed upon MDD should be unintrusive . In other words, any techniques imposed on MDD should be loosely coupled with MDD. This requirement ensures that the benefits of MDD will not be compromised. For example, our proposed approach uses AI planning, which interacts with MDD indrectly through specification transformations.

3) Artificial intelligence techniques are mandatory. It is a violation of the MDD principles if we manually put the detail-level templates to the "appropriate" places because otherwise designers and developers cannot simply focus on the high-level problem solving, but will have to worry about the low-level technical details as well. Therefore, artificial intelligence techniques will be helpful in "intelligently" locating the low-level details and assembling these details into the appropriate places. In our case, we use AI planning to select and organize existing components to achieve this goal.

4) Code reuse. Irrespective of whether we use heterogeneous models, FOP, or the approach proposed in this paper, code reuse is critical in code generation.

B. *Limitations of the Case Study*

When specifying the semantics, we used the combination of XML and PDDL [9], an AI planning specification language. Figure 11 shows how we specify the semantics of a model in Rational Rose. In fact, OWL-S [8] uses a similar approach by mixing XML and formal specification languages. However, such methods may impose extra burdens on designers and developers because they have to know PDDL as well as other high-level specification languages. In addition, such methods may increase the possibility of failing to find a solution simply because the way in which the specification is written may not match the specifications of components in the component repository. Therefore, it would be beneficial if the knowledge in the component repository is presented in a way that can be easily used during the specification. Ontology may help mitigate this problem. A standardized high-level specification language may help reduce the workload on software designers and developers.



Name: LocalInspection   Parent: InspectionSystem
Type: Class
Stereotype: Server Page
Export Control: [✓ Public] Protected Private Implementation
Documentation:

```
<Spec>
        <Objects>
                <Object Type="">user</Object>
                <Object Type="">pass</Object>
                <Object Type="">session</Object>
                <Object Type="">cn</Object>
                <Object Type="">vdt</Object>
                <Object Type="">vid</Object>
                <Object Type="data">c</Object>
        </Objects>
        <precond>
                <prop>hasUser user</prop>
                <prop>hasPass pass</prop>
                <prop>hasConfNo cn</prop>
        </precond>
```

Figure 11. Example of Formal Specification in Rational Rose

V. Related Works

MDA (Model-Driven Architecture) [6][11] has attracted wide attention. The developers start from the design of high level PIMs (Platform Independent Models) and use transformations to map models to a lower level. Hence, developers can concentrate on the development of PIMs, which are a higher level of abstraction than the actual code. PSMs are generated from PIMs through transformation. Also, the codes are in turn generated from PSMs. These processes can be automated to increase productivity. In addition, by focusing on PIMs, developers can put more efforts on dealing with business issues, which is another favorable factor for speeding up the development process. However, transformations are good at generating the static infrastructural code instead of the behavioral business code.

To overcome the limitation of the transformation method, the concept of "heterogeneous models" [14] is introduced to empower MDA to generate business code. In this type of model, PIM and PSM are still specified with the original modeling language. Segments written in low-level languages are embedded in the appropriate parts of the high-level components. The major advantage of this approach is that existing code can be reused and business code can be generated. However, the mix of high-level models and low-level segments may make the design difficult to understand and may neutralize MDA's benefits of portability and documentation.

Frame oriented programming (FOP) [13] was proposed to generate both infrastructural and business code. This approach is used in industry and is an effective form of template driven code generation. The business code generation is based on the reuse of existing templates through name matching, i.e., the names of the operations in the specification should match the names of existing templates. Name matching may lose the semantic meaning and, therefore, lead to errors.

VI. Conclusions and Future Work

In this paper, we point out a missing link between the high-level models and the business code in MDD. We analyze the root cause of the missing link: Object modeling alone is insufficient in generating a complete system. To establish the link, we present a comprehensive software development approach based on MDD that is able to generate both infrastructural code and business code automatically. Specifically, we use AI planning to connect functional, dynamic, and object modeling together in MDD. AI planning is loosely coupled with MDD and, therefore, does not hurt the MDD's advantages of portability and documentation.

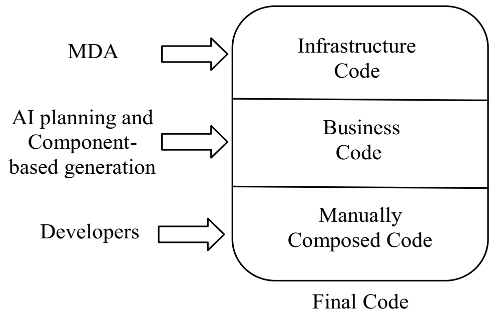


Figure 12. Sources to Obtain the Final Code [3]

To evaluate the proposed approach, we conducted a case study to automatically generate a Web-based E-Government inspection management system. We showed how to use AI planning to facilitate the construction of dynamic models (e.g., activity diagrams and sequence diagrams) from the functional models (e.g., use cases). Based on the functional models and dynamic models, we constructed the object models. The semantics of the object models were formally specified and subsequently transformed into AI planning problems.

Eventually, as shown in Figure 12, the infrastructural code was generated by MDA and the business code was generated by AI planning and component-based program generation approach. Although the code was completely generated in this case study, in practice it is possible that some parts of the system may not be automatically generated. In such cases, software developers will have to manually develop the code to fill in the blank.

In the next step, we will enhance the proposed approach to overcome the limitations discussed in Section IV.B, i.e., to make the knowledge of the underlying component repository readily and easily available in the high-level design.

REFERENCES

[1] B. Bruegge and A. H. Dutoit, Object-Oriented Software Engineering: Using UML, Patterns and Java, 3rd Edition, *Prentice Hall*, 2009

[2] J. Fu, F. B. Bastani, and I. Yen, "Automated AI Planning and Code Pattern Based Code Synthesis". *ICTAI* 2006, pp. 540–546.

[3] J. Fu, F. B. Bastani, I. Yen, "Model-Driven Prototyping Based Requirements Elicitation", The 14*th Proceedings of Monterey Workshop*, 2007.

[4] J. Fu, V. Ng, F. B. Bastani, and I. Yen, "Simple and Fast Strong Cyclic Planning for Fully-Observable Nondeterministic Planning Problems", *IJCAI*-2011, pp. 1949–1954.

[5] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, "Transformation: The Missing Link of MDA", *Proceedings of the* 1*st International Conference on Graph Transformation*, Barcelona, Spain (2002), pp. 90–105.

[6] A. Kleppe, J. Warmer, and W. Bast, MDA Explained: The Model Driven Architecture: Practice and Promise. *Addison-Wesley*, 2003.

[7] J. Liu, F. B. Bastani, and I. Yen, Code Pattern: An Approach for Component-Based Code Synthesis, *Proceeding of the* 7th *World Multiconference on Systemics*, Cybernetics and Informatics, Orlando, FL, pp. 330–336, 2003.

[8] D. Martin et al, OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/, 2004

[9] D. McDermott, *et al.*, "The PDDL Planning Domain Definition Language", The *AIPS*-2004 *Planning Competition Committee*, 2004.

[10] S. McIlraith, T.C. Son, and H. Zeng, "Semantic web services", *IEEE Intelligent Systems*, 16(2):46–53, March/April 2001.

[11] Object Management Group, "MDA Guide: Version 1.0.1", OMG document omg/03–06–01, 2005.

[12] Rational Rose Family, IBM/Rational Software Corp., 2003, available: www.rational.com/products/rose/index.jsp.

[13] F. Sauer, "Metadata driven multi-artifact code generation using Frame Oriented Programming", *OOPSLA*, 2002.

[14] B. Selic, "Model-driven development: Its essence and opportunities", 9*th IEEE International Symposium on Object and component-oriented Real-time distributed Computing* (*ISORC*), pp. 313–319, 2006

[15] S. Sendall and W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development", *IEEE Software*, 20, No. 5, 42–45 (2003).

[16] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen, Model-Driven Software Development: Technology, Engineering, Management. *John Wiley*, Chichester (2006)

[17] Object Management Group, MDA Guide, Version 1.0.1, OMG document omg/03-06-01, 2005.

[18] S. S. Yau and N. Dong, "Integration in Component-based Software Development Using Design Patterns", *The Twenty-Fourth Annual International Computer Software and Applications Conference*, COMPSAC, Taipei, Taiwan, 369.