Model-Driven Prototyping Based Requirements Elicitation

Jicheng Fu, Farokh B. Bastani, and I-Ling Yen

Department of Computer Science
The University of Texas at Dallas
P.O. Box 830688, EC 31
Richardson, TX 75083-0688 USA
{jxf024000,bastani,ilyen}@utdallas.edu

Abstract. This paper presents a requirements elicitation approach that is based on model-driven prototyping. Model-driven development fits naturally in evolutionary prototyping because modeling and design are not treated merely as documents but as key parts of the development process. A novel rapid program synthesis approach is applied to speed up the prototype development. MDA, AI planning, and component-based software development techniques are seamlessly integrated together in the approach to achieve rapid prototyping. More importantly, the rapid program synthesis approach can ensure the correctness of the generated code, which is another favorable factor in enabling the development of a production quality prototype in a timely manner.

Keywords: Requirements Elicitation, Prototyping, Component-Based Software Development, Code Patterns, Model-Driven Development.

1 Introduction

The primary measure of success in a software system is the degree to which it meets the purpose for which it is intended [23]. Therefore, requirements engineering (RE) activities are vital in ensuring successful projects. In [11], RE is defined as a branch of software engineering concerned with real world goals, functions, and constraints on software systems. RE facilitates the transformation from informal requirements to formal specifications, which serve as the basis for subsequent development. However, the secret behind the scene for the transformations is difficult to formulate because of the problems of uncertainty, ambiguity, inconsistency, etc., inherent in the process.

Prototyping is a popular requirements elicitation technique because it enables users to develop a concrete sense about software systems that have not yet been implemented. By visualizing the software systems to be built, users can identify the true requirements that may otherwise be impossible. Prototyping was once regarded as the solution to RE. It has many advantages [4][15], including:

- Reduced time and cost. Problems can be detected in the early stages. Therefore, the overall cost is greatly reduced.
- Concretely present the system operations and facilitate design decisions.

B. Paech and C. Martell (Eds.): Monterey Workshop 2007, LNCS 5320, pp. 43–61, 2008.

- Stakeholders from all parties can actively get involved in the development process.

Prototyping is especially useful when there is a great deal of uncertainty or when early feedback from stakeholders is needed [4]. There are two types of prototyping, i.e., rapid prototyping (throwaway) [15][17] and evolutionary prototyping [16]. Rapid prototyping focuses on the demonstration of functionality and obtaining early feedback on requirements that are poorly understood. The essential idea is to develop a prototype system containing any unclear requirements as quickly as possible. There may be bugs in the prototypes and the overall quality of the implementation may not be good. But these are tolerable in rapid prototyping. Hence, this type of prototype is referred to as quick-and-dirty and will be discarded after any unclear requirements have been clarified [4].

Evolutionary prototyping, on the other hand, is developed as a portion of the actual system. It focuses on the requirements that have already been well understood. New requirements and features are incrementally added as the development proceeds in an iterative manner. The prototype is developed to be of production quality and will not be thrown away [4].

However, prototyping is not thriving as expected due to the following reasons [21]:

- Management can get confused by the prototype and the production quality version to be built. They may expect that the final deliverable will come quickly based on enhancement and refinement of the prototype;
- Poor quality codes from the prototype may remain in the final system due to the tendency of reusing previously written code fragments.
- Lack of mechanisms for requirements traceability.
- Prototypes may not be developed quickly due to the system complexity and technical limitations.

The last two reasons are the most important factors that hinder the use of prototyping. Technical people may tend to make the prototype overcomplicated, resulting in some artifacts that are not linked back to the original requirements. Another tendency is the omission of some functionality because stakeholders may be absorbed in some aspects, e.g., user interfaces, etc., while neglecting other aspects. The most valuable property of prototyping is the fact that it can be done quickly. The lack of systematic rapid development approaches makes it hard to fulfill this property. Without this property, prototyping cannot have significant impact on industry.

It is, therefore, desirable to have a prototyping approach that can leverage the advantages of rapid and evolutionary prototyping. Specifically, prototypes should be developed quickly and still maintain satisfactory quality. To achieve this goal, we need to meet the following objectives:

(1) Make software design a part of the development process.
(2) Achieve a certain level of automation to speed up the development.
(3) Make requirements traceable.

Based on these objectives, we propose a model-driven development (MDD) based prototyping approach. The use of model-driven approaches is especially amenable to requirements engineering because it meets the aforementioned objectives. First, in MDD, system design has become a part of the development process. UML 2.0,

developed to support MDD, has changed the view that UML diagrams only serve as temporary documents and will be put aside at later points during the development process. Combined with OCL (Object Constraint Language), UML is able to specify models in a formal way. OCL is a declarative and precise specification language, which has no side-effects and does not change the state of the system [30]. It enables errors to be found early in the life-cycle, when fixing a fault is relatively cheap.

Second, MDD can automate the generation of infrastructural code (i.e., code frames) through transformations between platform independent models (PIMs) and platform specific models (PSMs) and between PSM and code. Specifically, PIM and PSM are designed to raise the level of abstraction. PIMs are models with high level abstractions that are independent of the implementation technology [9]. PSMs are bound to specific platforms and implementation technologies. PSMs are generated from PIMs through transformation and the code is in turn generated from PSMs. These processes can be automated to increase productivity. Thus, developers can concentrate on the development of PIMs, which are at a higher level of abstraction than the actual codes. This is another favorable factor for speeding up the development process.

Third, traceability is a desired feature for the design of model-driven development tools. The existing MDD tools support a certain level of traceability. Hence, it makes the development process amenable to requirements changes. It is always easier to indicate what part of a PIM is affected by the changed requirements than to determine code segments that must be modified. When parts of the code are traced back to elements in the PIM, it would be much easier to make an impact analysis of the requested changes [9].

Although transformations that map models to the next level are typically used in MDD [24][28], there are some doubts about the practicality of generating complete systems solely via transformations. Transformations are good at generating infrastructure codes instead of business codes. In order to further speed up the development of prototypes, a novel program synthesis technique is applied to the proposed approach. The program synthesis technique combines AI planning and component-based synthesis techniques to achieve automated generation of business/logic code. Specifically, we design and implement a fast planning graph based iterative planner, called FIP [6]. It can deal with nondeterministic actions (actions that can generate multiple possible effects) and generate parameterized procedure-like generic reusable plans, which are called procedural plans. FIP can help automate the selection and organization of underlying components to achieve the given goal. The underlying component-based synthesis technique serves as the basis for the final code generation. It is based on a component-based software development (CBSD) technique, code pattern [13][14], which is concerned with reusing existing software components to build larger applications at a lower cost and risk and in less time. The AI planning and component-based program synthesis technique can be seamlessly integrated with MDA to achieve even more rapid program synthesis. In this hybrid system, the development of PIM still relies on human intervention. However, PIM is independent of any implementation details and has a higher abstraction level than code. Hence, the designers can put more efforts on the business-logic related aspects of the system. Then, the static aspect of the system will be generated through MDD's transformation technique and the dynamic aspect will be generated through the AI planning and component-based program synthesis technique.

The rest of this paper is organized as follows: Section 2 overviews the techniques involved in the proposed model-driven development based prototyping approach. Section 3 presents a novel rapid program synthesis approach, in which MDA, AI planning, and component-based program synthesis techniques are seamlessly integrated together. Section 4 discusses requirements elicitation through the proposed prototyping approach based on the advanced rapid program synthesis approach. Section 5 concludes the paper and identifies some future research directions.

2 Overview

As rapid prototyping focuses on unclear requirements and evolutionary prototyping focuses on well understood requirements, neither of them alone is sufficient to represent a complete system. The proposed approach intends to combine the advantages of both methods and develop a prototype in a timely manner and of production quality. In this sense, the proposed approach is a rapid evolutionary prototyping approach.

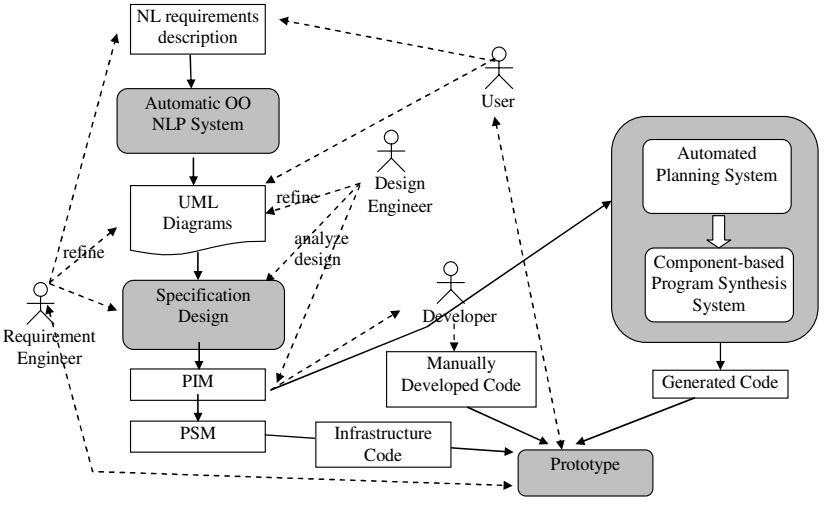

Fig. 1. People and techniques involved in the proposed approach

Fig. 1 gives an overview of the people and techniques involved in the proposed rapid evolutionary prototyping approach. During the requirements specification process, use cases are the first tangible things that stakeholders interact with. They document initial requirements and provide scenarios illustrating interactions with end users or other systems to achieve specific business goals. Use cases and other UML elements can be automatically generated by tools [7][25] that employ natural language processing (NLP) techniques to capture essential and relevant software requirements from natural language descriptions. This can help automate Object-Oriented Analysis (OOA) though the tools are not mature and only aid the requirements acquisition and analysis process. Human involvement is mandatory, especially when contradictions exist in the requirement specification.

Although MDA can generate the infrastructure code through transformations, it is not good at generating code for the dynamic aspects of the system. In order to speed up the development process as well as improve the quality of the implemented prototype, a novel rapid program synthesis approach is used. The techniques involved in the approach are MDA, AI planning, and component-based code synthesis, which are organized in a hierarchy and seamlessly integrated together. The top level is the PIM of MDA. PIM is specified using UML with OCL. It presents planning problems to the underlying automated planning system (APS), which is located in the middle of the hierarchy. Based on the planning problem, the AI planner in the APS generates a procedural plan, in which its underlying components are chosen and organized to achieve the given goal. The generated plan is then fed to the component-based synthesis system that is located at the lowest level. The final code is then generated by the code synthesis system. The developers only need to focus on the incomplete parts where the planner cannot find a suitable solution. This can alleviate the developers' burden and increase the development speed and reliability of the system. Section 3 discusses the details of the novel rapid program synthesis approach.

After the system is complete, users can visually operate it and formulate new requirements to cope with any problems. These will be fed back to the requirements engineers and the development cycle is repeated.

3 Rapid Program Synthesis

In our proposed evolutionary prototyping approach, rapid program synthesis technique plays a critical role. It ensures that the prototype is built in a timely manner and with production quality.

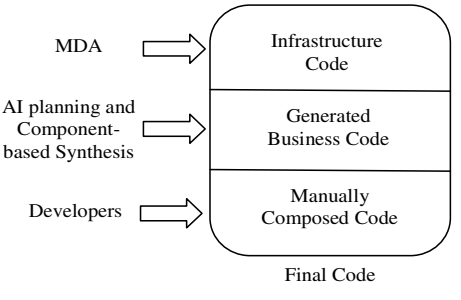

Fig. 2. Ways to obtain the final code

Fig. 2 shows how the final codes are obtained. The infrastructure codes (static aspects of the system) are generated by MDA through transformation. The AI planning and component-based synthesis method can generate business codes, which constitute the behavioral aspects of the system. The parts that cannot be generated automatically have to be implemented manually.

In Section 3.1, we introduce how MDA helps generate the infrastructural code through transformation. In Section 3.2, we introduce how the dynamic aspects of the

system are generated by the AI planning and Component-based synthesis approach. Then, in Section 3.3, we discuss how to integrate the AI planning and component-based synthesis approach with MDA so that both the static and some parts of the dynamic aspects of the system can be automatically generated.

3.1 MDA

Model-driven architecture (MDA) has attracted considerable research interests and is predicted to be the next generation software development method. MDA transforms models written in one language into models in another language. The direction of transformation is usually from high level models to low level models. Fig. 3 illustrates the relationships between PIM, PSM, and codes. PIM is designed independently of any implementation details. It comes at a higher level of abstraction. PIM is then transformed into PSM, which is a domain specific model that relies on specific domains and technology. In the final step, PSM is transformed into code. The mainstream MDA tools, e.g., [8], support these transformations.

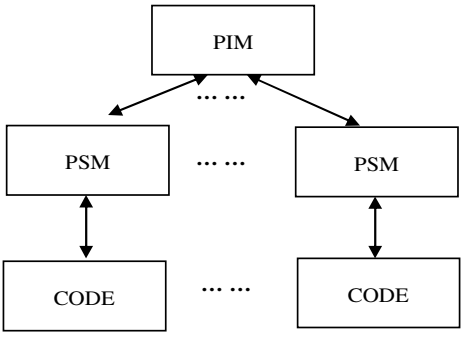

Fig. 3. Relationships between PIM, PSM, and code [9]

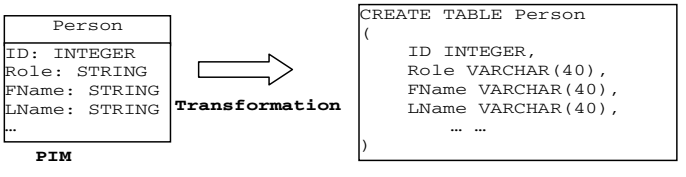

Fig. 4. Example of PIM to relational transformation

Transformation introduces automations in generating models and codes and, thus, the productivity is increased greatly. For example, assume that a "surprise" test management system is proposed to be developed for the case study in [19]. The system keeps track of the surprise test records of screeners. For a surprise test, the inspector purposely introduces fake weapons into a normal bag as a test. If the screener misses too many tests, he/she will be sent for training. This surprise test management system can be used to illustrate the transformation-based prototyping approach. The PIM to

relational transformation can be fully automated by generating the corresponding SQL clauses. Suppose that there is a database used to store users (screeners, inspectors, etc.), test to be conducted, test history, etc. For example, the PIM "Person" is defined as shown in Fig. 4. The attribute "Role" is used to distinguish inspectors and screeners.

It is very natural to do the transformation from PIM to its relational counterpart automatically through transformation. However, transformation is only good at generating code related to the static aspects of the system, i.e., infrastructure code. For example, Fig. 5 shows the transformations between PIM and PSM and between PSM and code. J2EE technology is used in this example to illustrate the idea. The PIM model "Person" is transformed into three PSM models tailored to fit within J2EE specifications. The PSM models (EJBs) are in turn transformed into code. We call the code as the infrastructure code because it only contains static code frames and/or getter/setter methods. The business code is absent from the transformation.

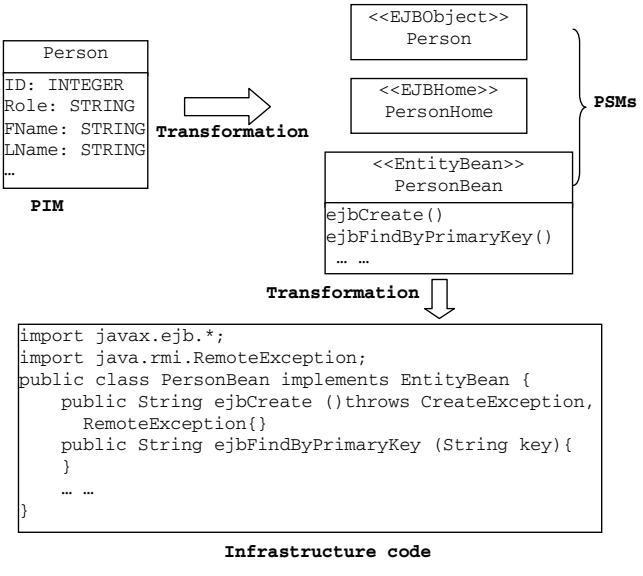

Fig. 5. Example of PIM to PSM and PSM to code transformations

To overcome the limitation of the transformation method, the concept of "heterogeneous models" [27] 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. Code segments written in low level languages are embedded in the appropriate parts of the high level components. The major advantages of this model are that existing codes can be reused and business code can be generated. However, this model has many disadvantages,

- The heterogeneous models mix high level models and low level code segments together and make the design difficult to understand.
- The heterogeneous models exacerbate maintenance difficulty because the changes in the high level models may lead to changes in the embedded code

segments. If the high level models are designed to be transformed to different platforms, code segments achieving the same functionality but supporting different platforms need to be added.

- The heterogeneous models neutralize MDA's benefits of portability and documentation.

In this sense, it is desirable to have a program synthesis technique that is not tightly coupled with the high level models and would not affect the benefits of MDA. Our AI planning and component-based synthesis method discussed in Section 3.2 can achieve this goal.

3.2 AI Planning and Component-Based Synthesis

Raising the level of abstraction and increasing the level of reuse have been proven to be the right way to develop software systems [22]. Our AI planning and component-based code synthesis approach closely follows this principle by integrating AI planning techniques with Component-Based Software Development (CBSD) methods. Specifically, AI planning is a problem solving technique that works on high level abstractions of actions. The problem solving process is declarative, i.e., users only need to focus on specifying the initial and goal conditions and the AI planner helps generate a plan leading the system from the initial state to the goal state.

Another reason that makes AI planning appealing is that it can overcome some limitations of the deductive code synthesis method [20], which was once regarded as the answer to code synthesis. Similar to AI planning, deductive code synthesis also enables users to work on high level specifications. Code can be generated as a by-product of the proof by the theorem prover. The first limitation is that the deductive code synthesis process may not terminate. Actually, this is a problem inherent in any deductive methods [29]. When the theorem prover runs longer than expected, it is not possible to infer whether no solution exists or whether the prover needs more time to finish the proof. The second limitation is that it is difficult for the deductive code synthesis methods to generate loop constructs. Even a short iterative program has been proven to be difficult to reason about [12]. The FIP planner is not subject to these limitations. FIP enhances classical Graphplan [1], which is guaranteed to terminate regardless of whether a plan exists or not. Also, FIP is designed to support the generation of loop and conditional constructs in its procedural plans. All of these make FIP a full-fledged technique for automated code synthesis.

To increase the level of reuse, CBSD techniques can be used to achieve the goal. CBSD is designed to use existing software components as building blocks to construct larger applications. This approach can help lower the overall development cost and reduce the development time. However, software developers face a steep learning curve to grasp under what conditions the components can be used, the ways the components can be composed together, and all the constraints on the usages of the components. The code pattern technique [13][14] is designed to overcome the problems facing CBSD and is applied to our automated code synthesis approach.

In Section 3.2.1, we briefly introduce the background knowledge of AI planning and the FIP planner. In Section 3.2.2, we introduce the CBSD approach using code patterns. In Section 3.2.3, we discuss how to integrate FIP and code patterns together to achieve automated program synthesis.

3.2.1 FIP

We first briefly define the terminologies that are used in this paper.

Definition 1 (Action). Traditionally, an action in AI planning is defined as a triple, $a = \langle pre(a), add(a), del(a) \rangle$, where $pre(a)$ is the precondition of the action a; $add(a)$ is the post-condition achieved by the action a; and $del(a)$ is the delete effect that is no longer valid after the execution of the action a.

Definition 2 (Planning Problem). A planning problem is defined as a triple $P = \langle s_0, g, O \rangle$, where s_0 is the initial condition of the planning problem; g is the goal to be achieved; and O is a set of actions.

Definition 3 (Plan). Given a planning problem $P = \langle s_0, g, O \rangle$, a plan is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ that leads the system from the initial condition s_0 to the goal state g.

The reason that we have developed our own AI planner instead of using existing techniques is two-folds. First, existing planning techniques are not sufficiently expressive. The majority of AI planners are limited to deterministic domains and deal with only sequential planning, i.e., the actions in the generated plan are organized in a sequential manner. These planners are called classical planners. Each action is deterministic, i.e., the application of the action brings the system from the current state to a single other state. For example, a robot uses its arm to move a block from position *A* to position *B*. For classical planners, the effect of this action can always be predictable. As long as the robot can hold the block, it will definitely move the block to the expected destination. However, in reality, due to mechanical constraints, the robot's arm may drop the block during the moving process. Therefore, classical planning techniques make some impractical assumptions about the real world. For requirement elicitation, [11] points out that requirements engineers tend to set up goals and make some assumptions that are too idealistic. These assumptions are either not achievable or are very likely to be violated. Hence, to model the real world with more precision, it is desirable to require AI planners to be able to deal with nondeterministic actions.

Second, some efficiency and scalability problems have been reported for existing AI planners. There have been research works on nondeterministic planning domains, in which actions can have multiple nondeterministic effects. MBP [3] and Kplanner [12] are two such examples. However, as reported in [10], the CPU time of MBP may grow exponentially as the size of the planning problem grows. Kplanner suffers from the scalability problem due to its inherent mechanism of trying different loop bounds to generate the final plan.

Based on these reasons, we have developed FIP that can deal with nondeterministic actions and achieve highly efficient planning. FIP is based on planning graph [1]. It decomposes a nondeterministic action into a set of classical actions and conducts the planning process in two phases. In the first phase, a weak plan [3] is generated. The plan is weak because it only indicates one possible path leading to the goal. In this plan, only the ideal situations generated by actions are included. It is actually the optimistic shortest path leading to the goal. Based on this weak plan, FIP deals with the effects that are omitted in the first phase and generates a complete plan in the second phase. The search for a complete plan is conducted based on the shortest path along the weak plan. Hence, the overall search distance is optimal. In addition, the

planning graph is not a complete state space. It only contains states that are derivable from the initial conditions. Thus, the search space is much smaller than those of MBP and Kplanner. All these factors have made FIP a powerful and efficient planner qualified for dealing with practical problems.

3.2.2 Code Pattern

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 pre- and post-conditions $\{P, R\}$. The functionality of a pattern *p* can be represented as $\{P\}cp\{R\}$.

NAME	GetJDBCDBConnection
INTENT	Establish the connection with the database
CONTEXT	JAVA/JDBC
SOLUTION	1. Load JDBC driver; 2. Establish the DB connection

CODE TEMPLATE	Code_template
	Interface
	IN: String driver;
	String dbURL;
	String userName;
	String pwd;
	OUT: Connection con;
	End_interface
	Body
	try {
	Class.forName(driver);
	} catch(java.lang.ClassNotFoundException e) {
	System.err.print("ClassNotFoundException: ");
	System.err.println(e.getMessage());
	}
	
	try {
	con = DriverManager.getConnection(dbURL, userName,
	pwd);
	} catch(SQLException ex) {
	System.err.println("SQLException: " + ex.getMessage());
	}
	
	End_body
	Constraint
	Pre: Known(driver) and Known(dbURL) and
	Known(userName) and Known(pwd)
	Post: Known(con)
	End_constraint
	End_code_template

Fig. 6. Code pattern example for JDBC

For example, in the surprise test management system, database operations for storing, retrieving, and managing screeners' records are necessary. Code patterns can be used to capture the typical usages of the JDBC components as well as their interactions. In Fig. 6, a simple code pattern about how to obtain a JDBC DB connection is defined. A code pattern consists of an interface, a pattern body, and a constraint section. The pattern interface contains pattern parameters which are used to customize the pattern. Pattern parameters are also called ports. Three types of ports are possible, namely, input ports, output ports, and input/output ports. An input port is a data

source, an output port is a sink, and an input/output port can be either a source or a sink depending on the pattern context. For this particular example, there are four input ports and one output port. The precondition specifies the condition under which the code pattern can be applied while the post-condition indicates the effect achieved after the execution of the code template body.

Four code pattern composition operations, including one instantiation operation (Map) and three functional operations (Concatenate, Invert, and Splice) have been formally defined for glue code synthesis. The instantiation operation, *map*, is used to instantiate a pattern to obtain a concrete code segment. For example, "driver = sun.jdbc.odbc.JdbcOdbcDriver; dbURL = jdbc:odbc:AirTravel; …" can be used to instantiate the code template in the code pattern body in Fig. 6 to obtain a segment of concrete code. The concatenate operation is used to connect two or more code patterns together sequentially to form a flow of data or actions. The invert operation obtains a code pattern that performs the inverse operation of the original pattern. The splice operation joins two code patterns together according to their internal loop constructs. It interleaves the internal code of the two code patterns and merges code frames inside the loop constructs.

The code pattern approach is especially attractive for large enterprises because they may already have a substantial repository of existing software systems and, hence, seldom need to construct a new system from scratch. Code patterns can be used to record typical usages of code segments that have been proven to be correct and are repetitively used in the system construction. This kind of reuse is an effective way to save cost and time. When the number of code patterns grows large, they can be organized in a code pattern repository for future use.

3.2.3 The Integration of AI Planning and Code Pattern

As discussed above, the integration of AI planning and code pattern can raise the level of abstraction and increase the level of reuse in system construction. As shown in Fig. 7, the AI planning and code pattern based automated synthesis system consists of two major parts, namely, the prototype Code Pattern Integration System (CPIS) [14] and the Automated Planning System (APS).

The CPIS at the top portion of Fig. 7 consists of a code pattern repository, a graphical user interface, a code pattern parser, and a code pattern composer. The code pattern repository stores all the code patterns. The GUI interface is presented to enable the system users to add, retrieve, and edit code patterns from the repository. The code patterns in the repository come from two major sources, i.e., patterns that are input from the GUI interface and the composite patterns generated by the code pattern composer.

The code pattern parser is responsible for checking the validity of the code template. It is implemented based on JavaCC and supports multiple programming language grammars, e.g., C++, JAVA, etc. The errors in the code template of the code pattern can be detected when it is loaded into the CPIS.

The code pattern composer supports the pattern operations, namely, map, concatenate, invert, and splice, to compose code patterns. The system users work on the code patterns in the repository and use the pattern operations to compose composite patterns to achieve semi-automated synthesis of the glue code. The productivity would be increased greatly if the code pattern operations can be automated.

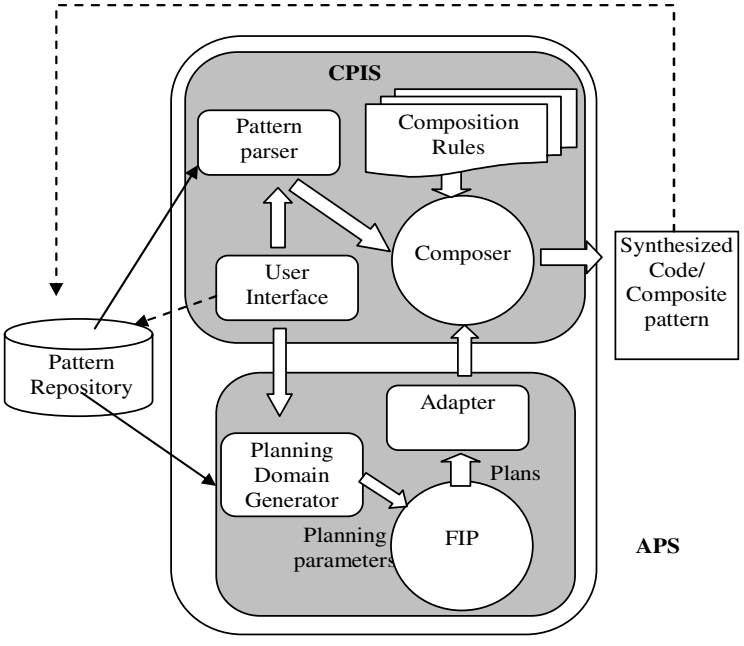

Fig. 7. Architecture of the code synthesis system

The automated planning system (APS) at the bottom portion of Fig. 7 is the core of the whole code synthesis system and can achieve the goal of automating the code pattern operations. It consists of three major parts, namely, planning domain generator, the FIP planner, and the plan adapter. As shown in Definition 4, a code pattern has a constraint portion consisting of pre-/post-conditions. The formalism of code pattern makes it naturally fit in the definition of AI planning actions. Hence, code patterns are modeled as planning actions. The planning domain generator serves as the bridge between code pattern repository and the APS and models code patterns as a planning domain. [5] also presents details about how code pattern operations are modeled in the planning system to facilitate plan generation as well as code synthesis. Given a planning problem, the AI planner, FIP, works on the actions that are derived from the code patterns and generates parameterized procedure-like generic reusable plans, i.e., *procedural plans*. The procedural plans are translated into preprocessed patterns by the plan adapter and fed to the code pattern composer to synthesize composite code patterns.

It should be emphasized that this program synthesis system is not limited to code patterns. The underlying components could be any components that can be abstracted with pre/post-condition constraints. For example, web services are well suited to the proposed architecture as shown in Fig. 7. The synthesis system is also an open system in which different component-based synthesis techniques may exist together. In this case, there will be multiple adapters for the AI planner to translate the generated plans to different underlying systems.

The input to the program synthesis system as shown in Fig. 7 is the planning problem, which is defined as a triple $P = (s_0, g, O)$, where s_0 is the initial condition, g is the

goal to be achieved, and O is the set of planning actions. As O is generated from the code pattern repository by the planning domain generator, the users just need to specify s_0 and g without having to know the details about how to achieve g.

3.3 The Integration of MDA and AI Planning and Component-Based Synthesis

As shown in Fig. 5, the code generated through transformation contains only code structures, which include the definitions of classes and operations, and/or the implementation of static getter/setter operations that are derived from the private attributes. The dynamic aspects of the system still remain to be completed. Therefore, we need a technique that is able to generate business code and complete some of the dynamic aspects of the system.

On the other hand, our AI planning and component-based code synthesis approach can generate business code based on the underlying code patterns. If it is integrated with MDA, it could greatly increase the productivity by generating the business code for the system dynamics.

To conduct the integration, we analyze MDA and its modeling process. UML is the de facto modeling language for MDA. However, UML is not good at modeling dynamic (or behavioral) parts [9]. The introduction of OCL 2.0 mitigates this problem and provides more choices for constructing high quality models. OCL is a formal modeling language that can be used to express conditions (pre-/post-conditions and invariants) and build software models. It is defined as an assistant language for UML. Hence, the combination of UML 2.0 and OCL 2.0 is the key to make the integration successful.

Specifically, pre-/post-conditions on operations can be used to express the system dynamics [9]. Formally, they can be expressed with a pair (P, R) representing the pre-condition and post-condition, respectively. As discussed in Section 3.2.3, the input to the AI planning and component-based code synthesis system is also a pair, (s_0, g), where s_0 is the initial state and g is the goal. The similarity of the two pairs (P, R) and (s_0, g) strongly suggests that the constraints on the operations can be formulated as a planning problem. Specifically, P is treated as the initial condition s_0 and R represents the goal g. The code synthesis system takes the input and generates the final code to fill in the body of the operation if the planning problem is solvable. The generated code is correct and is guaranteed to achieve the goal due to the following reasons:

(1) Code pattern is formally designed. Its functionality is expressed by the constraints, i.e., pre-/post-condition as shown in Definition 4.
(2) The code template in the code pattern is a proven solution to a recurring problem.

The AI planning and component-based code synthesis system tries to generate code for each operation based on its pre-/post-conditions. As shown in Fig. 2, the final code comes from three sources, namely, MDA, automated code synthesis system, and the developers. The multiple ways to automate the code synthesis could greatly speed up the development process and make the proposed prototyping method more practical. Moreover, the generated code (from MDA and code synthesis system) is correct and has good quality. This is another favorable factor for the proposed rapid evolutionary prototyping approach.

3.4 Analysis

Our rapid program synthesis approach (MDA + AI Planning + Component-based synthesis) has the same advantages as the heterogeneous models [27] discussed in Section 3.1, i.e., reuse the existing code to achieve the business code generation. However, our method is not subject to the disadvantages of the heterogeneous models.

First, the rapid program synthesis approach is not coupled with any high level models and does not hurt the MDA hierarchy. MDA, AI Planning, and component-based synthesis techniques can be seamlessly integrated together. Second, our approach does not complicate the maintenance process. The change of high level models will not result in maintenance burdens. Code can be regenerated along with the infrastructure code when the transformations are executed between different levels. Third, the rapid program synthesis approach does not hurt the MDA's benefits of portability and documentation. The generation of system dynamics is parallel in the transformation between PIM to PSM and from PSM to infrastructure code. In addition, the rapid program synthesis approach does not make any changes in the PIM. Thus, the PIM can still fulfill the function of high-level documentation that is needed for any software system [9].

4 Requirements Elicitation Via Prototyping

Based on the aforementioned advanced rapid program synthesis technique, we propose a prototyping approach that is intended to combine the advantages of the rapid and evolutionary prototyping. The rapid program synthesis technique ensures that the prototype can be developed rapidly and with good quality. In addition, the proposed prototyping approach implements the requirements regardless of whether they are poorly understood or well understood. In other words, the proposed approach will not be subject to the limitations of classical rapid prototyping and evolutionary prototyping.

Although the rapid program synthesis approach discussed in Section 3 can rapidly generate the correct code, it cannot generate a complete system fully. Manually composed code accounts for a certain portion in the prototype as shown in Fig. 2. In order to ensure that this portion of code does not compromise the prototype's quality, we apply a technique, baseline, that is similar to the operational prototyping approach [4]. A baseline corresponds to a well built prototype, in which the software is developed with production quality and only well understood requirements are included.

For the well-understood requirements, the standard MDA development cycle [9] (as shown in the left portion of Fig. 8) is followed to implement these requirements in a high quality manner. This is equivalent to the evolutionary prototyping. Then, a baseline is set up to record that the implemented prototype is of production quality. There is no poor quality code in the prototype within the baseline.

In the next step, the end users are trained to operate the prototype. This process may inspire them to clarify some of the unclear requirements or to come up with new requirements. The users may also experience some problems. All of these observations will be collected and sent to the requirements and design engineers.

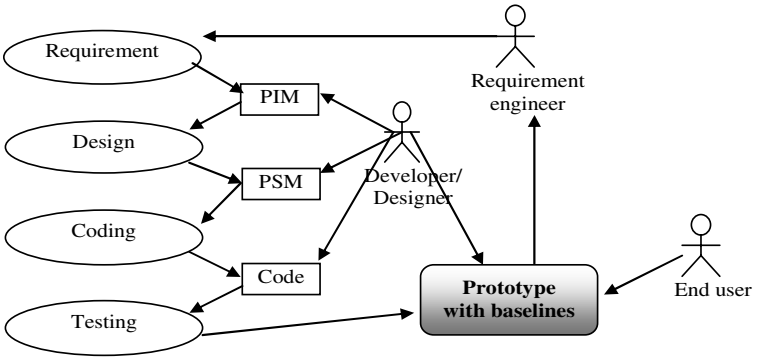

Fig. 8. Rapid evolutionary prototyping

For requirements that are critical but poorly understood, the throwaway (rapid) prototyping method is applied to implement them over the baseline. The implementation should be done as quickly as possible to illustrate the functionality to the users. After the users have identified the true requirements from the quick-and-dirty prototype, the portion that is not included in the baseline will be thrown away. The cost is not too high as the traditional throwaway prototyping is because the rapid program synthesis approach can help generate code to speed up the development. The automatically generated code is much cheaper than code that is manually composed.

Then, the MDA development cycle is repeated and the newly identified and well understood requirements are implemented with good quality to set up the next baseline. Afterwards, the implemented prototype is sent to the users again and the same processing steps are repeated if needed.

4.1 Discussion

With the rapid program synthesis technique presented in Section 3, the throwaway prototype over the baseline can be implemented quickly. Although the generated code is correct, it may not produce the desired effect because the requirements themselves may not be correct. The code generated based on the incorrect requirements are not valuable and must be discarded.

The proposed rapid evolutionary prototyping approach has some similarities with operational prototyping, e.g., the use of baselines, the way of handling poorly understood requirements, etc. There are also some major differences. First, operational prototyping uses conventional development strategies to implement requirements that are well understood. In contrast, the proposed approach applies the MDA development strategy, in which the design becomes part of the development and more stakeholders (software, design, and requirements engineers, etc.) are actively involved in the development process. The development focus has shifted from code to PIM, which is a higher level of abstraction. The artifacts that are created during the development process are models.

Second, our rapid program synthesis technique makes the proposed prototyping approach a practical method for requirements elicitation. It greatly speeds up the development process and ensures the quality of the generated code. The development cost

can be reduced as well. This is especially true for implementing poorly understood requirements. The code of the quick-and-dirty prototype would be discarded after the requirement elicitation. But the development cost of the automatically generated code is relatively cheaper than that of the manually composed code. If the generated code accounts for a large portion of the prototype, it implies that the cost can be greatly reduced.

4.2 Example

We still use the "surprise" test management system as an example to illustrate how the proposed prototyping approach works. As shown in Fig. 9, the surprise test management system consists of three subsystems, namely, user management, test management, and analysis subsystems.

The user management subsystem is a conventional information management system, which includes the major use cases of "add", "edit", "retrieve", and "delete" users. The requirements regarding the user management subsystem are well understood.

The test management subsystem is the key part of the system. It includes the major use cases of "test generation", "record test results", "retrieve tests", and "decision making". The requirements regarding this subsystem are not completely clear. Specifically, the users may have conflicting requirements about test generation. They cannot make an agreement about how screeners are chosen for the test and how inspectors are identified to conduct the test.

The analysis subsystem includes the major use cases of "report generation", "records evaluation", and "trend analysis". This subsystem is supposed to use the data mining technologies to implement the requirements. But the specific requirements regarding this subsystem are poorly understood as well. The users still do not have a clear idea about exactly what kind of reports needs to be generated and how the series of results could help in trend analysis.

The proposed prototyping approach implements the system in the following steps. First, the well understood requirements are implemented. Hence, the user management subsystem and part of the test management subsystem (e.g., record test results, retrieve tests, and decision making) are implemented in a quality manner. As the implementation relates to conventional database application development, abundant code patterns that capture the typical usages of JDBC and other database related operations exist for facilitating the code generation. A baseline is set up for the prototype indicating the completion of the well understood requirements.

Then, the users can operate the prototype and the problems found are collected. As the test generation function is absent from the prototype, the users cannot have a complete experience about the overall system. Hence, the test generation is critical but poorly understood. It is implemented by using the rapid (throwaway) prototyping strategy to illustrate its functionality over the baseline. In the prototype, the screeners are chosen at random for the test and the inspectors are identified in a round robin manner from the set of available candidates whose schedules are clear at the time when the test is supposed to be conducted.

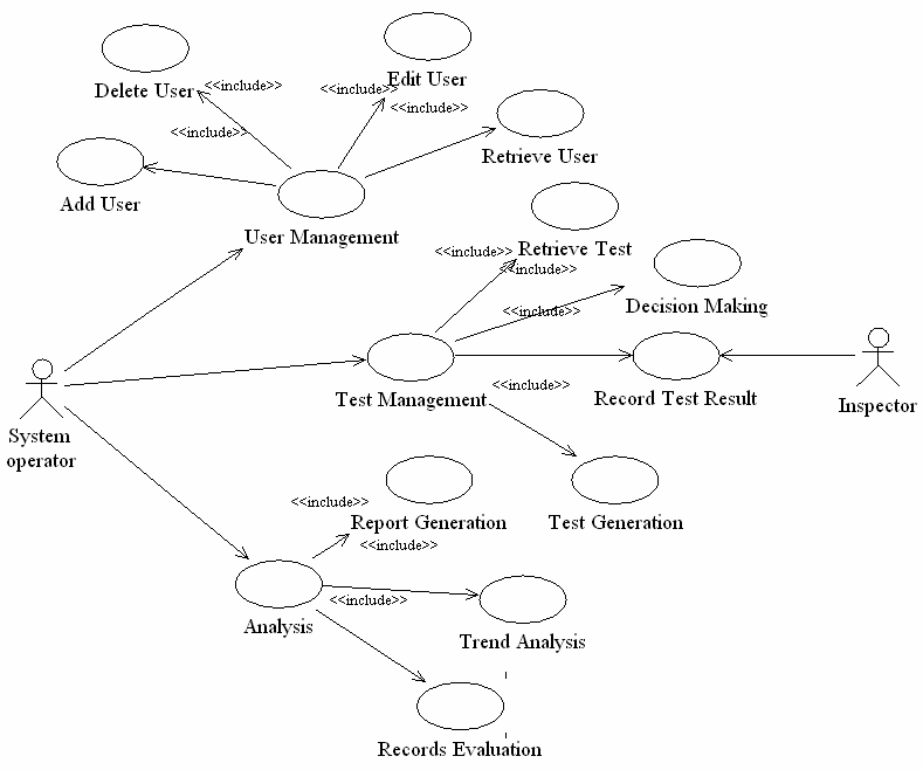

Fig. 9. Surprise test management system use case

Suppose the users reach an agreement after operating the prototype. They agree that the screeners who are most likely to fail the test should be chosen for the test and the way of identifying inspectors in the prototype is considered acceptable. After the requirements become clear, the part that is not included within the baseline is discarded, including the code for the inspector identification. Afterwards, the MDA development cycle is followed to implement the newly clarified requirements with good quality. Machine learning technique, e.g., Markov Decision Processes (MDPs) [26], can be used to predict which screeners are more likely to fail the test. Code patterns are available for solving the typical recurring problems in the MDPs area. The rapid program synthesis approach can help accelerate the prototype development. Then, another baseline is created and the prototype is sent to the users again.

For the analysis subsystem, the users can hardly come up with concrete requirements before they really touch the system and have the historical data available. This is especially amenable to the prototyping approach. After the prototype is in good shape to achieve the design goals for test management, the users will have better understandings about what they really need. The clarified requirements are incrementally fed back and implemented with the help of the proposed prototyping approach.

5 Conclusions

It is estimated that to fix a defect found during requirements engineering costs two orders of magnitude less than to fix the same defect after the product has been delivered [2][18]. This asserts the essential role of requirements engineering in the software development process. Software prototyping is an important requirements elicitation technique that can help find defects at an early stage and, thus, make the project more likely to succeed.

We have proposed a model-driven development based prototyping approach for requirements engineering. It inherits the advantages of prototyping elicitations without the disadvantages, such as untraceable requirements and tendency of reusing previously written code fragments, etc., by applying model-driven development principles and advanced program synthesis techniques, in which MDA, AI planning, and component-based software development techniques are seamlessly integrated together. The proposed approach is a rapid evolutionary process that iteratively refines the requirements, design, and implementation and yields high quality systems with the help of the novel rapid program synthesis technique.

References

1. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence 90, 281–300 (1997)
2. Boehm, B.: Industrial software metrics top 10 list. IEEE Software 4(5), 84–85 (1987)
3. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. Artificial Intelligence 147(1–2), 35–84 (2003)
4. Davis, A.: Operational prototyping: A new development approach. Software 9(5), 70–78 (1992)
5. Fu, J., Bastani, F.B., Yen, I.: Automated AI planning and code pattern based code synthesis. In: ICTAI 2006, pp. 540–546 (2006)
6. Fu, J., Bastani, F.B., Ng, V., Yen, I., Zhang, Y.: FIP: A fast planning-graph-based iterative planner, Technical Report. UTDCS-03-08, UT-DALLAS (2008)
7. Harmain, H.M., Gaizauskas, R.: CM-Builder: A natural language-based CASE tool. Journal of Automated Software Engineering, 157–181 (2003)
8. http://www.andromda.org/
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
10. Kuter, U., Nau, D.: Forward-chaining planning in nondeterministic domains. In: Proceedings of the National Conference on Artificial Intelligence (AAAI-2004), pp. 513–518 (2004)
11. Lamsweerde, A., Letier, E.: Handling obstacles in goal-oriented requirements engineering. TSE 26(10), 978–1005 (2000)
12. Levesque, H.: Planning with loops. In: Proc. of the IJCAI 2005 Conference, Edinburgh, Scotland (2005)
13. Liu, J., Bastani, F.B., Yen, I.: Code Pattern: An approach for component-based code synthesis. In: Proceeding of the 7th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, FL, pp. 330–336 (2003)

14. Liu, J., Bastani, F.B., Yen, I.: A formal foundation of the operations on code Patterns. In: The International Conference on Software Engineering and Knowledge Engineering, Taipei, Taiwan, Republic of China ((2005)
15. Luqi: Knowledge-based support for rapid software prototyping. IEEE Expert 3(4), 9–18 (1988)
16. Luqi: Software evolution through rapid prototyping. Computer 22(5), 13–25 (1989)
17. Luqi, Berzins, V., Yeh, R.: A prototyping language for real time software. IEEE Transactions on Software Engineering 14(10), 1409–1423 (1988)
18. Luqi, Guan, Z., Berzins, V., Zhang, L., Dloodeen, D., Coskun, C., Pueett, J., Brown, M.: Requirements document based prototyping of CARA software. International Journal on Software Tools for Technology Transfer 5(4), 370–390 (2004)
19. Luqi, Kordon, F.: Advances in Requirements Engineering: Bridging the Gap between Stakeholders' Needs and Formal Designs. In: Paech, B., Martell, C. (eds.) Monterey Workshop 2007. LNCS, vol. 5320, pp. 15–24. Springer, Heidelberg (2008)
20. Manna, Z., Waldinger, R.: Fundamentals of deductive program synthesis. IEEE Transactions on Software Engineering 8(18), 674–704 (1992)
21. Mcclendon, C.M., Regot, L., Akers, G.: The Analysis and Prototyping of Effective Graphical User Interfaces (October 1996)
22. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Reading (2004)
23. Nuseibeh, B., Easterbrook, S.: Requirements engineering: A roadmap. The Future of Software Engineering. In: 22nd International Conference on Software Engineering, pp. 35–46. ACM-IEEE (2000) (special issue)
24. Object Management Group: MDA Guide: Version 1.0.1, OMG document omg/03-06-01 (2005)
25. Overmyer, S.L.V., Rambow, O.: Conceptual modeling through linguistics analysis Using LID. In: 23rd international conference on Software engineering (2001)
26. Puterman, M.L.: Markov Decision Processes. Wiley, Chichester (1994)
27. Selic, B.: Model-driven development: Its essence and opportunities. In: 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC), pp. 313–319 (2006)
28. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley, Chichester (2006)
29. Stickel, M.E., Waldinger, R.J., Chaudhri, V.K.: A Guide to SNARK (2005), `http://www.ai.sri.com/snark/tutorial/tutorial.html`
30. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Reading (2003)