Data Lifecycle Verification Method for Requirements Specifications Using a Model Checking Technique

Yoshitaka Aoki, Hirotaka Okuda, Saeko Matsuura
Graduate School of Engineering and Science, Shibaura Institute of Technology,
Saitama-City, Japan
{nb12101, ma11043, matsuura}@shibaura-it.ac.jp

Shinpei Ogata
Department of Computing, Shinshu University
Nagano-City, Japan
ogata@cs.shinshu-u.ac.jp

Abstract—**A key to success in developing high quality software is to define valid and feasible requirements specifications to enable the production of high quality source code with minimal extra development rework. To provide invariable services to all users at any time, the data lifecycle functions of create, read, update, and delete (CRUD) are essential for handling persistent data. These important operations should, therefore, be verified at the start of development. In UML2UPPAAL, a support tool that verifies such functions, requirements specifications written in UML are transformed into finite-state automata in UPPAAL. UML2UPPAAL enables developers with knowledge of UML to benefit from the UPPAAL model checking tool without requiring UPPAAL knowledge. This paper proposes a data lifecycle verification method that uses the UPPAAL model checking tool and focuses on CRUD operations in the requirements analysis phase.**

Keywords—Verification; Model Checking; Requirements Specifications; UML; CRUD

I. INTRODUCTION

A key to success in developing high quality software is to define valid and feasible requirements specifications to enable the production of high quality source code with minimal extra development rework. Requirements specifications should have a verifiable form to guarantee their adequateness and completeness in the early stages of development. However, uncertain and ambiguous software requirements often make it difficult for developers to describe requirements specifications in verifiable form during their analysis. Although it offers insufficient verification formalization, the Unified Modeling Language (UML) [1] is a useful, common tool for formalizing requirements specifications while enabling their description in natural language. We propose a method of model-driven requirements analysis [2][3] using UML. Our method automatically generates a web user-interface prototype from a UML requirements analysis model written in activity diagrams and class diagrams. This method enables developers to confirm the validity of input and output data for each page and page transition on the system by directly operating the prototype.

Model checking has been a favored technique for improving reliability in the early stages of software development. We therefore propose a verification method in which the requirements analysis model written in UML meets essential properties that any system should meet by using the UPPAAL model checking tool [4].

Enterprise systems typically must provide invariable services to many users at a given time; therefore, the data lifecycle functions of *create, read, update,* and *delete* (CRUD) are essential for handling persistent data. These important operations should be verified at the start of development. This paper proposes a method of verifying these essential CRUD functions by using the UPPAAL model checking tool.

In UML2UPPAAL, a support tool that verifies such functions, requirements specifications written in UML are transformed into finite-state automata in UPPAAL. UML2UPPAAL enables developers with knowledge of UML to benefit from the UPPAAL model checking tool without requiring UPPAAL knowledge. This paper proposes a data lifecycle verification method that uses the UPPAAL model checking tool and focuses on CRUD operations in the requirements analysis phase.

The rest of the paper is organized as follows. Section II discusses the problems of verifying requirements specifications in terms of formalization and the applicability of model checking techniques. Section III outlines our verification method. Section IV explains UML2UPPAAL, which can be used to implement our method and support developers who have insufficient knowledge of model checking techniques. Section V describes case studies and the effectiveness of our method.

II. REQUIREMENTS SPECIFICATIONS VERIFICATION PROBLEMS

A. *Problems of Writing Requirements Specifications*

The primary cause of the failure of IT projects is often attributed to inadequate and incomplete requirements analysis [5]. IEEE Std 830 [6] has been recognized as a standard of requirements specifications construction. Although developers may create requirements specifications according to the standard, it is often difficult for them to fully address the interrelationship among all document components to achieve adequateness and completeness. This is because the initial requirements are written in a natural language and screen images, which are not related to the other documents in a verifiable way. Formal specification techniques, such as the Vienna Development Method (VDM) [7] and the B-method [8], provide promising approaches to formalizing requirements specifications. However, uncertain and ambiguous requirements often make it difficult for developers to describe requirements specifications in a verifiable form at the start of analysis.

UML is a promising tool for formalizing requirements specifications because of its popularity among development teams. However, step-by-step formalization is insufficient for verification. We therefore propose a verification method in which our requirements analysis model written in UML is specified as a formal description in stages by using a model checking technique.

B. Problems with Applying a Model Checking Technique

Model checking is regarded as an effective technique for improving reliability in the early stages of software development. A model checking tool uses temporal logic to model a system as a network of automata extended with integer variables, structured data types, user defined functions, and channel synchronization. Based on these properties, a system model and query expressions can be defined to specify properties to be checked. When the specified properties are not satisfied, the tool provides counterexamples that show how the properties can be falsified. The simulator helps detect the cause of defects by tracing the processes in which the counterexamples occur.

Model checking is a technique for automatically verifying a model by exhaustively checking all paths to detect properties that developers are often apt to overlook. However, because the path and state formulas should be defined by items that are used in the model, it is typically difficult for developers to define an appropriate model and formulas at all times.

Path formulas can define properties such as reachability, safety, and liveness. Reachability means that the specified state will be reached at some point in time. Safety means that something bad will never happen. Liveness means that something expected will eventually happen. State formulas need defining by expressions related to several process IDs or variables of the state.

In our requirements analysis model, a use case is defined by an activity diagram comprised of several sequences of user and system actions representing normal flows and exceptional flows in the use case. Data used in the activity diagram are classified by class diagrams for the system input/output and entity data, as shown in Figure 1. Based on a lifecycle of these entity data and actions related to them, we specify a requirements analysis model in strict descriptions to enable the automatic defining of query expressions for the model to verify the specified safety properties.

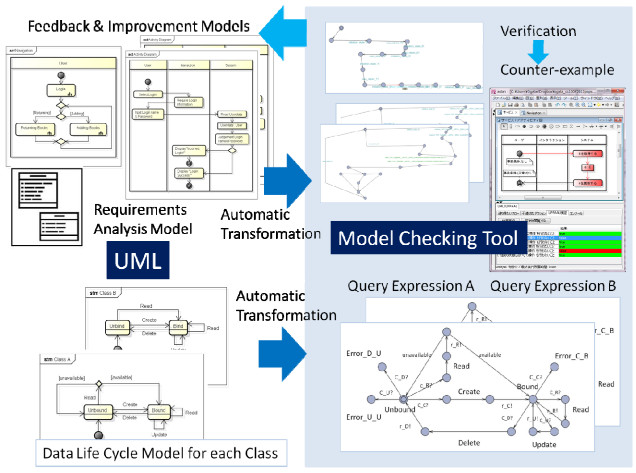


Figure 1. Verification Method using UML and a Model Checking Tool

Figure 1 shows an outline of our verification method using a model checking tool. Semi-formal UML models are automatically transformed into a network of finite automata and query expressions; these are used for producing counterexamples when the requirements analysis model has defects relating to the data lifecycle of all classes.

III. DATA LIFECYCLE VERIFICATION METHOD

A. Requirements Specifications in UML

We have proposed a method of model-driven requirements analysis using UML [2][3]. We analyze use cases and functional requirements of services. In particular, because end user needs obviously appear within the interaction between a user and system, our method proposes to clearly model the interaction.

More specifically, we identify business processes as use cases from the following questions.

- Based on the specified business rules, what types of input data and conditions are required to correctly execute the use case?
- To observe the business rule, what types of conditions should be required when the use case is not executed? Moreover, how should the system handle these exceptional cases?
- According to the above conditions, what types of behaviors are required to execute the use case?
- What types of data are outputted by these behaviors?

Based on the above questions, both business flow and business entity data, which are required for executing the target business tasks, are defined in UML by activity diagrams and a class diagram.

An activity diagram specifies not only normal and exceptional action flows but also data flows that are related to these actions. An action is defined by an action node; data is defined by an object node being classified by a class that is defined in a class diagram. Accordingly, these two kinds of diagrams enable specifications of business flows in connection with the data. This is one of the features of our method on how to use activity diagrams and class diagrams. In particular, the interaction between a user and system includes requisite various flows and data on user input, conditions, and output to correctly execute a use case.

The second feature of our method is an activity diagram that has three types of partitions: user, interaction, and system. These partitions enable ready identification of the following activities: user input, interaction between a user and system caused by the conditions for executing a use case, and the resulting output.

The third feature is a prototype consisting of web pages written in HTML that are automatically generated from the above two diagram types. The prototype, a kind of model of the final product, enables end users to clearly and easily confirm the requisite business flows in connection with the data. The generated prototype describes the required target system, except for the user interface appearance and internal business logic processing. Additionally, the prototype enables developers to confirm and understand the correspondence between their models and the final system. Developers define two kinds of diagrams based on requirements analysis from different viewpoints, such as action flows, data flows, and structure. The automatically generated prototype enables them to easily understand the consistency between their models and the target system. To facilitate a full understanding of the correspondence between each diagram and the target system, a prototype can be generated in the requirements analysis phase whenever the developer needs it. The requirements analysis model is defined using the modeling tool Astah [9].

When clients confirm that the prototype satisfactorily represents their requirements, the confirmation represents client validation that the specifications meet their expectations from an actual usage perspective.

B. Data Lifecycle Model Definition in UML

It is important that developers can verify the specifications to confirm their feasibility. To accomplish this objective, developers must confirm that a sequence of actions and data flows within the system partition of the activity diagrams can produce the expected output data from the specified input. The system-side prototype helps developers confirm the following facts.

- Input data being defined by the user can be transformed into entity data of the system.
- The existing entity data that should be generated via the other use cases and above-mentioned entity data can generate the target output data following the specified action sequence.

As a result of these considerations, developers can effectively define entity classes. During this confirmation process, it is not difficult for developers to adjust actions in the system partition in accordance with CRUD actions.

An object node has the role of a variable that stores an instance being created by the *create* action in the activity diagram. The object of the verb in the CRUD function description usually relates to the object node. The verbs shown in Table I represent CRUD functions in an activity diagram. For example, CRUD functions can be represented as “create an object,” “delete the object,” “update the object,” and “get an object.” The target object node for *create* and *read* is located at the next node of the action, as shown in Figure 2.

TABLE I. VERBS FOR CRUD ACTIONS

Action Type	Verbs
Create	create, generate
Read	read, get, search
Update	update, add, insert, change
Delete	delete

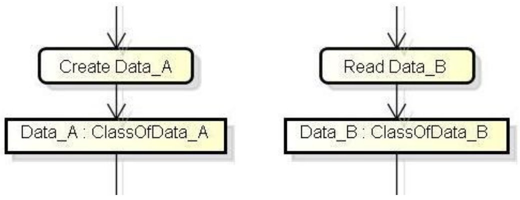


Figure 2. Relation between Object and Verb in *Create* and *Read* Actions

As a result of these adjustments, a sequence of actions in the activity diagram represents the state of changes of system entity data over the whole service by these CRUD actions.

On the other hand, entity data itself should satisfy the data lifecycle constraint of a class. For example, to *update* or *delete* an object, the object node must be bound in advance to some concrete instance object.

These essential properties of entity data are defined by using a state machine diagram in UML, as shown in Figure 3. A state machine diagram consists of several states that must be distinguished and transitions among these states. Each transition is executed by an event, if necessary, when some guard conditions are satisfied.

In this paper, we intend to distinguish whether or not each entity data is binding to an instance object, so that the system defined by the whole of activity diagrams can guarantee the correct execution of use cases in accordance with the CRUD data lifecycle.

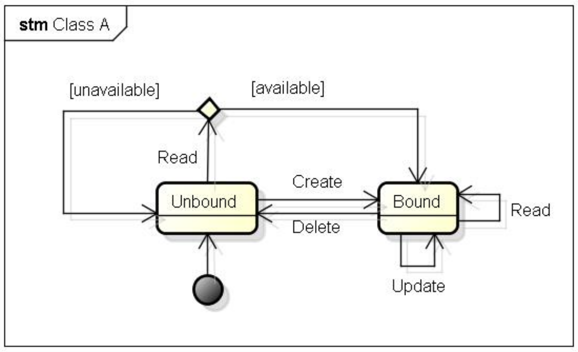


Figure 3. CRUD Data Lifecycle of a Class

Figure 3 shows a basic data lifecycle of a class. A state machine is defined for each class and named by the class. The initial state of each instance object in Class A is "unbound." After *create*, the state is changed to "bound." If the state is "bound," the instance object can accept actions such as *update*, *read,* and *delete.* If the state is "unbound" and the instance object can be obtained by the *read* action, the state is changed to "bound." If it cannot be obtained, the state remains "unbound."

However, classes do not always have the same data lifecycle. The basic state machines are therefore modified to meet the specified class. For example, if all instance objects in a class have read-only status, the data lifecycle is modified, as shown in Figure 4.

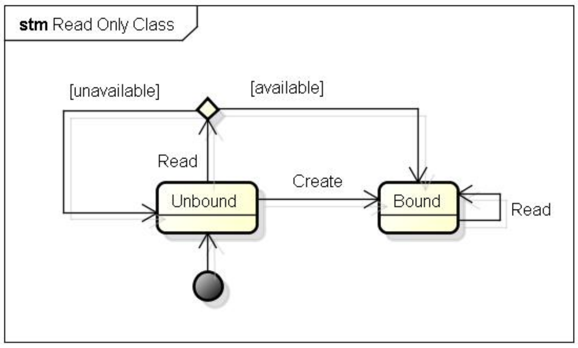


Figure 4. CRUD Data Lifecycle of a Read- Only Class

The states that should be distinguished within a class must be specified by guard conditions on the flow in the corresponding activity diagram. Figure 5 shows guard descriptions when the *read* action is executed.

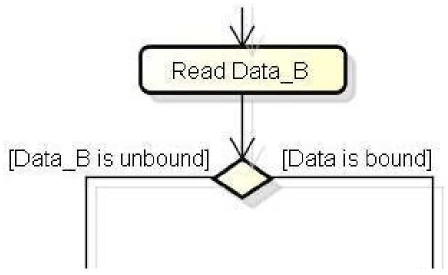


Figure 5. Guard Descriptions in an Activity Diagram

As a result, a type of CRUD action term in an activity diagram equals an event in a state machine diagram of the object in the action. A guard description equals a sentence of "<<An object>> is <<a state>>" on the control flow in the activity diagram, as shown in Tables II and III.

TABLE II. CORRESPONDENCE BETWEEN ACTION AND EVENT

Verbs of Action for an Object in Activity diagram	Event in State Machine Diagram of all Objects in a Class
create, generate	Create
read, get, search	Read
update add insert change	Update
delete	Delete

TABLE III. CORRESPONDENCE BETWEEN GUARD AND STATE

Guard description for an <<Object >>in Activity diagram	State in State Machine of all <<Objects>> in a Class
<Object> is unbound	Unbound
<<Object>>t is bound	Bound

As mentioned earlier, verifiable forms can be incrementally introduced to the requirements specifications in UML. At this point, it can be verified whether or not there are contradictions between all service flows defined in all activity diagrams and the data lifecycles of all entity objects appearing in the system partition of the activity diagram.

C. Verification Method

This section explains how to transform the requirements analysis model and specified data lifecycle models from UML to UPPAAL, and how to generate the query expressions.

The UPPAAL model consists of several locations and transition arrows among them, as shown in Figure 6. A location expresses a state of the system, and the transition arrow indicates several conditions named *Guard* and a sequential processing event during it named *Update*. In Figure 6, START, LOC1, and LOC2 are names of each location. "i1==0" and "i1>0" are *Guard* expressions and "flg=true" and "flg=false" represent *Update* expressions.

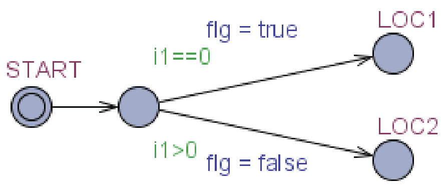


Figure 6. Basic Components of the UPPAAL Model

The requirements analysis model includes all use cases of a target system and a navigation model to integrate them. Figure 7 shows the entire structure of transforming UML models into UPPAAL models and query expressions.

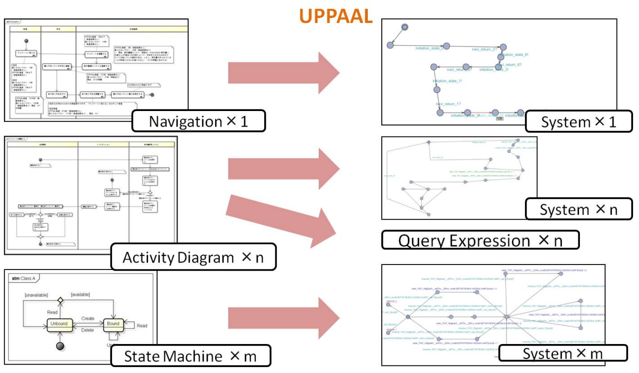


Figure 7. Transformation of UML to UPPAAL

Firstly, each activity diagram corresponding to a use case is transformed into one system model in UPPAAL. In this model, a CRUD action is transformed into a transition of three locations with channel synchronization.

Figure 8 shows the correspondence between a flow in an activity diagram and a transition in a UPPAAL system model. All nodes, such as action, object, decision, merge, start, end, and so on, are transformed into locations in UPPAAL. The control flow and data flow are each transformed into transitions, except for CRUD actions.

For example, the *create* action is transformed into a transaction sequence of three locations. The first location represents a pre-state of calling the *create* action, and the second location represents a state of creating. The third location represents a post-state of creating. The first transition flow has a synchronization channel named "c_C!" and the second transition flow has a synchronization channel named "r_C?" "c" denotes "call" and "r" denotes "return," respectively.

These synchronization channels synchronize with other channels in a system being transformed from a state machine diagram of the corresponding object class. In this case, the corresponding object means that it is an objective word of the *create* action.

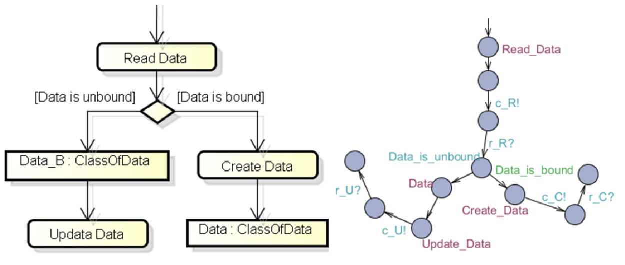


Figure 8. Activity Diagram and the Corresponding UPPAAL Model

A state machine diagram in Figure 3 is transformed into the UPPAAL model in Figure 9.

Two states are transformed into the locations named "Unbound" and "Bound," respectively. Each transition is transformed into a transition sequence of three locations, in the same way that CRUD actions are transformed. However, the channel in this model fires by calling from the system relating to the activity diagram. In this case, the first transition is fired by the corresponding object channel "c_C!" After creating, the channel "r_C!" synchronizes the channel "r_C?" in the caller system.

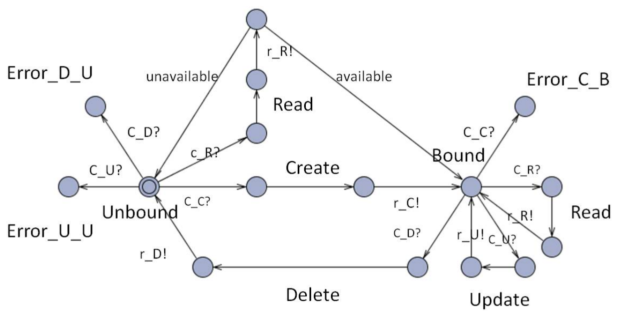


Figure 9. Transformed Data Lifecycle

A state machine diagram defines the data lifecycle of a class by using restricted actions, such as CRUD. It specifies all behaviors that all objects in the class can perform. That is, it specifies negative properties that should never happen. The state machine diagram in Figure 3 specifies that the *update* and *delete* operations should not be applied to it if an object is unbound.

The state that will never happen is then designated in the transformed UPPAAL model, as shown in Figure 9. Error_D_U, Error_U_U, and Error_C_B denote the impossible states. These states are defined for every object appearing in all activity diagrams.

As a result, we can automatically define query expressions on safety property in accordance with these models as follows.

A[] not Error_D_U_<<Object>>
A[] not Error_U_U_ <<Object>>
A[] not Error_C_B_<<Object>>

Because all names of locations in the UPPAAL model are defined by the original nodes in the activity diagrams, query expressions for the reachability property can also be automatically generated.

A navigation model integrates all activity diagrams according to the pre-conditions and post-conditions, which are a combination of several labels being added to the start or end nodes in each activity diagram. According to these conditions, all system models transformed from the activity diagrams are integrated as a UPPAAL model.

IV. UML2UPPAAL

UML2UPPAAL is a support tool that implements the above-mentioned verification method. Figure 10 shows the architecture of UML2UPPAAL, which is implemented as a plugin of the UML modeling tool Astah.

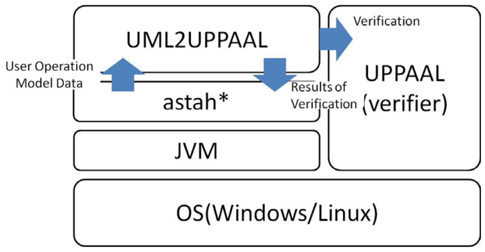
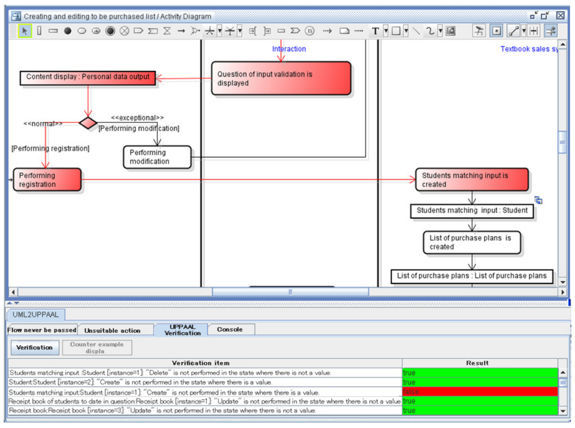


Figure 10. UML2UPPAAL Architecture


Creating and editing to be purchased list / Activity Diagram
Interaction
Textbook sales sy
Content display : Personal data output
Question of input validation is displayed
<<normal>>
<<exceptional>>
[Performing modification]
[Performing registration]
Performing modification
Performing registration
Students matching input is created
Students matching input : Student
List of purchase plans is created
List of purchase plans : List of purchase plans
UML2UPPAAL
Flow never be passed
Unsuitable action
UPPAAL Verification
Console
Verification
Counter example displa

Verification item	Result
Students matching input :Student [instance=1]: "Delete" is not performed in the state where there is not a value.	true
Student:Student [instance=2]: "Create" is not performed in the state where there is a value.	true
Students matching input:Student [instance=1]: "Create" is not performed in the state where there is a value.	false
Receipt book of students to date in question:Receipt book [instance=1]: "Update" is not performed in the state where there is not a value	true
Receipt book:Receipt book [instance=3]: "Update" is not performed in the state where there is not a value.	true

Figure 11. UML2 UPPAAL

After defining a requirements analysis model using Astah, a developer can verify it with the same tool environment. As shown in Figure 11, the result of the verification is presented by highlighting the defective items in the model. The results of executing the query expressions are shown in the lower part of the screen. During this work, developers are not required to have knowledge of UPPAAL; they only need knowledge of UML to use UML2UPPAAL and obtain the benefits of the UPPAAL model checking tool.

V. Case Studies

A. *Outline of Case Studies*

We conducted a case study to evaluate the effectiveness of our method. First, five graduate students modified their UML models of the following four systems. The modifications were performed to maintain the rule of the descriptions of CRUD actions in an activity diagram. The first two systems are the currently running systems in our university. Table IV shows the scale of each model.

- Group work support system for project-based learning (PBL): GWSS
- Learning Management System: LUMINOUS
- University co-op text book sales system: COOP
- Laboratory library management system (two types): Library1, 2

TABLE IV. Scale of Models

Model	COOP	GWSS	LUMINOUS	Library1	Library2
Number of Classes	110	162	58	33	45
Number of Attributes	387	157	112	91	125
Number of Use case	7	8	8	5	6
Number of Actions	391	315	183	119	138
Average of Cyclomatic Numbers	22.9	28.2	14.9	15	12.3
Average of Number of Flows and Actions	106.5	85.7	56.1	64.3	58
Average of Number of Model elements	65.5	60.5	39.5	43.5	39.57

B. *Verification Results*

Next, the experimenters defined data lifecycle models for the specified entity data by using state machine diagrams. Having minimal knowledge of UPPAAL, they could find 83 defects in their models. The main defects found by this experiment were:

- Ten omissions of defining proper guard conditions against the nondeterministic property on the *Read* action.
- Two mistakes involving the impossible actions of *Update* and *Delete* being applied to unbounded objects.
- One mistake caused by complicated flows in which some objects could not create during the service because the position of the *Create* action was incorrect.

A navigation model is typically useful for generating a prototype system so that a user can operate it simultaneously with the final product. However, there were some cases in our experiment in which the pre-conditions and post-conditions affected the state of the object. As a result, at times there were objects of the same class but from a different data lifecycle in the activity diagram. A data lifecycle was defined for each class; however, it was necessary to adjust the state machine for the effects of the pre-conditions on the target object.

Moreover, there were instances when a complicated use case caused defects in the data lifecycle because loops occurred in an activity diagram at least two times.

It therefore must be considered that the association between classes affects the data lifecycle.

VI. Related Work

Several researchers have proposed respective formal approaches to verifying specified features in the early stages of software development. Yatake [10] verified that all object states satisfy the invariant conditions between collaborative object behaviors by using a theorem-proving system. However, it requires a large quantity of strict definitions to clarify all the actions and data relating to the invariant. It is generally difficult to perform such strict work during a changeable phase, such as requirements analysis.

It is important to conduct stepwise specifications refinement by checking several verifiable features in the early stage of software development. Choi [11] proposed a verification method of the consistency between the page transition specification on a web-based system and the flow chart defining the process streams. We have also proposed a common verifiable feature in enterprise systems, such as the conditions for CRUD of entity data. Moreover, we can automatically generate the query expressions.

Achenbach [12] compared the abstraction techniques in various model checking tools and applied these tools to real-world problems. For example, the open/close behavior of the file I/O stream was modeled using the transition between states such as open, close, and error. This approach is very similar to ours. However, unlike our approach, this paper did not discuss the method on the assumption that the requirements specifications have been validated by the clients.

Several researchers have proposed support methods to effectively use model checking tools [13][14][15].

Trcka [13] proposed a method to verify the nine predefined query expressions using a Petri net, which can specify behaviors such as read, write, and delete. This study may be similar to our method. However, because query expressions depend on the properties specified by state machine diagrams, our method can be extended to verify the other properties.

Several studies [14][15] have proposed a method to transform UML models into process or protocol meta language (PROMELA) for using the model checking tool SPIN. However, because developers need to directly operate the model checking tool, they are required to have knowledge of both UML and SPIN. It is convenient that UML2UPPAAL can be used only with knowledge of UML.

VII. CONCLUSION

This paper proposed a verification method of requirements specifications in UML in the beginning phase of development using a model checking technique. UML2UPPAAL is a support tool for verifying the entity data lifecycle by transforming requirements specifications written in UML into finite automata in UPPAAL. A key attribute of UML2UPPAAL is that developers with knowledge of UML can benefit from the UPPAAL model checking tool without having UPPAAL knowledge. We are planning to apply our method to verify a security policy for requirements specifications [16] based on the Common Criteria [17] for Information Technology Security Evaluation, which is an international standard (ISO/IEC 15408) for computer security certification.

ACKNOWLEDGMENT

This work has been conducted as a part of the "Research Initiative on Advanced Software Engineering in 2012" supported by the Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency (IPA), Japan.

REFERENCES

[1] UML, http://www.uml.org/,[retrieved: July, 2013].

[2] S. Ogata, and S. Matsuura, "A UML-based Requirements Analysis with Automatic Prototype System Generation," Communication of SIWN, Vol.3, Jun. 2008, pp.166-172.

[3] S. Ogata. and S. Matsuura, "A Method of Automatic Integration Test Case Generation from UML-based Scenario," WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS, Issue 4, Vol.7, Apr 2010, pp.598-607 .

[4] UPPAAL, http://www.uppaal.com/, [retrieved: July, 2013]..

[5] Standish Chaos Report, http://blog.standishgroup.com/

[6] IEEE Computer Society, IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830 (1998).

[7] VDMTools, http://www.vdmtools.jp/ , [retrieved: July, 2013].

[8] K. Lano and H. Haughton, "Specification in B: An Introduction Using the B Toolkit", Imperial College Press, 1996

[9] astah*, http://www.change-vision.com/,[retrieved: July, 2013].

[10] K. Yatake, T. Aoki and T. Katayama, "Collaboration-based verification of Object-Oriented Models", Computer Software, Vol.22, No.1, 2005, pp.58-76. (in japanese)

[11] E. Choi, T. Kawamoto, and H. Watanabe, "Model Checking of Page Flow Specification", Computer Software, Vol.22, No.3, 2005, pp.146-153. (in japanese)

[12] M. Achenbach and K. Ostermann, "Engineering Abstractions in Model Checking and Testing", Source Code Analysis and Manipulation, Proc. of .SCAM '09.,2009, pp.137-146

[13] N. Trcka, Wil M.Aalst, and N.Sidorova., "Data-Flow Anti-Patterns: Discovering Dataflow Errors in Workflows," Proc. of the CAiSE 2009, 2009, pp.425-439.

[14] P. Bose, "Automated translation of UML models of architectures for verification and simulation using SPIN," Proc. of the ASE, 1999, pp.102-109.

[15] L. Jing, L. Jinhua, and Z. Fangning, "Model Checking UML Activity Diagrams with SPIN," Proc. of the CiSE 2009, 2009, pp.1-4.

[16] A. Noro and S. Matsuura, "UML based Security Function Policy Verification Method for Requirements Specification", Proc of 2013 IEEE 37th International Conference on Computer Software and Applications, 2013, pp.832-833.

[17] Common Criteria, "CC/CEM v3.1 Release4", http://www.commoncriteriaportal.org/cc/,[retrieved: July, 2013].