Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

User-Friendly GUI in Software Model Checking

Shoichi Yokoyama, Haruhiko Sato, and Masahito Kurihara
Graduate School of Information Science and Technology,
Hokkaido University,
Sapporo, JAPAN
{yokosho,haru}@complex.eng.hokudai.ac.jp, kurihara@ist.hokudai.ac.jp

***Abstract*—Model Checking is an automatic technique for verifying finite-state concurrent systems such as communication protocols and sequential circuit designs. It has a number of advantages over traditional approaches to this problem that are based on simulation, testing, and deductive reasoning. Model checking tools are, however, not widely introduced into industry, and one of the reasons is that they are tricky and difficult to use for engineers. In this paper, we extend the functions of Java PathFinder, a software model checker to verify executable Java bytecode programs, and propose a graphical user interface with a high degree of usability. Our GUI depicts the state transition graphs of not only a whole program but also each thread as a result of verification. Users can get much information through the GUI, for example, the internal states of a program and the correspondence relation between the graphs, with interactive mouse operation. One of the key features of our GUI is a variable-value-based graph abstraction that allows users to focus upon an aspect they are interested in. Our GUI also has an intuitively easy-to-use interface for users to input linear temporal logic (LTL) formulae as a program specification based on the Specification Pattern System.**

***Index Terms*—Model Checking, GUI, Java Pathfinder**

I. Introduction

Today, various kinds of large-scale and complicated computer systems such as concurrent systems are widely used in applications where failure is unacceptable: electronic commerce, highway and air traffic control systems, medical instruments, and so on. Clearly, as the involvement of such systems in our lives increases, the need for their reliability is critical and the burden for ensuring their correctness also increases.

The principal validation methods for complex systems are simulation, testing, deductive verification and model checking. Simulation and testing usually involve providing certain inputs and observing the corresponding outputs. These methods can be a cost-efficient way to find many errors. However, in these methods, errors detected in concurrent systems are not reproducible. Moreover, checking all of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible. Deductive verification normally uses axioms and proof rules to prove the correctness of systems. However, use of this method is rare because this is a time-consuming process that can be performed only by experts who are educated in logical reasoning and have considerable experience.

On the other hand, model checking [1] has attracted attention recently, which is an automatic technique for verifying finite state concurrent systems. The procedure normally uses an exhaustive search of the state space of the system to determine if the system satisfies some specifications. Given sufficient memory spaces, the procedure will always terminate with a correct answer. Moreover, in case of negative result, this technique can provide users with an error trace. However, model checking technique is not widely introduced into industry, although it has a number of advantages over other methods as seen above. One of the reason is that model checking tools are tricky and difficult to use for engineers.

To make model checkers easy to use, we present in this paper a graphical user interface (GUI) with them with a high degree of usability. We extend the functions of Java PathFinder [7], a software model checker, and propose input and output GUI functions which are intuitively understandable for users.

The paper is organized as follows. In Section II, we briefly describe model checking technique, and review some model checkers as related works. We then describe the Specification Pattern System [8], which is a basic concept of our specification input GUI, in Section III. In Section IV, we present our user-friendly GUI, and in Section V, we evaluate our GUI by comparing it with traditional ones. Section VI contains the conclusion and possible future work.

II. Model Checking

A. Model Checking Technique

Model Checking is one of the formal verification methods toward finite-state concurrent systems. This method treats a system as a state transition graph called a Kripke structure, and verifies the properties required to satisfy on the system with exhaustive search. Let AP be a set of atomic propositions which expresses properties of a system. A Kripke structure M over AP is a 4-tuple $M = \langle S, S_0, R, L\rangle$, where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; $R \subseteq S \times S$ is the transition relation, which must be total (i.e., for every state $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions to be true in that state.

The properties in model checking are typically expressed as formulae of temporal logic. If a verification result is negative, this method can output an error trace that is an execution path leading to an error state.

B. Temporal Logic

In model checking, the properties that the system must satisfy are usually given in temporal logic, which can assert how the behavior of the system evolves over time. There are some kinds of temporal logic: CTL, LTL, ACTL and so on. In

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

this section, we describe Linear Temporal Logic (LTL), which is the basic of our property description GUI.

LTL is temporal logic describing events along a single computation path. To build up expressions describing properties, LTL uses not only atomic propositions and Boolean connectives such as conjunction, disjunction, and negation, but also temporal operators, which describe properties of a path through the computation tree. There are five basic operators: **X**, **F**, **G**, **U** and **R**. Each operator expresses the following property:

$\mathbf{X}\phi$:	ϕ holds in the second state of the path. (ne**X**t time)
$\mathbf{F}\phi$:	ϕ holds at some state on the path. (in the **F**uture)
$\mathbf{G}\phi$:	ϕ holds at every state on the path. (**G**lobally)
$\psi\mathbf{U}\phi$:	There is a state on the path where ϕ holds, and at every preceding state on the path, ψ holds. (**U**ntil)
$\psi\mathbf{R}\phi$:	ϕ holds along the path up to and including the first state where ψ holds. But ψ is not required to hold eventually. (**R**elease)

Let AP be the set of atomic propositions. The syntax of LTL formulas is given by the following rules:

- If $p \in AP$, then p is a LTL formula.
- If f and g are LTL formulas, then $\neg f \mathrm{C} f \vee g \mathrm{C} f \wedge g \mathrm{C} \mathbf{X} f \mathrm{C} \mathbf{F} f \mathrm{C} \mathbf{G} f \mathrm{C} f \mathbf{U} g \mathrm{C} f \mathbf{R} g$ are LTL formulas.

A path in a Kripke structure $M = \langle S, S_0, R, L \rangle$ is an finite sequence of states, $\pi = s_0, s_1, ...$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. We use π^i to denote the *suffix* of π starting at s_i. If f is a LTL formula, $\pi \models f$ means that f holds along path π. The relation $\models$ is defined inductively as follows (assuming that f and g are LTL formulas):

1. $\pi \models p$	$\Leftrightarrow$	$p \in L(s_0)$.
2. $\pi \models \neg f$	$\Leftrightarrow$	$\pi \not\models f$.
3. $\pi \models f \wedge g$	$\Leftrightarrow$	$\pi \models f$ and $\pi \models g$.
4. $\pi \models f \vee g$	$\Leftrightarrow$	$\pi \models f$ or $\pi \models g$.
5. $\pi \models \mathbf{X}f$	$\Leftrightarrow$	$\pi^1 \models f$.
6. $\pi \models \mathbf{F}f$	$\Leftrightarrow$	there exists a $i \geq 0$ such that $\pi^i \models f$.
7. $\pi \models \mathbf{G}f$	$\Leftrightarrow$	for all $i \geq 0$, $\pi^i \models f$.
8. $\pi \models f\mathbf{U}g$	$\Leftrightarrow$	there exists a $i \geq 0$, such that $\pi^i \models g$ and for all $0 \leq j < i$, $\pi^j \models f$.
9. $\pi \models f\mathbf{R}g$	$\Leftrightarrow$	for all $j \geq 0$, if for every $i < j$ $\pi^i \not\models f$ then $\pi^j \models g$.

C. Model Checkers

SPIN [4] is one of the most famous model checkers. It verifies properties described as CTL or LTL of a system written with SPIN-specific process description language called *Promela*. SPIN provides a GUI application called XSPIN, which includes editors of Promela language and temporal logics. XSPIN can show the static image of the state transition graph of a whole system as verification result.

LTSA (Labelled Transition System Analyzer) [5] and UPPAAL [6] are model checkers that have characteristics in their GUI. LTSA verifies properties described as LTL of systems written with LTSA-specific system description language called Finite State Process (FSP). As verification result, it can show not only the state transition graph of a whole system, but also the ones of each concurrent module separately. Moreover, its animator function can display the dynamic behavior the system as animation.

In UPPAAL, users can build up the state transition graph of the system visually and graphically with mouse operation. And in addition to the state transition graph, it can display the sequence diagram of the system as verification result.

In many model checkers including above three, the system must be modeled in each tool-specific language. In contrast to those, Java PathFinder (JPF) [7] is a software model checker for Java bytecode. JPF is a Java virtual machine that executes a program not just once, but theoretically in all possible way, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths. It also shows the static image of the state transition graph of a program as verification result.

III. Describing Formal Specification

As mentioned in the previous section, in model checking, specifications to be verified are written in temporal logics. However, describing such a formal specification is a complicated work and needs high-level skills, which causes the delay in the penetration of the technique.

In this section, we describe Specification Pattern System [8], which is one of the solutions to the problem. We use this concept as the basis of our property description GUI. As a related work, we also introduce Prospec [9], which is a GUI application of Specification Pattern System.

A. Specification Pattern System

Specification Pattern System (SPS) [8] is a collection of generalized descriptions of commonly occurring requirements on the permissible state/event sequences in finite-state models. It describes the essential structure of some aspect of a system's behavior and provides expressions of this behavior in a range of common formalisms.

SPS defines eight property patterns (TABLE I) and five scopes (TABLE II). A scope is the extent of the program execution over which the pattern must hold. In SPS, a property is basically constructed by the combination of one property pattern and one scope. For example, "The OK button is enabled after the user enters correct data." can be specified by the combination of **Response** pattern and **Global** scope.

B. Prospec

Prospec (Property Specification) [9] is a tool that provides visual and textual guidance for specifying common properties of systems. This work builds on the SPS. Prospec provides a process for elicitation and specification of properties and

TABLE I
PROPERTY PATTERNS.

Occurrence Patterns	
Absence	being free of certain events/states. (Never)
Universality	containing only states that have a desired property. (Always)
Existence	containing an instance of certain events/states. (Eventually)
Bounded Existence	containing at most a specified number of instances of a designated state transition or event.
Order Patterns	
Precedence	an occurrence of the first is a necessary pre-condition for an occurrence of the second.
Response	an occurrence of the first, must be followed by an occurrence of the second. (Follows)
Chain Precedence	a scalable pattern. 1 stimulus - 2 responses, 2 stimuli - 1 response, ...
Chain Response	a scalable pattern. 1 cause - 2 effects, 2 causes - 1 effect, ...

TABLE II
SCOPES.

Global	the entire program execution.
Before	the execution up to a given state/event.
After	the execution after a given state/event.
Between L **and**	any part of the execution from one given state/event to another given state/event.
After L **until**	like between but the designated part of the execution continues even if the second state/event does not occur.

generates formal specifications in LTL, etc that can be used by a various tools such as theorem provers, model checkers, and runtime monitors.

IV. PROPOSED GUI

The purpose of our work is to create the GUI in model checking that the user can use easily and intuitively. To fit the purpose, we extend the GUI of JPF. JPF can directly treat the Java bytecode, therefore is easier to use than other model checkers that can treat the systems modeled only in their tool-specific languages. We consider that the difficulty to use model checkers mainly consists of two type of problems: one is a poor understandability of verification output, and another is a difficulty to input specifications of a system in formal descriptions such as LTL. Therefore, we propose an output interface that shows users a verification result in a intuitively-understandable way and is useful as a debugger, and a property description interface that users can build up specifications of a system without special knowledge.

A. Output Interface of Verification Result

As a verification result, JPF shows a static image of the state transition graph of a whole system. Our output interface is a Java application that shows not only it but also the state transition graphs of each thread that runs in the system (Fig. 1). Blue, green and red circle nodes (*state node*) in the graph represent a normal state, a successful end state, and an error state respectively. An orange rectangle node (*transition node*) between state nodes represents a transition. The graph layout is decided based on the hierarchical drawing algorithm of a general directed graph by Sugiyama et al. [10].

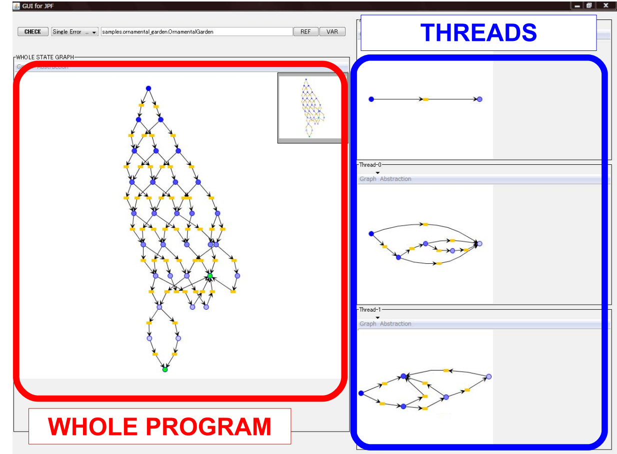

Fig. 1. Overview of proposed GUI.

When the user clicks a state node in a thread transition graph, the detailed information of the state, which is the contents of corresponding virtual machine stack, is displayed. On the other hand, clicking a transition node, users can know the detailed information on the transition, which includes the name of the bytecode last executed on the transition, the name of the thread executing the bytecode, the corresponding line in source code to the bytecode with the line number, and the file name of the source code.

We present two key GUI functions: a display function of correspondence relation between the state transition graphs, and an abstract function of a state transition graph based on a designated variable in the program.

The former is a function to show users mutual correspondence relation between the state transition graphs of a whole program and each thread. When a state node in the state transition graph of a whole program is clicked, the state nodes in the one of each thread corresponding to the clicked state are highlighted (Fig. 2). In reverse, when state node in the thread transition graph is clicked, the state nodes in the one of a whole program corresponding to the clicked state are highlighted.

The latter is a function to show users an abstracted state transition graph of a whole program. This abstraction is based on values of a variable in the program. The variable is designated by users before the verification with its dedicated interface. Using this function, the adjacent nodes at which the values of the designated variable are equal are abstracted as a one cluster. In the abstracted graph, corresponding to scaling operation with mouse wheel, the further users zoom in the graph, the more detailed structure of the graph users can know, and vice versa (Fig. 3).

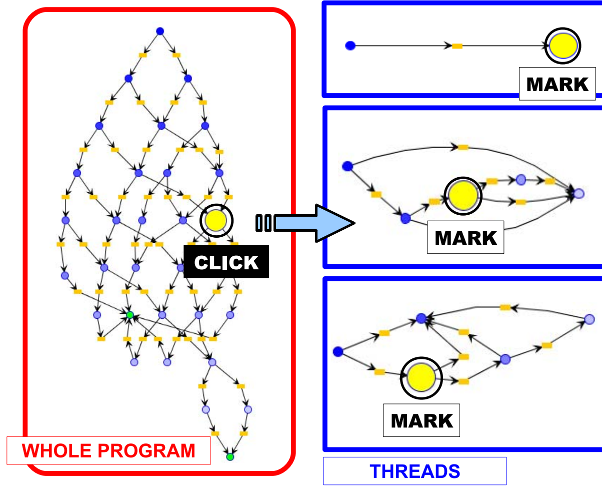

Fig. 2. The function to show correspondence relation between nodes in a whole transition graph and ones in each thread transition graph.

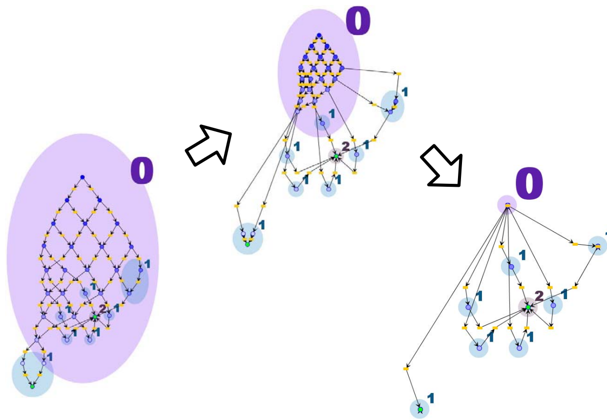

Fig. 3. The function to abstract a whole transition graph based on values of the variable designated by users.

B. Property Description Interface

We propose a user-friendly GUI that users can intuitively build up LTL formulae to be verified in JPF. We also implement a LTL verification function because JPF originally doesn't equip it. In our property description interface, users take two steps to build up a LTL property.

At a first step, users define atomic propositions that are used in LTL formulae. In this process, we propose a way to define them with the combination of primitive-type (boolean, byte, short, int, long, char, float) or String-type variables in the program, its specific values, and three operators we prepare (=, >, <). For example, the atomic proposition that is true when "an integer-type variable called i is 0" is defined by a combination of integer-type variable i, its value 0, and a = operator. We also present a user interface for users to process this step easily.

After the first step, users build up properties that are

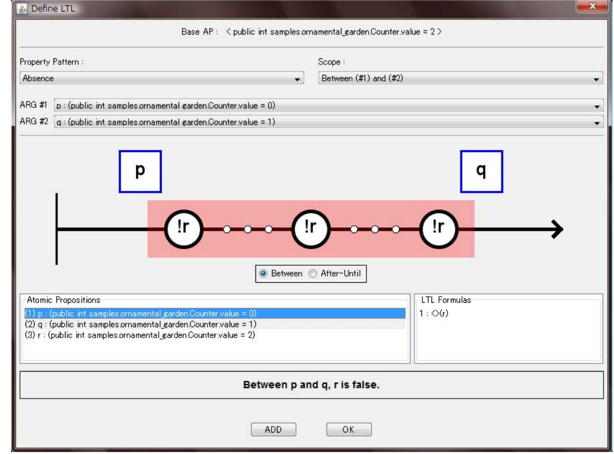

Fig. 4. Proposed GUI to describe LTL formulae.

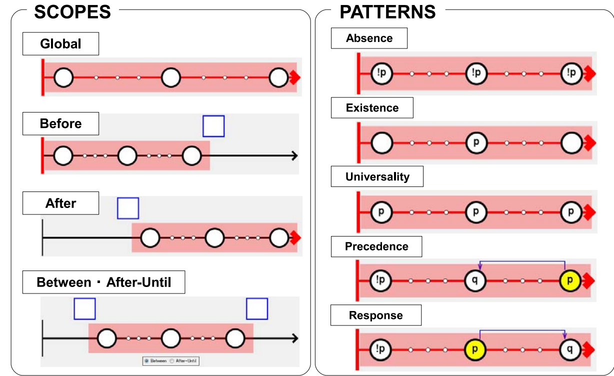

Fig. 5. The expressions of scopes and patterns in proposed GUI.

program specifications using the atomic properties defined in the first step. The overview of our GUI at this step shows on Fig. 4. The basic concept used in our interface is a SPS. We adopt 5 all scopes and 5 property patterns (Absence, Existence, Universality, Precedence and Response) in SPS. In this interface, users can build up a property through selecting a property pattern, a scope, and their arguments from combo boxes placed at the top of the window. Moreover, our interface has a function that users can build up a property intuitively with mouse operation on the diagram placed at the center of the window. The red rectangle (*score area*) in the diagram represents a scope of SPS. Users can interactively change this area with mouse drag and decide the scope. Each scope in our interface shows on the left-side diagram in Fig. 5. **Between** and **After-Until** scopes are distinguished by selecting a radio button. The argument of a scope (if any) is described in the square box on the shoulder of its area. The circles in a scope area represent states on an execution path, and the properties that holds at each state are described in them. The arguments of scope and property pattern in SPS can be specified with mouse operation through dragging an atomic proposition or

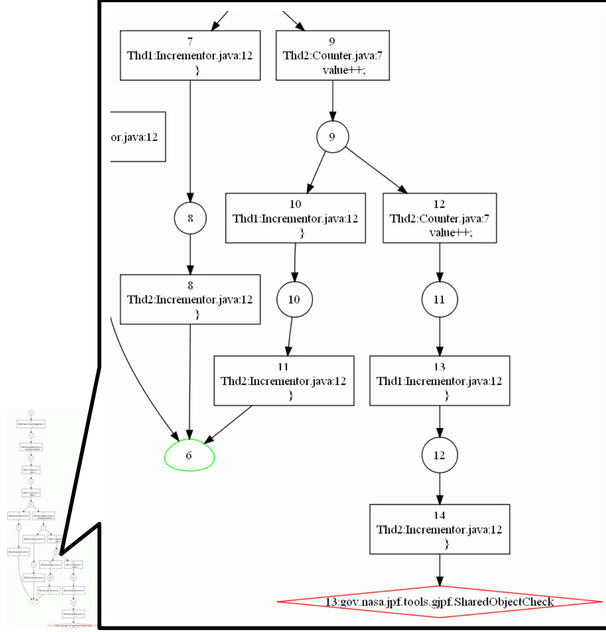

Fig. 6. The verification output of JPF.

LTL formula from the list of them placed at the bottom of the window, and drop it to each box or one of circles. According to the selected property pattern, the diagram of state circles change. The diagrams of each property pattern show on the right-side diagram in Fig. 5.

V. Evaluation

A. *Output Interface of Verification Result*

To evaluate usability of our output interface as a debugger, we consider a simple error-prone example program that has an asynchronous access to a shared variable in it. This example program has an integer-type shared variable called *value* whose initial value is 0, and each of two thread (INC_1, INC_2) increments the *value* once in the process without exclusive access control. Intuitively the *value* is expected to be 2 at the end of program, but actually it may be 1 because of the lack of exclusive access control. In the verification of this example program, we consider the terminal state where the *value* is not 2 to be an error.

As a result of verification, JPF detects the error and outputs a static image of the state transition graph of the whole program (Fig. 6). In this image, a green circle and a red diamond mean a successful terminal state and an error state respectively. The state 9 is a last branch point to them. Therefore we can see that, it is highly possible that the processes executed shortly after the state 9 are closely connected with the cause of the error. However, users can not identify the cause from this image, because this error arises from a bytecode-level problem but this image shows only the information of the source code. Moreover, it is difficult to intuitively understand each of the behaviors of two threads only from this image.

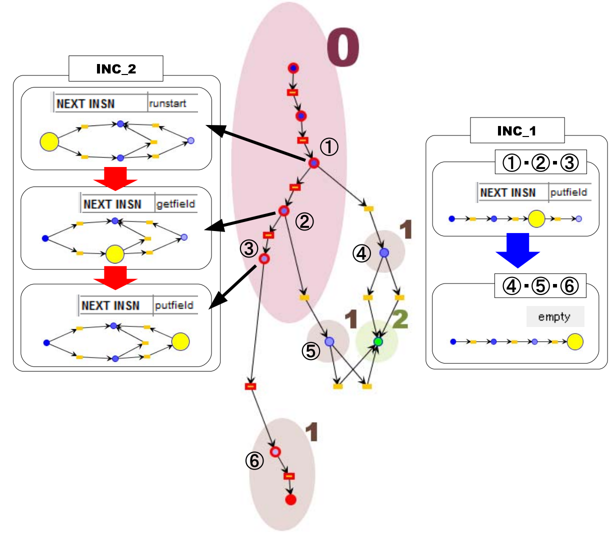

Fig. 7. The verification output of proposed GUI.

On the other hand, the verification result of our GUI shows on Fig. 7. The state transition diagram of the whole program placed at the center is abstracted based on the value of the variable *value*. In addition to this transition graph, our GUI shows the state transition graphs of each thread separately, which is placed at the both side. This function makes users easily and clearly understand the behaviors of each thread working in the program. Furthermore, the abstraction function makes the timing where the value of the designated variable changes clear, therefore users can specify the processes that may cause the error. Our output interface can show the bytecode-level information of the program. From Fig. 7, users can understand the cause of this error is that INC_2 get the old value of the variable *value* with the bytecode **getfield** between after INC_1 increments the value and before it returns the result with the bytecode **putfield**.

For comparison, the verification result of the program by LTSA shows on Fig. 8. The key function of LTSA is an animator, but our proposed GUI has a same function as it. In the state transition graph, all state nodes are just simply aligned, and the information about each transition is described on it. Therefore, it becomes too complicated to understand the behavior, if the program is large-scale. Moreover, the code translation task from an original program to a LTSA-specific FSP puts a burden on users and may cause mis-translations.

B. *Property Description Interface*

For example, we consider the property on a queue "Between after enqueue is executed and before the queue becomes empty, dequeue must be executed.". The LTL formula representing the property is as follows:

$$\mathbf{G}((\text{enque} \wedge \neg\text{empty}) \rightarrow (\neg\text{empty}\,\mathbf{U}((\text{deque} \wedge \neg\text{empty}) \vee \mathbf{G}\neg\text{empty})))$$

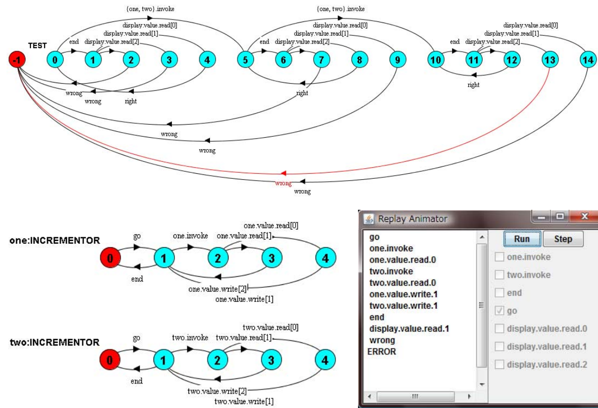

Fig. 8. The verification output of LTSA.

The property is not so complex but this LTL formula is too complicated for users without special knowledge to describe. On the other hand, the expression of the property in our interface shows on Fig. 9. This diagram is more intuitively understandable than the LTL formula, on which users can build up it with mouse operation.

Although our interface can not describe the all of property patterns in SPS because of the difficulty to express it graphically, [8] shows that most of general specifications can be described only by a few property patterns. Our interface supports all of the principal property patterns, therefore the lack of expressive power of properties compared to SPS is not a much serious problem.

VI. Conclusion and Future work

In this work, we presented user-friendly input and output GUI of software model checker. We extended Java PathFinder and proposed mainly three GUI functions: the function to show correspondence relation between nodes in the transition graph of whole program and ones in each thread transition graph, the function to show fine information inside a program, and the function to abstract a whole transition graph based on value of the variable designated by users. Moreover, we proposed a LTL-based property description GUI which users can use without knowledge on complicated logic. Also, we showed that these GUI functions enhanced usability of Java PathFinder as a debugger.

As future work, we plan to improve drawing performance of large-scale graphs and extend the expression power which property description interface can treat. Furthermore, we plan to evaluate our GUI by inquiry survey from users.

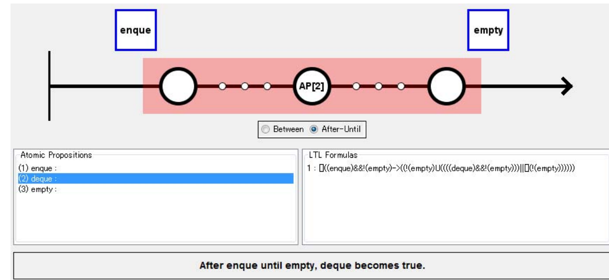

Fig. 9. The diagram representing the property in proposed GUI.

References

[1] E. M. Clarke, O. Grumberg and D. Peled: *Model Checking*, MIT Press, 2000.

[2] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda: "Model Checking Programs", *Automated Software Engineering Journal*, Volume 10, Number 2, 2003.

[3] M. B. Dwyer, G. S. Avrunin and J. C. Corbett: "Patterns in property specifications for finite-state verification", In *Proceedings of the 1999 International Conference on Software Engineering*, pp. 411-420, IEEE, 1999.

[4] G. J. Holzmann: *THE SPIN MODEL CHECKER*, Addison-Wesley Pub, 2003.

[5] J. Magee and J. Kramer: *CONCURRENCY: STATE MODELS & JAVA PROGRAMMING*, John Wiley & Sons Inc, 2006.

[6] G. Behrmann, A. David and K. G. Larsen: "A Tutorial on UPPAAL", In *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, pp. 200-236, Springer, 2004.

[7] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda: "Model Checking Programs", *Automated Software Engineering Journal*, Volume 10, Number 2, 2003.

[8] M. B. Dwyer, G. S. Avrunin and J. C. Corbett: "Patterns in property specifications for finite-state verification", In *Proceedings of the 1999 International Conference on Software Engineering*, pp. 411-420, IEEE, 1999.

[9] O. Mondragon, A. Q. Gates and S. Roach: "Prospec: Support for Elicitation and Formal Specification of Software Properties", *Proceedings of the 3rd Workshop on Runtime Verification*, pp. 1-22, Elsevier, 2003.

[10] P. Eades and K. Sugiyama: "How to draw a directed graph", *Journal of Information Processing*, Vol. 13, No. 4, pp. 424-437, 1990.