Innovations Syst Softw Eng (2009) 5:231–241
DOI 10.1007/s11334-009-0096-1

ORIGINAL PAPER

Software monitoring through formal specification animation

Hui Liang · Jin Song Dong · Jing Sun · W. Eric Wong

Received: 31 March 2009 / Accepted: 13 July 2009 / Published online: 5 August 2009

Abstract This paper presents a formal specification-based software monitoring approach that can dynamically and continuously monitor the behaviors of a target system and explicitly recognize undesirable behaviors in the implementation with respect to its formal specification. The key idea of our approach is in building a monitoring module that connects a specification animator with a program debugger. The requirements information about expected dynamic behaviors of the target system are gathered from the formal specification animator, while the actual behaviors of concrete implementations of the target system are obtained through the program debugger. Based on the information obtained from both sides, the judgement on the conformance of the concrete implementation with respect to the formal specification is made timely while the target system is running. Furthermore, the proposed formal specification-based software monitoring technique does not embed any instrumentation codes to the target system nor does it annotate the target system with any formal specifications. It can detect implementation errors in a real-time manner, and help the developers and users of the system to react to the problems before critical failure occurs.

H. Liang · J. S. Dong
School of Computing, National University of Singapore, Singapore, Singapore

J. Sun (✉)
Department of Computer Science,
The University of Auckland, Auckland, New Zealand
e-mail: j.sun@cs.auckland.ac.nz

W. E. Wong
Department of Computer Science,
University of Texas at Dallas, Texas, USA

1 Introduction

With the capabilities of detecting, diagnosing, and recovering from software faults, program monitoring provides additional defense against catastrophic software failure. It can be used as a complement to formal verification and software testing in which higher reliability of software systems can be achieved. Recently, there has been increasing attention from the research community to the development of techniques and tools for runtime monitoring of software systems [1–8]. For example, Java with Assertions (Jass) [3] extends Java to allow annotating Java programs with specification in the form of assertions such as method pre- and post-conditions, class invariants, and so on. A pre-compiler translates the annotated program into pure Java code. Compliance with the specified annotation is dynamically tested during runtime. jContractor [1,9] allows contracts to be associated with any Java classes or interfaces. Contract methods can be included directly within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class byte code and performs on-the-fly byte code instrumentation to enable checking of contracts during the program's execution. Spec# [2] is a Design-by-Contract extension of C#. The type system of C# is extended to include non-null types and checked exceptions.It also uses method contracts in the form of pre- and post-conditions as well as object invariants. With the Spec# compiler, the user can perform run-time checks for method contracts and invariants. ProTest [10] is an automatic test environment for B specifications. After generating a set of test cases, ProTest simultaneously performs animation of the B machine and the execution of the corresponding implementation in Java, and assigns verdicts on the test results.

In order to monitor software systems, some of the above-mentioned techniques add instrumentations to the target

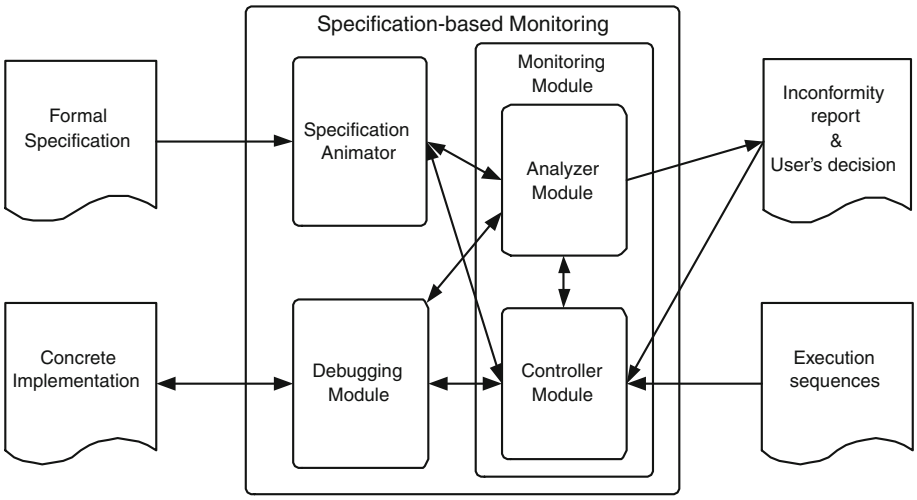

Fig. 1 Formal specification-based software monitoring system

program, while others annotate the concrete implementation with extra formal specifications to obtain dynamic information about the target programs. There are a few disadvantages of such approaches. First of all, adding instrumentation code is itself a difficult task involving all the complexities of programming. Moreover, it generally leads to changes in the program, which raises the possibility that through collecting information to analyze target system behavior, the monitoring system is actually altering that behavior of the target system. Finally, annotating the concrete implementation with extra formal specifications leads to the lack of separation between the concrete implementation of target systems and their high-level requirements specification.

In this paper,[1] we propose a novel formal specification-based software monitoring approach. As shown in Fig. 1, the key idea of our approach is to build a monitoring module that connects a specification animator and a program debugging module. Based on the information obtained from the specification animator and the debugging module, the monitoring module, which consists of a controller and an analyzer, dynamically checks the conformance in behaviors of the particular implementation with respect to the formal specification. The checking results are then fed back to the users of the system. One of the major advantages of our proposed approach is that it does not embed any instrumentation codes to the target system, nor does it annotate the target system with any formal specifications. The implementation and the specification are still kept separated at different abstract levels. It is only during the monitoring process the two behaviors are extracted and compared for consistencies.

The remainder of the paper is organized as follows. Section 2 introduces the background information about Z formal specification language and specification animation. Section 3 presents an overview of the formal specification-based software monitoring technique, which describes a prototype system that we developed and discusses a few technical challenges that we encountered during the development. Section 4 demonstrates the effectiveness of the proposed monitoring technique with a case study. Section 5 analyzes the merits and the limitations of the proposed approach. Section 6 concludes the paper and discusses the future work.

2 Background

2.1 The Z specification language

The Z specification language has been a widely accepted formal language for specifying software and hardware systems. Based on set theory and first order predicate logic, it models a system by describing its states and the ways in which the states can be changed. This modeling style makes Z not only a good match to imperative, procedural programming languages but also a natural fit to object-oriented programming [12].

The specification written in Z notation typically includes a number of state schemas and operation schemas. A state schema encapsulates variable declarations and related predicates (invariants). The system state is determined by the values taken by those state variables subject to restrictions imposed by state invariants. An operation schema defines the relationship between the 'before' and 'after' states corresponding to one or more state schemas. Moreover, the *schema calculus* of Z notation provides a convenient way to construct formal descriptions of complex operations from simple operation schemas. Detailed information about the syntax and semantics of Z notation can be found in [13,14].

As an example, the Z specification shown in Fig. 2 describes a *Queue* with a First In First Out (FIFO) behavior in nature. The state schema declares that *Queue* is composed of a sequence of integers and further restrains the size of the queue to be less than ten. The *InitQueue* schema describes the operation that initializes the *Queue* as empty. The operation

[1] This paper is a revised/extended version of the conference paper [11].

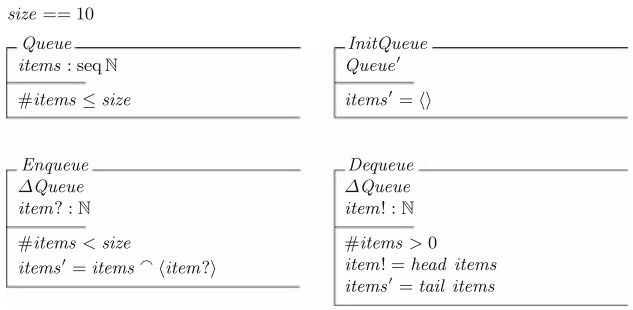

Fig. 2 A FIFO *Queue* in Z notation

schemas *Enqueue* and *Dequeue* describe the operation that attaches a new item to the tail of a queue and the operation that deletes an item from the head of a queue, respectively.

2.2 Specification animation

Specification animation is a technique to explore and exhibits the dynamic behaviorial properties of the formal specifications. It assists system designers in systematically verifying whether the expected properties and behaviors of a system are specified correctly and consistently in the specification by giving the designers an early view of the high-level dynamic behavior of the formal requirement description. It also provides a way to validate the formal specifications with the end users and field experts to ensure that what is formally described is really what was desired [15,16].

Specification animators are executable systems that interpret a formal specification into a high-level dynamically executable form. Animators have been developed for various formal languages. For example, PiZA [17] is an animator for Z. Possum [18,19] is an animator for Z and Z-like specification languages. B-Model animator [20] is an animator for B Method's model-oriented specification language. The specification animator used in our prototype monitoring system is Jaza [21]. It is an animator for Z, which has a strong support for quantifiers and various less-often-used Z constructors (such as μ, λ, θ terms). It provides more efficient and convenient evaluation of schemas on ground data values. And it has the ability to search for example solutions of a schema or predicate. Jaza supports many different representations of sets, which makes it more advanced in its execution than other animators for Z. Moreover, Jaza can handle not only unpredictable performance characteristics but also nondeterministic schemas.

3 Formal specification-based software monitoring

3.1 Overview

As shown in Fig. 1, our formal specification-based monitoring approach gathers the information about the dynamic behavioral properties of the formal specification through specification animation. And with the debugging module, the information about the dynamic behavior of the concrete implementation is obtained through program debugging. Taking the execution sequences provided by the user as input, the monitoring module controls the specification animator and debugging module so that the concrete implementation can run in parallel with the animation of the formal specification. With the information obtained from the specification animator and debugging module, the monitoring module makes a judgement on the conformance of the concrete implementation with respect to the formal specification. If any inconformity is found, it reports to the user. The user then needs to make a decision about how to deal with such an inconformity. In our approach, the monitoring module functions as an external observer of the target system. Moreover, it is designed to monitor the target system and respond in a timely manner while the target system is running. This means that our monitoring approach not only gathers information, but also dynamically interprets the gathered information and responds appropriately.

3.2 Real-time monitoring versus program debugging

Based on the above design, we have implemented a prototype monitoring system to demonstrate our approach. The prototype works with the formal specification written in the Z language and the concrete implementation programmed in the Java language. We use the Jaza [21] animator for exploring the dynamic properties of the Z formal specification and the ***jdb*** [22] debugger for extracting execution information of Java programs. Note that ***jdb*** is a debugger supplied by Sun in the Java Developer's Kit (JDK), which is implemented using the Java Debugger API.

Our monitoring system can be executed in two different modes, i.e., real-time monitoring mode and program debugging mode. The former provides a on-the-fly and real-time monitoring of the target systems, which is quite useful in continuous behavior checking of safety critical systems. The latter provides a batch mode in examining a sequence of behaviors of the target systems, which can be used during software testing phase for running predefined test cases and oracles. After the formal specification and concrete implementation are loaded to the monitoring system, the system extracts the operations and state variables defined in the formal specification as well as the methods and class variables defined in concrete implementation. With the assistance from the users, the monitoring system matches the operations defined in the formal specification with corresponding methods defined in the concrete implementation. It also performs similar matching for the state variables and corresponding class variables. To briefly illustrate how the prototype works, the Z specification in Fig. 2 is used as an example.

In the real-time monitoring mode, as shown in Fig. 3, the user manipulates the monitoring system by indicating the methods to be executed and inputting parameters (if necessary) in a step-by-step manner. The commands for executing corresponding operations are automatically generated for the animator. The monitoring system checks the running result of the implementation execution with the corresponding specification animation result to determine whether there is any inconsistency. This way, the system can achieve the on-the-fly monitoring of concrete implementations against formal specifications.

In the program debugging mode, as shown in Fig. 4, the user inputs all of the methods that are expected to be executed into the monitoring system in a predefined sequence. The system automatically generates the sequence of corresponding running commands for the animator. It performs the dynamic checking whenever a method and the corresponding operations schema have been executed/animated in a parallel way. Figure 4 shows how the monitoring system reports the inconformity to users. After the execution of the operation of deleting an item from the queue, the monitoring reports an inconformity. With a careful examination of the implementation we found that the operation *Dequeue* is not implemented correctly as in FIFO manner. Moreover, the monitoring system also provides the users with two choices to handle the inconformity: (1) stop monitoring to fix the problem in the implementation; (2) re-initialize the animator with the corresponding execution result from the implementation and continue the debugging process.

4 Case study: monitor a robotic assembly system

To further demonstrate the effectiveness of our approach, we applied a more realistic and reasonable-sized case study—a robotic assembly system example [23–25]. Autonomous Nano-Technology Swarm (ANTS) mission [26–28] is one of NASA's future space exploration missions which use intelligent swarms of spacecrafts [29,30]. During the ANTS mission, a transport spacecraft launched from the earth towards the Lagrangian point carries an assembling laboratory. The autonomous, pico-class, low-power, and low-weight spacecrafts that explore the asteroid belt for asteroids with certain scientific characteristics will be assembled in that laboratory. Each spacecraft is equipped with a solar sail, which means it relies primarily on power from the sun, using only tiny thrusters to navigate independently. Also, each spacecraft has onboard computation, artificial intelligence, and heuristics mechanism for control at the individual and team levels, and it has communicating mechanism for the communication within swarm and the data transfer back to the earth. Moreover, approximately 80% of the spacecrafts will be workers. The workers will carry a single specialized instrument, such as a magnetometer and an X-ray, gamma-ray, visible/IR, or neutral mass spectrometer, for the collection of a specific type of data from asteroids in the belt [28,31].

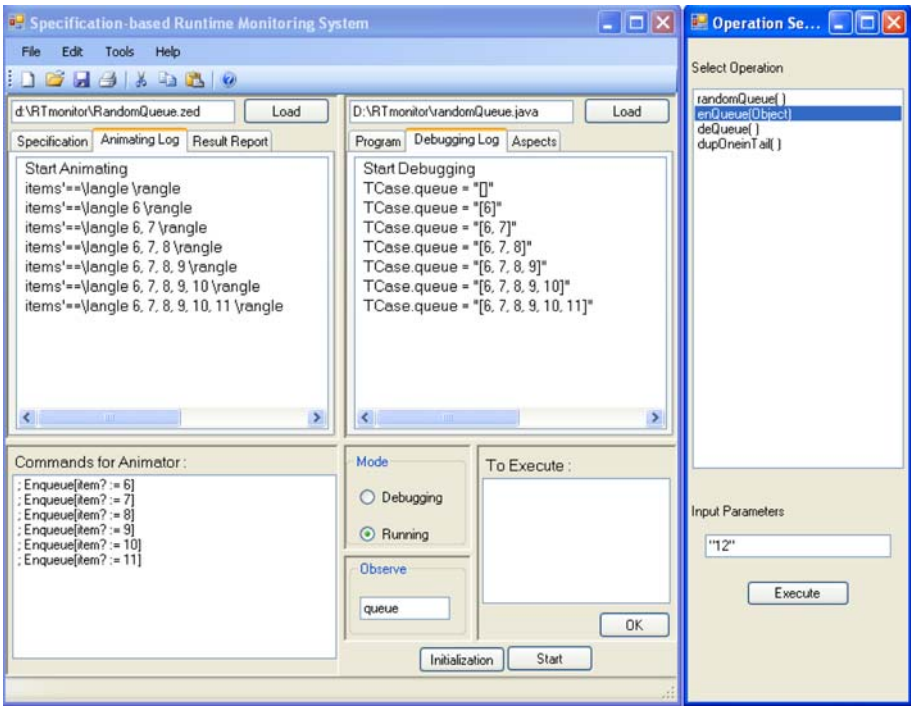

Fig. 3 Real-time monitoring mode

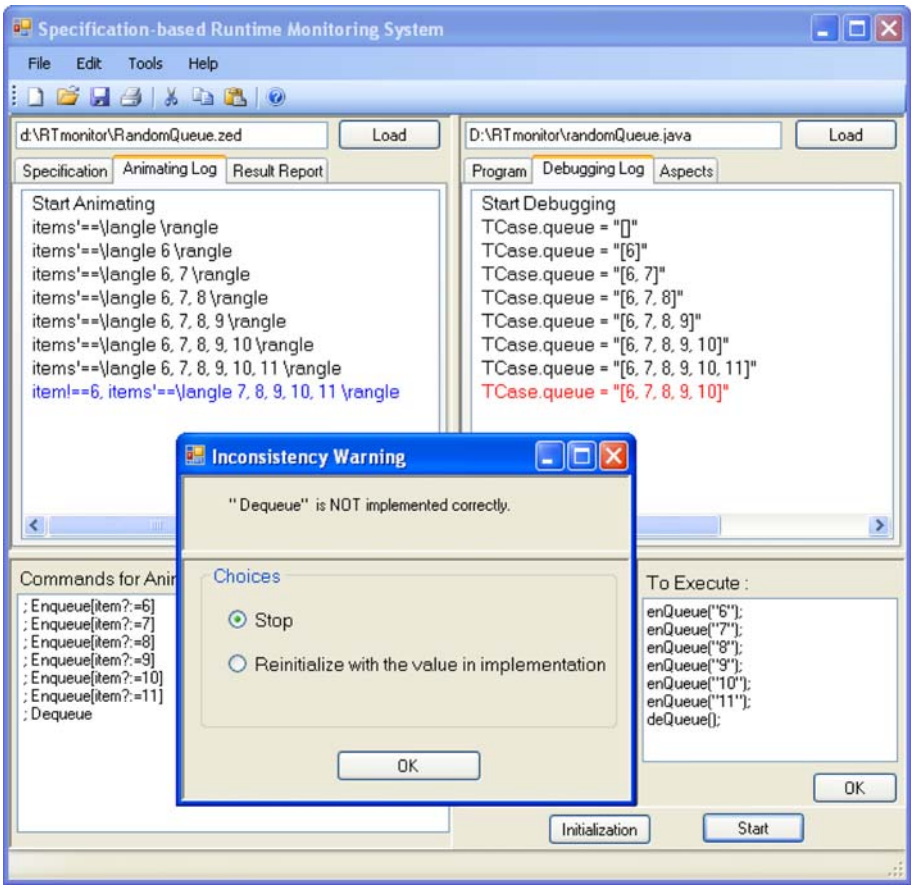

Fig. 4 Program debugging mode

Thus the correct assembly of spacecrafts is crucial to the ANTS mission. We reuse the framework of the robotic assembly system which has been studied in [23–25], and extend it so that it can be used for the assembly of spacecrafts in the ANTS mission. As shown in Fig. 5, an assembly unit consists of a robot system, a conveyor belt, and an assembly-tray. The conveyor belt normally carries the objects to be assembled. The robot system consists of two arms, a vision system, and a stack. The vision system can recognize the objects to be assembled and record their numbers. The stack is for temporarily storing the objects. The basic behaviors of the left/right arms can be described as follows:

- Initially, the arms are free and the stack is empty. Whenever both arms are free and the stack is empty, and the vision system recognize an object, then the left/right arm picks up the item from the conveyor belt and begins the process of assembly.
- With the assembly in progress, if the object on the left/right arm is the same as a part of the half-assembled product, the left/right arm will push that object into the stack; otherwise, the object will be assembled.
- Whenever the left/right arm is free but the stack is not empty and the top item of the stack is not same as any part of half-assembled product, the left/right arm picks up (pops) an object from the stack rather than picks up an object from the conveyor belt and installs it in the half-assembled product.

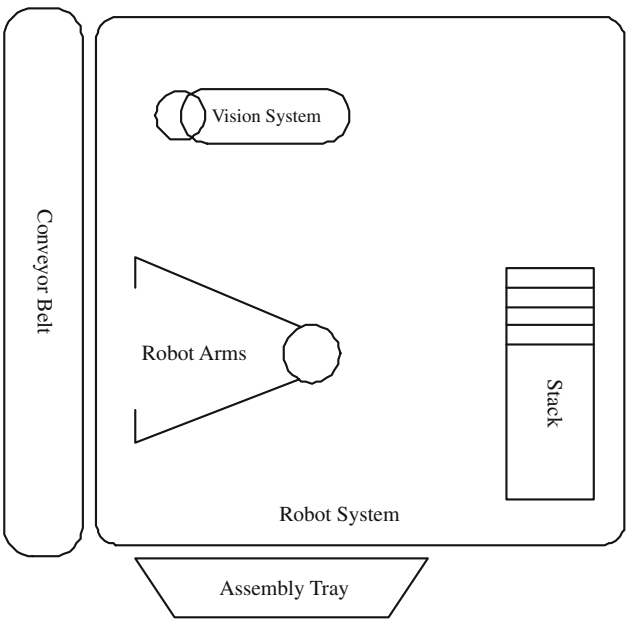

Fig. 5 Robotic assembly system

- If the assembly of one piece of product has been completed, the product will be released and placed on an assembly-tray.

As introduced before, the spacecraft consists of five main parts, namely, power system, navigation system, control system, communication system, and specialized instrument (e.g., magnetometer or spectrometer). In the extended robotic assembly system framework, the left arm of the robot will be responsible for installing the power system, navigation system, and control system while the right arm will be in charge of the installation of the communication system and the specialized instrument. A formal description of the robotic assembly system in Z notation is defined as follows:

$$Part ::= PowerSys \mid NavigationSys \mid ControlSys \mid CommunicationSys \mid MagnetoMeter \mid SpectroMeter$$

RobotSystem

$$leftarm, rightarm : \text{seq } Part$$
$$tempstack : \text{seq } Part$$
$$currentproduct : Part \pfun \mathbb{N}$$

InitRobotSystem

$$RobotSystem'$$

$$leftarm' = \langle\rangle \land rightarm' = \langle\rangle \land tempstack' = \langle\rangle$$
$$currentproduct' = \varnothing$$

The behaviors of the left arm can be summarized as follows. When the left arm is empty, if the stack is also empty or the item at the top of the stack is not an eligible one to be picked by the left arm, the left arm will get an item from the conveyor belt. If the item which was picked up by the left arm from the conveyor belt is identical to any component existing in the spacecraft-to-be, the left arm will push it to the stack. Otherwise, the left arm will install it in a designated position. On the other hand, when the left arm is empty, if the stack is not empty and the item at the top of the stack is an eligible item for the left arm, the left arm will pick the top item of the stack up and install it rather than grasp any item from the conveyor belt. The formal specifications are defined in the following.

LeftArmPick

$$\Delta RobotSystem$$
$$part? : Part$$

$$tempstack = \langle\rangle \lor head\ tempstack \in \text{dom } currentproduct \lor head\ tempstack \notin \{PowerSys, NavigationSys, ControlSys\}$$
$$part? \in \{PowerSys, NavigationSys, ControlSys\}$$
$$leftarm = \langle\rangle \land leftarm' = \langle part? \rangle$$
$$tempstack' = tempstack \land rightarm' = rightarm$$
$$currentproduct' = currentproduct$$

LeftArmGetFromStack

$$\Delta RobotSystem$$

$$\#tempstack > 0 \land \#leftarm = 0$$
$$head\ tempstack \notin \text{dom } currentproduct$$
$$head\ tempstack \in \{PowerSys, NavigationSys, ControlSys\}$$
$$leftarm' = \langle head\ tempstack \rangle \land rightarm' = rightarm$$
$$currentproduct' = currentproduct \land tempstack' = tail\ tempstack$$

LeftArmRelease

$$\Delta RobotSystem$$
$$part! : Part$$

$$\#leftarm = 1 \land leftarm(1) \notin \text{dom } currentproduct$$
$$part! = leftarm(1) \land leftarm' = \langle\rangle$$
$$leftarm(1) = PowerSys \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 1\}$$
$$leftarm(1) = NavigationSys \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 2\}$$
$$leftarm(1) = ControlSys \Rightarrow currentproduct' = currentproduct \cup \{part! \mapsto 3\}$$
$$tempstack' = tempstack \land rightarm' = rightarm$$

LeftArmPushToStack

$$\Delta RobotSystem$$
$$parttopush! : Part$$

$$\#leftarm = 1 \land leftarm(1) \in \text{dom } currentproduct$$
$$parttopush! = leftarm(1) \land leftarm' = \langle\rangle$$
$$tempstack' = \langle parttopush! \rangle \frown tempstack$$
$$rightarm' = rightarm \land currentproduct' = currentproduct$$

Generally, the right arm of the robot works in a similar way as the left one. However, the right arm has to deal with some more intricate situations since it is in charge of the installation of not only the communication system but also the specialized instrument. There are two kinds of specialized instrument, only one of them will be included in a spacecraft in the ANTS mission. And, there is no way to replace it with another one if a specialized instrument has been included in the spacecraft. Therefore, it has to be checked whether a specialized instrument has been included in the spacecraft-to-be when the right arm tries to install or pick up a specialized instrument from the stack. The formal specifications are defined as follows.

RightArmPick

$\Delta RobotSystem$

$part? : Part$

$tempstack = \langle\rangle \lor head\ tempstack \in \text{dom}\ currentproduct \lor$
$head\ tempstack \notin \{CommunicationSys, SpectroMeter, MagnetoMeter\} \lor$
$\text{dom}\ (currentproduct \rhd \{5\}) \cup \{head\ tempstack\} =$
$\{SpectroMeter, MagnetoMeter\}$
$part? \in \{CommunicationSys, SpectroMeter, MagnetoMeter\}$
$rightarm = \langle\rangle \land rightarm' = \langle part?\rangle \land tempstack' = tempstack$
$currentproduct' = currentproduct \land leftarm' = leftarm$

RightArmGetFromStack

$\Delta RobotSystem$

$\#tempstack > 0 \land \#rightarm = 0$
$head\ tempstack \notin \text{dom}\ currentproduct$
$\text{dom}\ (currentproduct \rhd \{5\}) \cup \{head\ tempstack\} \neq$
$\{SpectroMeter, MagnetoMeter\}$
$head\ tempstack \in \{CommunicationSys, SpectroMeter, MagnetoMeter\}$
$rightarm' = \langle head\ tempstack\rangle \land leftarm' = leftarm$
$currentproduct' = currentproduct \land tempstack' = tail\ tempstack$

RightArmRelease

$\Delta RobotSystem$

$part! : Part$

$\#rightarm = 1 \land rightarm(1) \notin \text{dom}\ currentproduct$
$\text{dom}\ (currentproduct \rhd \{5\}) \cup \{rightarm(1)\} \neq$
$\{SpectroMeter, MagnetoMeter\}$
$part! = rightarm(1)$
$rightarm(1) = CommunicationSys \Rightarrow$
$currentproduct' = currentproduct \cup \{part! \mapsto 4\}$
$(rightarm(1) = MagnetoMeter \lor rightarm(1) = SpectroMeter) \Rightarrow$
$currentproduct' = currentproduct \cup \{part! \mapsto 5\}$
$rightarm' = \langle\rangle \land tempstack' = tempstack \land leftarm' = leftarm$

RightArmPushToStack

$\Delta RobotSystem$

$parttopush! : Part$

$\#rightarm = 1 \land (rightarm(1) \in \text{dom}\ currentproduct \lor$
$\text{dom}\ (currentproduct \rhd \{5\}) \cup \{rightarm(1)\} =$
$\{SpectroMeter, MagnetoMeter\})$
$parttopush! = rightarm(1) \land rightarm' = \langle\rangle$
$tempstack' = \langle parttopush!\rangle \frown tempstack$
$leftarm' = leftarm \land currentproduct' = currentproduct$

Finally, when a product is assembled, the release operation can be described as follows.

ReleaseProduct

$\Delta RobotSystem$

$producttorelease! : Part \pfun \mathbb{N}$

$\#currentproduct = 5$
$currentproduct(PowerSys) = 1$
$currentproduct(NavigationSys) = 2$
$currentproduct(ControlSys) = 3$
$currentproduct(CommunicationSys) = 4$
$(currentproduct(MagnetoMeter) = 5 \lor$
$currentproduct(SpectroMeter) = 5)$
$producttorelease! = currentproduct \land currentproduct' = \varnothing$
$leftarm' = \langle\rangle \land rightarm' = \langle\rangle \land tempstack' = tempstack$

Once the implementation of such a robotic assembly system has been developed, in order to check whether the implementation conforms to the system requirements which were expressed by the formal specification, we can load both the implementation and the formal specification into our specification-based monitoring system and perform the checking. As shown in Fig. 6, the system starts monitoring with a sequence of method execution, i.e., "*leftArmPick ("PowerSys")*; *leftArmRelease ()*; *leftArmPick ("ControlSys")*; *leftArmRelease ()*; *rightArmPick ("Magneto-Meter")*; *rightArmRelease ()*; *rightArmPick ("CommunicationSys")*; *rightArmRelease()*; *rightArmPick("SpectroMeter")*; *rightArmRelease()*; *leftArmPick ("Navig-ationSys")*; *leftArmRelease ()*". After method *rightArmRelease()* is executed for the second time, which installs the communication system in the spacecraft, the monitoring system detects an inconformity and reports to the user that the operation schema *RightArmRelease* is not implemented correctly. According to the specification, the operation schema *RightArmRelease* defines that if the right arm is holding an item and the held item is not identical to any existing component of the spacecraft-to-be, the right arm will install the item to a designated position, in this case—the communication system to the fourth socket. When we double check the corresponding execution of the Java implementation, i.e., the *rightArmRelease()* method shown below, we found that the arm actually puts the communication system to the second socket. Thus, an inconsistency between the specification and implementation is identified.

```
public void rightArmRelease() {
    if(!rightarm.isEmpty())
    {   Object releasedPart = rightarm.get(0);
        if (!currentproduct.contains(releasedPart))
        {   if(releasedPart == ''CommunicationSys'')
            { currentproduct.setElementAt(releasedPart, 1);}
            else if (releasedPart == ''MagnetoMeter''||
                                releasedPart == ''SpectroMeter'')
                {currentproduct.setElementAt(releasedPart, 4);}
            rightarm.clear();
        }
    }
}
```

If we choose to continue the monitoring, when the method *rightArmRelease()* is executed for the third time installing a spectrometer to spacecraft, the monitoring system detects another inconformity. To localize the cause of the inconformity, we revisit the formal specification and the Java code. The operation schema *RightArmRelease* specifies that only one specialized instrument (i.e. magnetometer or spectrometer) will be installed and there is no way to replace it with another one if a specialized instrument has been included in the spacecraft. However, in the Java code, there is no statement for checking whether a specialized instrument has been included in the spacecraft when the right arm tries to install a specialized instrument to the spacecraft. Thereby, another error in the implementation was detected. Furthermore, when the monitoring system proceeds with the methods sequence *leftArmPick("NavigationSys"); leftArmRelease();* the error in the method *leftArmRelease()* can be effectively identified as well.

5 Discussions and lessons learned

5.1 Merits of the formal specification-based monitoring technique

The formal specification-based monitoring technique proposed in this paper gets required information about dynamic behaviors of the formal specification and concrete

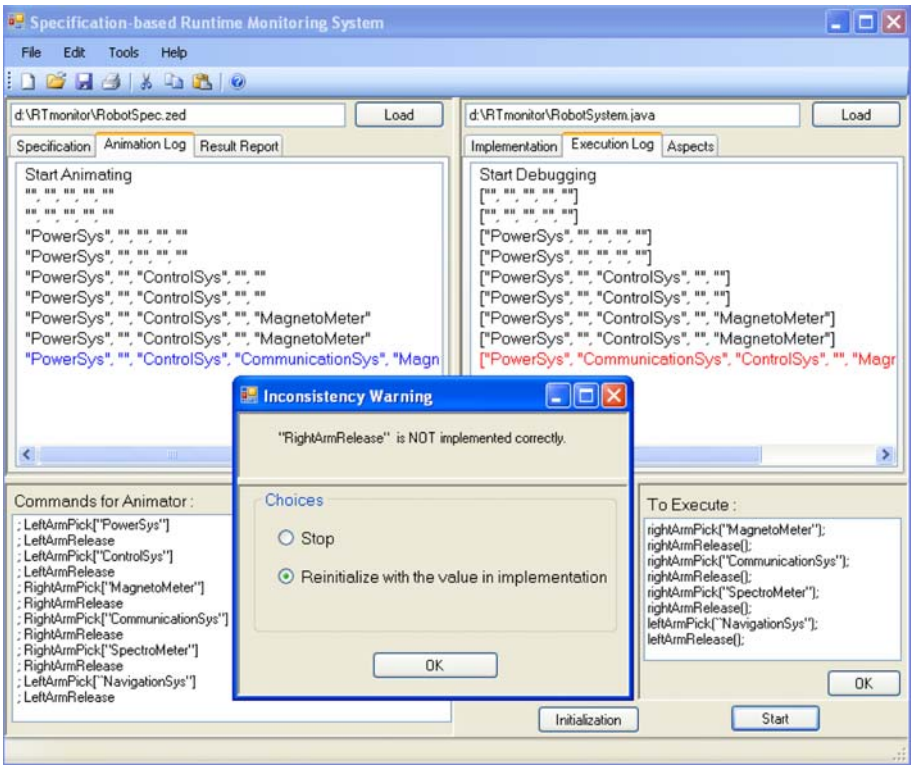

Fig. 6 Monitor a robotic assembly system

implementation of the target system through specification animating and program debugging, respectively. It does not embed any instrumentation code into the target system; therefore, the proposed formal specification-based runtime monitoring technique will not alter the running environment and the dynamic behaviors of the target system which is being monitored. Meanwhile, it does not annotate the concrete implementation with any extra formal specifications either; consequently, it allows the reuse of a highly abstract formal requirement specification when changes happen to the implementation of the target system.

Furthermore, the formal specification-based monitoring technique always monitors the current state of the system, continuously checks the conformance of implementation with formal specification and reports any detected inconformity immediately whenever any failures happen. Therefore, by ensuring that the current execution is conformable with its requirements at runtime, formal specification-based runtime monitoring can provide the developers and users with much higher confidence in the software than traditional testing. Besides, although the formal specification-based monitoring technique is weaker than formal verification as far as the ability to guarantee software correctness is concerned, it provides a dynamic verification technique by checking that the actual execution of a system is conformable with the expectation described by the formal specifications. Rather than checking that the design model of the system satisfies some properties, as formal verification does, formal specification-based monitoring checks that the results of particular computations when the system is executed are correct with respect to the formal specification. Thus, formal specification-based monitoring escapes from the state-space explosion problem that limits the scalability of formal verification techniques. Therefore, the formal specification-based monitoring technique can serve as a complement to traditional testing and formal verification techniques in software quality assurance.

5.2 Limitations of the formal specification-based monitoring technique

The limitation of the proposed formal specification-based runtime monitoring approach is often related to the capabilities of the specification animator. It is very likely that some aspects of the formal specification cannot be simulated by the animator. For example, Jaza, the animator that we used, does not support the type of *bag* and generic constructs in Z notation. It cannot handle user-defined infix/prefix/postfix functions or relations either.

There are also limitations resulting from the manner in which specifications are expressed. State-based specification languages such as Z and VDM do not allow the developer to easily express temporal and concurrent properties of a system, while process-oriented specification languages such as CSP and CCS are generally poor at expressing the structure and state of a system. So far, we focus on working with state-based specification languages. In the future, the formal specification-based monitoring system that we have developed will be extended to verify temporal and concurrent aspects of software systems.

Furthermore, the description granularity of the formal specification languages determines the granularity of inconformity detection that the proposed formal specification-based approach can achieve. For example, the Z specification language specifies a system by describing its state and the way in which the states can be changed. The formal specification of a system in Z notation consists of a sequence of state schema and operation schemas. Therefore, the monitoring system which works with specifications in Z notation can only detect which operation schema in the specification is not implemented as expected, but cannot determine which predicate in the specification is violated. Moreover, there are syntactic and semantic gaps between specification and programming languages. Usually, implementations address much more details than formal specifications do. Consequently, the proposed monitoring approach can detect which method in the program is not implemented correctly with respect to the corresponding operation schema in the formal specification, but cannot precisely locate the particular statement in the implementation that contains the error.

6 Conclusion

This paper presents a formal specification-based software monitoring approach. With the formal specification and concrete implementation of the target system, our specification-based monitoring approach uses a specification animator to exhibit the dynamic behavior of the formal specification, uses a program debugger to extract required information about the dynamic behavior of the concrete implementation, and checks the conformance of the concrete implementation with the formal specification, based on the information from the animator and the debugger.

Our monitoring approach gets required information about dynamic behaviors of the formal specification and concrete implementation of the target system through specification animation and program debugging respectively, rather than by embedding any instrumentation code into the target system or by annotating the concrete implementation with extra formal specifications. Consequently, our formal specification-based runtime monitoring technique will not alter the running environment and the dynamic behaviors of the target system which is being monitored. Moreover, our monitoring technique realizes the clear separation between the implementation-dependent description of monitored object and the

highly abstract formal specification of it, which allows the reuse of a formal requirement specification when changes happen to the implementation of the target system.

At present, our formal specification-based monitoring technique works at intra-class level—it can detect the incorrect implementation of methods in the class. To improve the monitoring technique so that it can work at inter-class level and detect errors caused by the improper invocations of methods between classes is a part of future work. Furthermore, monitoring distributed and parallel systems during execution can provide information that can be used to reconfigure the system, provide visualization of behavior, or steer its outcome [32]. Therefore, we also intend to extend our monitoring technique so that it can handle distributed and parallel systems.

Acknowledgments We would like to express our sincere gratitude to Professor Roger Duke and Professor Rudolph E. Seviora for their insightful comments and valuable suggestions on the work presented in this paper.

References

1. Abercrombie P, Karaorman M (2002) jContractor: Bytecode instrumentation techniques for implementing design by contract in Java. In: Proceedings of second workshop on runtime verification (RV'02)
2. Barnett M, Rustan K, Schulte W (2004) The Spec# programming system: an overview. In: Proceedings of international workshop on construction and analysis of safe, secure, and interoperable smart devices, pp 49–69
3. Bartetzko D, Fischer C, Moller M, Wehrheim H (2001) Jass—Java with assertions. In: Proceedings of first workshop on runtime verification, RV'01
4. Chen F, D'Amorim M, Roşu G (2004) A formal monitoring-based framework for software development and analysis. In: Proceedings of the 6th international conference on formal engineering methods (ICFEM'04), Springer, Heidelberg, pp 357–373
5. Drusinsky D (2000) The temporal rover and the ATG rover. In: Proceedings of the 7th international SPIN workshop on SPIN model checking and software verification, pp 323–330
6. Havelund K, Roşu G (2001) Java PathExplorer—a runtime verification tool. In: Proceedings of 6th international symposium on artificial intelligence, robotics and automation in space, ISAIRAS'01
7. Hlady M, Kovacevic R, Li JJ, Pekilis B, Prairie D, Savor T, Seviora R, Simser D, Vorobiev A (1995) An approach to automatic detection of software failures. In: Proceedings of sixth international symposium on software reliability engineering
8. Kim M, Kannan S, Lee I, Sokolsky O (2001) Java-MaC: a runtime assurance tool for Java. In: Proceedings of first workshop on runtime verification, RV'01
9. Karaorman M, Abercrombie P (2005) jContractor: introducing design-by-contract to Java using reflective bytecode instrumentation. Formal Methods Syst Des 27(3):275–312
10. Satpathy M, Leuschel M, Butler MJ (2005) ProTest: an automatic test environment for B specifications. Electron. Notes Theor Comp Sci 111:113–136
11. Liang H, Dong JS, Sun J, Duke R, Seviora RE (2006) Formal Specification-based Online Monitoring. In: ICECCS '06: proceedings of the 11th IEEE international conference on engineering of complex computer systems, Washington, DC, USA, IEEE Computer Society, Los Alamitos, pp 152–160
12. Jacky J (1997) The way of Z: practical programming with formal methods. Cambridge University Press, Cambridge
13. Spivey J (1989) The Z notation: a reference manual. Prentice-Hall, Englewood Cliffs
14. Woodcock J, Davies J (1996) Using Z: specification, refinement, and proof. Prentice-Hall, Englewood Cliffs
15. Miller T, Strooper P (2000) A framework for systematic specification animation. Technical report 02-35, The University of Queensland
16. Miller T, Strooper P (2002) Model-based specification animation using testgraphs. In: ICFEM '02: proceedings of the 4th international conference on formal engineering methods, Springer, Heidelberg, pp 192–203
17. Hewitt MA, O'Halloran C, Sennett CT (1997) Experiences with PiZA, an animator for Z. In: ZUM '97: Proceedings of the 10th international conference of Z users on the Z formal specification notation, Springer, Heidelberg, pp 37–51
18. Hazel D, Strooper P, Traynor O (1997) Possum: an animator for the SUM specification language. In: APSEC '97: proceedings of the fourth Asia-Pacific software engineering and international computer science conference, IEEE Computer Society, Los Alamitos, p 42
19. Hazel D, Strooper P, Traynor O (1998) Requirements engineering and verification using specification animation. In: ASE '98: Proceedings of the Thirteenth IEEE Conference on Automated Software Engineering, IEEE Computer Society, Los Alamitos, p 302
20. Waeselynck H, Behnia S (1998) B model animation for external verification. In: Proceedings of the second IEEE international conference on formal engineering methods, ICFEM '98, pp 36–45
21. Utting M (2000) Data structures for Z testing tools. In: Proceedings of FM-TOOLS
22. jdb - The Java debugger. (http://java.sun.com/)
23. Achuthan R, Alagar VS, Radhakrishnan T (1995) An object-oriented modeling of real-time robotic assembly system. In: Proceedings of the 1st IEEE international conference on engineering of complex computer systems (ICECCS '95), pp 310–313
24. Alagar VS, Ramanathan G (1991) Functional specification and proof of correctness for time dependent behaviour of reactive systems. Formal Aspect Comput 3(3):253–283
25. Dong JS, Colton J, Zucconi L (1996) A formal object approach to real-time specification. In: Proceedings of the 3rd Asia-Pacific software engineering conference (APSEC'96)
26. Curtis SA, Mica J, Nuth J, Marr G, Rilee ML, Bhat MK (2000) ANTS (Autonomous Nano-Technology Swarm): an artificial intelligence approach to asteroid belt resource exploration. In: Proceedings of international astronautical federation, 51st Congress
27. Curtis SA, Truszkowski WF, Rilee ML, Clark PE (2003) ANTS for human exploration and development of space. In: Proceedings of IEEE aerospace conference
28. Hinchey MG, Dai YS, Rouff CA, Rash JL, Qi MR (2007) Modeling for NASA autonomous nano-technology swarm missions and model-driven autonomic computing. In: AINA '07: Proceedings of the 21st international conference on advanced networking and applications, pp 250–257
29. Truszkowski WE, Hinchey MG, Rash JL, Rouff CA (2004) NASA's swarm missions: the challenge of building autonomous software. IT Prof 6(5):47–52
30. Truszkowski WE, Hinchey MG, Rash JL, Rouff CA (2006) Autonomous and autonomic systems: a paradigm for future space

exploration mission. IEEE Trans Syst Man Cybermet Part C Appl Rev 36(3):279–291
31. Hinchey MG, Rouff CA, Rash JL, Truszkowski WF (2005) Requirements of an integrated formal method for intelligent swarms. In: FMICS '05: Proceedings of the 10th international workshop on formal methods for industrial critical systems, pp 125–133
32. Schroeder BA (1995) On-line monitoring: a tutoiral. IEEE Comp 28(6):72–78