Formal Specification-based Online Monitoring

Hui Liang
School of Computing
National University of Singapore
lianghui@comp.nus.edu.sg

Jin Song Dong
School of Computing
National University of Singapore
dongjs@comp.nus.edu.sg

Jing Sun
Department of Computer Science
The University of Auckland
j.sun@cs.auckland.ac.nz

Roger Duke
School of ITEE
The University of Queensland
rduke@itee.uq.edu.au

Rudolph E. Seviora
Department of Electrical and Computer Engineering,University of Waterloo
seviora@swen.uwaterloo.ca

Abstract

With current trends towards more complex software system and use of higher level languages, a monitoring technique is of increasing importance for the areas such as performance enhancement, dependability, correctness checking and so on. In this paper, we present a formal specification-based online monitoring technique. The key idea of our technique is to build a linking system, which connects a specification animator and a program debugger. The required information about dynamic behaviors of the formal specification and concrete implementation of a target system is obtained from the animator and the debugger. Based on those information, the judgement on the consistency of the concrete implementation with the formal specification will be provided. Not embedding any instrumentation code into the target system, our monitoring technique will not alter the dynamic behavior of the target system. Animating the formal specification, rather than annotating the target system with extra formal specifications, our monitoring technique separates the implementation-dependent description of the monitored objects and the formal requirement specification of them.

1. Introduction

Monitoring techniques gather information about a computational process as it executes. And monitoring systems are increasingly seen as a viable solution to areas which are of growing concern, such as performance enhancement, dependability, performance evaluation, security and so on [19]. Moreover, current trends toward more complex programs and use of higher level languages make software monitoring more and more important for those areas. Monitoring an application to ensure the consistency with high-level requirement specifications is an efficient approach for correctness checking and it can also be used to detect runtime errors or as a verification technique. Recently, there has been increasing attention from the research community to the design of monitors which can be used to assure the correctness of a system at runtime [18, 17, 15, 10, 14]. Those monitoring approaches usually add instrumentation code to the program to collect interesting data at runtime. Adding instrumentation code is itself a difficult task involving all the complexities of programming. Moreover, it generally leads to changes in the program. And it raises the possibility that through collecting information to analyze target system behavior, the monitoring system is actually altering that behavior of the target system. This is referred as Heisenberg effect for software [19]. In some of the above approaches, monitoring is achieved by annotating the concrete implementation with extra formal specifications. Therefore, another main drawback of the above monitoring techniques is the lacking of separation between the concrete implementation of target systems and the high-level requirements specification of them.

In this paper, we propose a formal specification-based online software monitoring technique. Formal specifications present a clear, precise and unambiguous description of the required behaviours of the target system. Although the process of creating formal specification might be la-

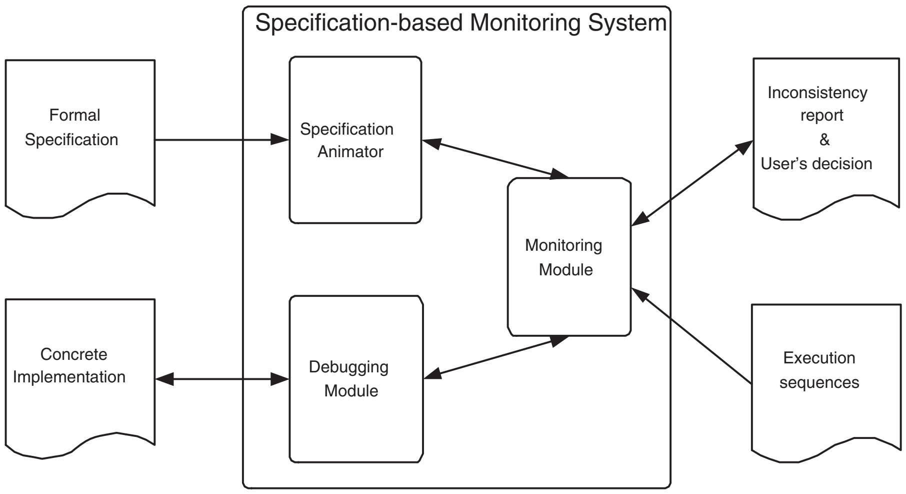


Figure 1. Formal specification-based online monitoring.

borious and time-consuming, formal specification can contribute a great deal to system validation, software verification and software testing. Moreover, we believe that abstract formal requirements specification can also contribute a lot to software monitoring. With the assumption that there exists formal specification for the monitored objects, the key idea of our approach is to build a linking system (monitoring module) which connects a specification animator and a program debugger. As shown in **Figure 1**, in our specification-based monitoring approach, a specification animator is used to exhibit the dynamic behavioural properties of the formal specification; a debugger is used to extract the information about the dynamic behaviour of the concrete implementation. The monitoring module (linking system) will dynamically check the consistency of the concrete implementation with the formal specification.

The main characteristics of our monitoring technique are as follows:

- Not embedding any instrumentation codes to the target system.
- Not annotating the target system with any formal specifications.

The rest of the paper is organized as follows. Section 2 introduces the Z specification language with *a queue system* as example and also presents background information on specification animation. Section 3 describes the formal specification-based online monitoring technique and discusses a few key issues in the development of a specification-based monitoring system. In Section 4, the monitoring technique is applied to the railway track line automatic locking scheme of a railway system. Section 5 reviews some related work. Section 6 concludes the paper and discusses the future works.

2. Background

2.1. The Z specification language

The Z specification language has been a widely accepted formal language for specifying the behaviours of software and hardware systems. Based on set theory and first order predicate logic, Z is a model oriented specification language. It models a system by describing its states and the ways in which the states can be changed. This modelling style makes Z not only a good match to imperative, procedural programming languages but also a natural fit to object-oriented programming [9]. Actually, the Z specification language includes two parts: the mathematical language and the schema language [24]. The schema language can be used to structure and compose descriptions, making it possible to build big schemas from small ones.

$$size == 10$$

$$\begin{array}{|l}
\textit{Queue} \\ \hline
items : \operatorname{seq} \mathbb{N} \\ \hline
\# items \leq size \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\textit{InitQueue} \\ \hline
Queue' \\ \hline
items' = \langle \rangle \\ \hline
\end{array}$$

$$\begin{array}{|l}
\textit{Enqueue} \\ \hline
\Delta Queue \\
item? : \mathbb{N} \\ \hline
\# items < size \land items' = items \frown \langle item? \rangle \\ \hline
\end{array}$$

Dequeue

$$\begin{array}{|l} \Delta Queue \\ item! : \mathbb{N} \\ \hline \#items > 0 \\ item! = head\ items \land items' = tail\ items \end{array}$$

DupOneinTail

$$\begin{array}{|l} \Delta Queue \\ item! : \mathbb{N} \\ \hline \#items > 1 \land \#items < size \\ item! \in \operatorname{ran} items \land item! \neq head\ items \\ items' = items \frown \langle item! \rangle \end{array}$$

The Z specification above describes a queue system which is a First In First Out(FIFO) queue in nature, but with the addition of the *DupOneinTail* operation. The *DupOneinTail* schema describes an operation that selects an item, which is not the first element, from the queue randomly and adds it to the end of the queue. It is obvious that *DupOneinTail* is a nondeterministic operation because there will be more than one possible results for the execution of it when there are more than two items in the queue. This queue system will be used as an example in Section 3 to explain our specification-based monitoring technique and system.

2.2. Specification animation

Specification animation exhibits the dynamic behaviourial properties of formal specification. It not only gives the specification designers a way to test whether their specifications behave as expected, but also validates the behaviour of formal specifications with the end users. The specification animation technique has been used to assist systematic validation of formal specifications [12, 13].

In the last decade, several animation tools have been developed for executing and interpreting formal specifications automatically. For example, PiZA [8] is an animator for Z, and Possum [6, 7] is an animator for Z and Z-like specification language. The animation tool used in our 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 at least twelve different representations of set. And this 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 online monitoring

This section describes our formal specification-based online monitoring technique. Section 3.1 presents an overview of the monitoring technique. Section 3.2 describes a monitoring system that we have developed for demonstrating the technique. Section 3.3 discusses a few key issues which we encountered in the development of the specification-based monitoring system.

3.1. Overview of the monitoring technique

Given a concrete implementation and formal specification, our formal specification-based monitoring technique dynamically checks the consistency of the concrete implementation with the formal specification.

The overall picture of the monitoring technique has been shown in **Figure 1**. With the specification animator, our specification-based monitoring technique gets the information about the dynamic behavioural properties of the formal specification through specification animating; and with the debugging module, the information about the dynamic behaviour of the concrete implementation is gathered 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 is run in parallel with the animation of the formal specification. Meanwhile, based on the information obtained from the specification animator and debugging module, the monitoring module provides judgement on the consistency of the concrete implementation with the formal specification. If any inconsistency is found, it will be reported to the user. Given the inconsistency report, the user needs to make a decision about how to deal with such an inconsistency. Then, the monitoring system will continue its work according to user's decision.

In our monitoring technique, the monitoring module functions as an external observer of the target system. Moreover, the monitoring module is designed to monitor the target system and respond in a timely manner while the target system is running. This means that our monitoring technique not only gathers information, but also dynamically interprets the gathered information and responds appropriately. Therefore, it is an online monitoring technique [19].

3.2. Specification-based monitoring system

To demonstrate our specification-based monitoring technique, we have implemented a specification-based monitoring system. The monitoring system works with the formal specification written in Z formal language and the concrete implementation programmed in Java programming language. In our monitoring system, the animator used for animating the specification is Jaza [21], the debugger used for extracting required information from the execution of Java program is ***jdb*** [1]. ***jdb*** is the debugger supplied by Sun in the Java Developer's Kit (JDK); and it is implemented using the Java Debugger API.

The monitoring system can work in two different modes: debugging mode and running mode. After the formal specification and concrete implementation have been loaded to the monitoring system, the system will extract the operations and state variables defined in the formal specification, and the methods and class variables defined in concrete implementation. The user needs to match the operations defined in the formal specification with corresponding methods defined in the concrete implementation; and do the same to the state variables and corresponding class variables.

In the debugging mode, after matching the operations/state variables in specification and methods/class variables in implementation, the user inputs all of the methods which are expected to be executed into the monitoring system as a whole sequence; and the system will automatically generate the sequence of corresponding running commands for the animator. Then, the system starts the dynamic checking of the behaviourial results of the implementation against the behaviourial results of the specification animation. In the debugging mode, our specification-based monitoring system can serve as an effective dynamic test execution and test result checking tool.

In the running mode, after matching the operations/state variables in the specification with the methods/class variables in the implementation, the user chooses the execution step by indicating the methods to be executed and inputting parameters(if necessary). The commands for executing corresponding operations will be automatically generated for the animator. The monitoring system will then check the running result of the implementation execution with the corresponding specification animation result to figure out whether there is an inconsistency. In the running mode, the user indicates what will be executed next in a step-by-step way. Thus, the system can achieve the on-the-flying monitoring of concrete implementations against formal specifications. In the running mode, the user guided execution sequence selection can be easily adapted to connect to a runtime execution system to achieve the real-time monitoring of reactive safety-critical systems.

3.3. Key issues related to the monitoring system

Mappings between concrete implementation and formal specification. The formal specification of a software/hardware system usually describes the system at a high level of abstraction with a formal language. Sometimes, the specification is not executable [5], but it is generally very expressive and includes many rich abstract data types. The abstract data types in a specification have no data representation specified, the implementations of their operations are also kept abstract [16]. To implement the specification, those abstract data types must be implemented by the existing data types in the programming language. There may be various potential concrete representations in the implementation for an abstract data type in the specification. For example, in the specification of the queue system displayed in Section 2.1, the type of state variable *items* is *sequence*. Suppose the programming language we use to implement the queue is Java, we may use the `java.util.Vector` or `ArrayList` classes to implement the abstract data type *sequence*.

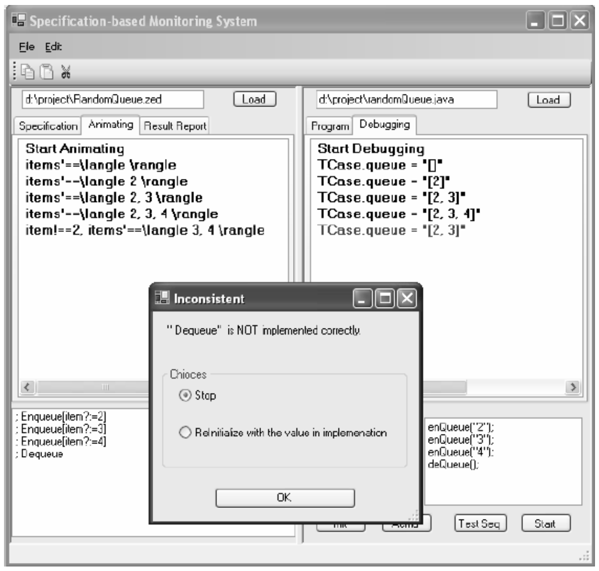
To monitor whether the concrete implementation is consistent with the formal specification, the first thing that needs to be done is to map the variables, data types and operations in formal specifications with their corresponding counterparts in concrete implementations. Our monitoring system automatically extracts the operations and state variables defined in the formal specifications, and the methods and class variables defined in concrete implementations; and it also provides the users with the mechanism to finish the mapping.

Inconsistency between concrete implementations and formal specifications. If the concrete implementation is not consistent with the formal specification, the monitoring system will detect and report such an inconsistency. What should the monitoring system do after an inconsistency has been revealed? If we let the animator keep running, all of the subsequent judgement will apparently be *inconsistent* due to the carry-over of inconsistency from the previous execution. Therefore, we propose two choices to the user. The first choice is to stop the monitoring process whenever an inconsistency is detected, and let the user to fix the problem in the implementation. However, this choice will reduce the effectiveness of the system as only one inconsistency can be found at a time. The second choice is to reinitialize the animator with the corresponding execution result from the implementation to keep the animation and the implementation in the same state before the next operation/method is executed. This choice is based on the assumption that the execute result of the implementation is correct.

```
class Queue {
  private Vector queue;
  Queue ()  {
      queue = new Vector();
  }
  public void enQueue (Object item) {
      queue.addElement(item);
  }
  public Object deQueue () {
      Object obj = null;
      int last;
      if (!queue.isEmpty()) {
            obj = queue.lastElement();
            last = queue.size();
         queue.removeElementAt(last-1);
      } return obj;
  }
  public Object dupOneinTail() {
      int n, i; Object obj = null;
      n = queue.size();
      if(n>1){
           i = (int)(n*Math.random( ));
           obj = queue.elementAt(i);
           queue.addElement(obj);
      } return obj;
  }
 }
```

For example, the Java code above is supposed to implement the queue system which is specified by the Z specification displayed in Section 2.1. After loading the specification and implementation to the monitoring system and finishing necessary mapping between the specification and implementation, we start the monitoring process. As shown in **Figure 2**, after the execution of the operation of deleting an item from the queue, the monitoring reports an inconsistency and points out that the operation *Dequeue* is not implemented correctly. When we checked the above Java code, we found that the method deQueue actually deletes the last item from the queue, not the first one, while the specification demands that the operation *Dequeue* delete the first item from the queue. When the monitoring system reports an inconsistency, it provides the two choices:(1) stop monitoring, (2) reinitialize the animator with the corresponding execution result from the implementation, as shown in **Figure 2**. If we choose to reinitialize the animator, the state variable *items* will be set to $\langle 2, 3 \rangle$ before the next operation is executed and the monitoring system can continue working. Alternatively, we can choose to stop monitoring, and fix the problem in the implementation.

Nondeterministic operations in the specification. An issue that complicates the matter is specification nondeterminism, i.e. the specification may involve the definition of nondeterministic operations, the execution of which may lead to more than one possible results. When encountering a nondeterministic operation in the specification, the animator will present a legal but stochastic result. However, the implementation is always deterministic. It may present a result that is legal to the specification but different from that of the animator. If the monitor only performs simple com-



d:\project\RandomQueue.zed

```
Start Animating
items'==\langle \rangle
items'--\langle 2 \rangle
items'==\langle 2, 3 \rangle
items'--\langle 2, 3, 4 \rangle
item!==2, items'==\langle 3, 4 \rangle
```

```
; Enqueue[item?:=2]
; Enqueue[item?:=3]
; Enqueue[item?:=4]
; Dequeue
```

d:\project\randomQueue.java

```
Start Debugging
TCase.queue = "[]"
TCase.queue - "[2]"
TCase.queue = "[2, 3]"
TCase.queue - "[2, 3, 4]"
TCase.queue = "[2, 3]"
```

```
enQueue("2");
enQueue("3");
enQueue("4");
deQueue();
```

" Dequeue" is NOT implemented correctly.

[✓ Stop]

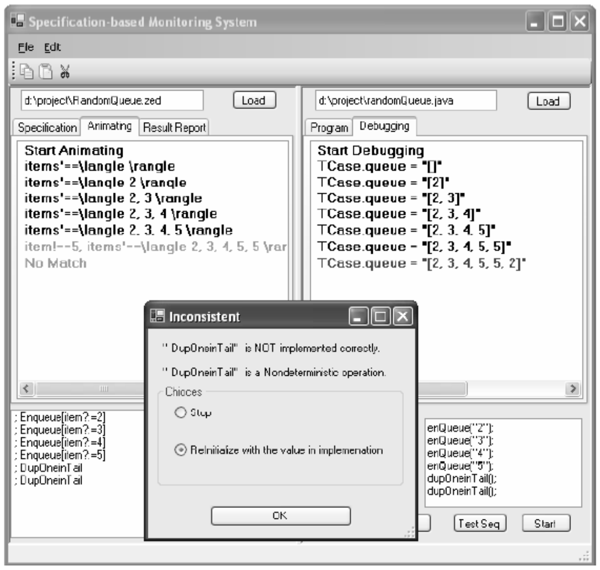
Reinitiaize with the value in implemenation

Figure 2. Inconsistency Appears.

parison of the two results, it will definitely make a wrong judgement. Therefore, how to make the monitoring system judge correctly in a nondeterministic situation is one of the major challenges of the specification-based monitoring technique. A possible solution for handling such nondeterminism is to make the animator present all the possible legal results and let the monitoring system take all of them into consideration when comparing the results between the implementation and the specification to avoid misjudgement. However, this approach may be time consuming and may considerably increase space complexity when the number of possible results are large.

Our specification-based monitoring system provides a mechanism for the user to indicate whether an operation is deterministic or nondeterministic. When a nondeterministic operation is encountered, the animator presents one possible result at a time. The monitoring system compares the result from specification animation with the corresponding result from implementation execution. If they are consistent, the system will decide that the implementation of this operation is correct. If not, the animator will continue to present another different possible result, the monitoring system will perform another comparison, the monitoring system will repeat the above process till the results are consistent or all of the possible animation results are presented. In the latter case, where the result from implementation is not consistent with any of the possible results from the animator, the system will make the judgement that the operation is not implemented correctly according to the formal specification.

From the Z specification of a queue system displayed in Section 2.1, we know that the operation *DupOneinTail* is



d:\project\RandomQueue.zed

```
Start Animating
items'==\langle \rangle
items'==\langle 2 \rangle
items'==\langle 2, 3 \rangle
items'==\langle 2, 3, 4 \rangle
items'==\langle 2, 3, 4, 5 \rangle
item!==5, items'==\langle 2, 3, 4, 5, 5 \rar
No Match
```

d:\project\randomQueue.java

```
Start Debugging
TCase.queue = "[]"
TCase.queue = "[2]"
TCase.queue = "[2, 3]"
TCase.queue = "[2, 3, 4]"
TCase.queue = "[2, 3, 4, 5]"
TCase.queue = "[2, 3, 4, 5, 5]"
TCase.queue = "[2, 3, 4, 5, 5, 2]"
```

" DupOneinTail" is NOT implemented correctly.

" DupOneinTail" is a Nondeterministic operation.

```
; Enqueue[item?=2]
; Enqueue[item?=3]
; Enqueue[item?=4]
; Enqueue[item?=5]
; DupOneinTail
; DupOneinTail
```

```
enQueue("2");
enQueue("3");
enQueue("4");
enQueue("5");
dupOneinTail();
dupOneinTail();
```

Figure 3. Handle nondeterminism.

a nondeterministic operation. There would be more than one possible results for a single execution of it. In **Figure 3**, it can be seen that method dupOneinTail, which is supposed to implement the operation *DupOneinTail*, appears twice in the execution sequence. When it is executed for the first time, it selects "5", which is the last element of the queue, and attaches it to the tail of the queue. This is allowed by the specification and the animator certainly finds a corresponding result that matches it. The monitoring system will not report any inconsistency. It only changes the color of the result in the window to indicate that the operation *DupOneinTail* is nondeterministic. However, when the method dupOneinTail is executed for the second time, it selects "2", which is the head of the queue, and attaches it to the tail of the queue. This should not be allowed according to the specification. The animator exhausts all the possible legal results and could not find a match. Therefore, the monitoring system concludes an inconsistency at this time. By examining the original implementation displayed in Section 3.3.2, we find that the inconsistency is indeed caused by the dupOneinTail method which does not implement one of the preconditions (*i.e.*, the selected item can not be the head of the queue) in the specification.

4. Case study – railway track line automatic blocking scheme

In this section, we demonstrate the application of our specification-based monitoring technique with the railway track line automatic blocking scheme [3] of a railway control system. Due to the inherent characteristics of a reactive safety-critical system, it should be monitored by an independent monitoring system which can make quick and precise determination whether the observed behaviours are acceptable or not. Our monitoring system satisfies the above requirement, therefore, it can be applied to ensure the continuous correct behaviours of a reactive safety-critical system.

In the railway system, in order to avoid that the trains which run in the same direction on the same track crash into one another "from behind", the track is divided into segments with visible signals at the segment connections. The trains may pass a signal if there are no trains in the approaching segment(signal is set to green), or if it is some while ago that a previous train passed the segment(signal is then set to yellow). Otherwise, if the approaching segment is occupied by another train, the current train is blocked as the signal is set to red.

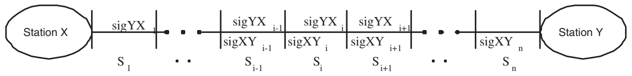


Figure 4. Automatic line signalling.

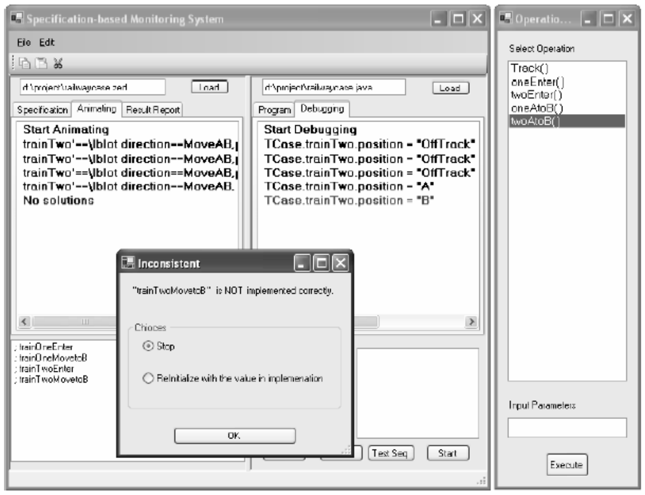
As shown in **Figure 4**, line l which connects exactly two stations: station X and station Y, is usually divided into segments $l = \langle s_1, s_2, ..., s_{i-1}, s_i, s_{i+1}, ..., s_n \rangle$. A line l can be in one of three possible states: OpenXY, OpenYX and Close. Each segment can be in two states: Free or Occupied. Segment s_i is in Free when no train is detected in the segment. Otherwise, segment s_i is in Occupied.

For each inner segment s_i,where i = $\langle 2, ..., n-1 \rangle$, there are two signals $sigXY_i$ and $sigYX_i$ which are for the two opposite directions of travel. Each signal is associated with four possible states: Red, Yellow, Green and Off.

Signal $sigXY_i$ is in Red state when line l is in OpenXY state and segment s_i is in Occupied state. It is in the Green state when line l is in OpenXY state and both segment s_i and s_{i+1} are in Free state. It is in Yellow state when line l is in the OpenXY state, segment s_i is in Free state and segment s_{i+1} is in Occupied state. It is in the Off state, when line l is in OpenYX or Closed state. Correspondingly, it is easy figure out the situations when signal $sigYX_i$ will be in the four different states.

For the first segment s_1 and the last segment s_n, there is only one signal $sigYX_1$ and signal $sigXY_n$, respectively. The signals in the opposite directions($sigXY_1$ and $sigYX_n$) are controlled manually, or by interlocking in the station [3].

As shown in **Figure 5**, with the formal specification and concrete implementation of the railway track line automatic blocking scheme loaded into our specification-based mon-



```
Start Animating
trainTwo'==\lblot direction==MoveAB,
trainTwo'==\lblot direction==MoveAB,
trainTwo'==\lblot direction==MoveAB,
trainTwo'==\lblot direction==MoveAB.
No solutions
```

```
Start Debugging
TCase.trainTwo.position = "OffTrack"
TCase.trainTwo.position = "OffTrack"
TCase.trainTwo.position = "OffTrack"
TCase.trainTwo.position = "A"
TCase.trainTwo.position = "B"
```

```
; trainOneEnter
; trainOneMovetoB
; trainTwoEnter
; trainTwoMovetoB
```

"trainTwoMovetoB" is NOT implemented correctly.

[✓ Stop] Reinitialize with the value in implemenation

Figure 5. Monitoring an aspect of a railway control system.

itoring system, the monitoring system works in running mode where the user will indicate what will be executed next in a step-by-step way. **Figure 5** also shows that the monitoring system finds an inconsistency and reports that operation *trainTwoMovetoB* is not implemented correctly. By checking the operation sequence, we know that the second train enters a segment while the first train is already on it. This is not allowed by the railway line automatic blocking scheme. Due to the page limitation, the full version of the formal specification in the Z specification language and the concrete implementation in Java programming language[1] are not included in this paper. The part of the formal specification which describes the above property of the railway system is shown as follows.

trainTwoMovetoB

$\Delta Track$

$trainTwo.position = A$
$status = OpenAB$
$trainTwo.direction = MoveAB$
$(segmentB.signalAB = Green$
$\quad \lor segmentB.signalAB = Yellow)$
$segmentA'.status = Free$
$segmentA'.signalAB = Yellow$
$segmentB'.signalAB = Red$
$segmentB'.status = Occupied$
$trainTwo'.position = B$
$trainTwo'.direction = trainOne.direction$
$trainOne' = trainOne \land status' = status$

[1]A complete version of the formal specification and the implementation of the railway track line automatic blocking scheme can be found at http://www.comp.nus.edu.sg/∼lianghui/spec-monitoring.pdf.

By checking the concrete implementation in Java, it is found out that the cause of the inconsistency is in method twoAtoB. As shown by the following code, the method twoAtoB which is supposed to implement the operation *trainTwoMovetoB* defined in formal specification, does not correctly implement the checking of signal as described by the specification. In the *if* statement of method twoAtoB, there should have been a substatement which checks the variable corresponding to the signal at the segment connection.

```
public void twoAtoB(){
   if(status == TrackState.OpenAB
     && trainTwo.position ==
                       TrainPosition.A
       && trainTwo.direction ==
                 TrainDirection.MoveAB)
   {segmentA.status = SegState.Free;
    segmentA.signalAB = SigState.Yellow;
    segmentB.signalAB = SigState.Red;
    segmentB.status = SegState.Occupied;
    trainTwo.position = TrainPosition.B;
    }
   }
```

5. Related work

Efforts have been put into the development of software monitoring techniques during the past few years. Zulkernine and Seviora [25] proposed a compositional approach to automatic monitoring of distributed systems, whose requirements specification is expressed in communicating finite state machines based formalism. Their monitor passively observes the external inputs, outputs, and partial internal states of a distributed system. With this information, the monitor interprets the specification of the target system and reports errors and failures. The main difference between the above work and our monitoring system is that we do not embed any extra observation code into the implementation, instead, we use the debugger to obtain the required information from the target system. Barnett *et al.* [2] presented a runtime monitor that uses executable specification to identify the behavioural discrepancies between a component and its specification. Like our monitoring system, the monitor developed by Barnett *et al.* can handle both deterministic and nondeterministic specifications and it does not need any instrumentation of the target components. However, to observe the dynamic behavior of the target system, it needs to snoop all the inter-component calls and returns.

Sankar and Mandal [17] have developed a methodology to continuously monitoring an executing Ada program for specification consistency. The user annotates an Ada program with constructs from Anna, a formal specification language. Those annotations are predicates that express constraints on Ada language constructs such as type, subtype, subprograms and exceptions. The compiler transforms the annotations into checking functions; and, in place of the transformed annotation, the compiler inserts a call to the checking function. The call statement acts as a sensor, observing the behavior of the target system; and the checking function judges the consistency. If any inconsistency occurs, diagnostic information is provided. In this approach, the instrumentation code can be installed automatically by the compiler. However, adding a new annotation leads to the recompiling of the target system. Moreover, there is potential that a large number of monitor will exist for a target system because a separate checking task is created for every annotation. The important difference between our monitoring technique and Sankar and Mandal's is that the implementation-dependent description of the monitored objects is separated from the high-level requirements specification of it in our monitoring system. Our monitoring system does not annotate the concrete implementation with any extra formal specifications. It gets the required information about dynamic behaviors of the formal specification and concrete implementation through animating and debugging respectively.

Our monitoring system can also help online testing. The basic idea of on-the-fly/online testing has been introduced in the context of labelled transition systems using ioco theory [20, 22]. Veanes *et al.* [23] developed a model-based testing tool named Spec Explorer, in which the model programs are written in the high level specification languages AsmL or Spec#. They formalized both the model programs and the implementation under test(IUT) as interface automata. The conformance relation between a model and an implementation is formalized as refinement between two interface automata. Correspondingly, the interface automata is used for conformance testing, including the handling of timeouts. Contrary to formalizing model program and IUT, in our monitoring system, the formal specification is animated and the consistency between the specification and implementation is judged based on the concrete data from animator and debugger, rather than based on the refinement between abstract models.

6. Conclusion and future work

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

Without embedding any instrumentation code into the target system, our formal specification-based online monitoring technique will not alter the running environment and the dynamic behaviours of the target system which is being monitored. Our monitoring technique gets required information about dynamic behaviors of the formal specification and concrete implementation of the target system through animating and debugging respectively, rather than by annotating the concrete implementation with extra formal specifications. Consequently, our monitoring technique realizes the separation between the implementation-dependent description of monitored object and the high-level requirements specification of it.

As an online monitoring technique, our formal specification-based monitoring technique dynamically gathers required information, interprets the gathered information and responds appropriately in a timely manner as the target system runs. It can contribute to increasing the dependability, correctness, robustness and security of the target system. It is competent for monitoring reactive safety-critical systems, and it also can contribute to effective dynamic test execution and test result checking.

In the future, we plan to extend our specification-based monitoring system so that it can work with the specifications written in Object-Z [4] and TCOZ [11], which are both extensions of Z and have strong capability for describing complex real-time system. The monitoring system based on them will be more competent. Monitoring distributed and parallel system during execution can provide information that can be used to reconfigure the system, provide visualization of behavior, or steer its outcome [19]. Therefore, we also intend to extend our monitoring technique so that it can handle distributed and parallel systems.

7. Acknowledgments

We would like to thank Dines Bjørner for many helpful discussions about the railway system.

References

[1] http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html.

[2] M. Barnett and W. Schulte. Spying on components: A runtime verification technique in specification and verification of component-based systems. In *Proc. of the Workshop on Specification and Verification of Component Based Systems - OOPSLA 2001*, 2001.

[3] D. Bjørner. *The SE Book: Principles and Techniques of Software Engineering*. Springer-Verlag, 2004.

[4] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[5] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *Software Eng. Journal*, 4(6):330–338, Nov. 1989.

[6] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the sum specification language. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, page 42. IEEE Computer Society, 1997.

[7] D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. In *ASE '98: Proceedings of the Thirteenth IEEE Conference on Automated Software Engineering*, page 302. IEEE Computer Society, 1998.

[8] M. A. Hewitt, C. O'Halloran, and C. T. Sennett. 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*, pages 37–51. Springer-Verlag, 1997.

[9] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[10] Y. S. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–979, 1992.

[11] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, Apr. 1998. IEEE Press.

[12] T. Miller and P. Strooper. A framework for systematic specification animation. Technical Report 02-35, The University of Queensland, 2000.

[13] T. Miller and P. Strooper. Model-based specification animation using testgraphs. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 192–203. Springer-Verlag, 2002.

[14] A. K. Mok and G. T. Liu. Efficient run-time monitoring of timing constraints. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 252, 1997.

[15] J. Peleska. Test automation of safety-critical reactive systems: Industrial application and future developments. In M.-C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, pages 538–556, New York, 1996. Springer-Verlag.

[16] I. Sanabria-Piretti. *Data Refinement by Rewriting*. PhD thesis, Department of computer science, Oxford University, 2001.

[17] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, 26(3):32–41, 1993.

[18] T. Savor and R. E. Seviora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real Time Technology and Applications Symposium*, 1997.

[19] B. A. Schroeder. On-line monitoring: A tutoiral. *IEEE Computer*, 28(6):72–78, 1995.

[20] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European International Conference on Software Testing, Analysis & Review*, 1999.

[21] M. Utting. Data structures for Z testing tools. In *Proceedings of FM-TOOLS*, 2000.

[22] M. van der Bij, A. Rensink, and J. Tretmans. compositional testing with IOCO. In *Formal Approaches to Software TEsting: Third International Workshop,, FATE2003*, pages 86–100, 2004.

[23] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proceedings of FSE/ESEC 2005*, 2005.

[24] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[25] M. Zulkernine and R. E. Seviora. A compositional approach to monitoring distributed systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2002.