Runtime Verification of Java Programs for Scenario-Based Specifications

Li Xuandong, Wang Linzhang, Qiu Xiaokang, Lei Bin
Yuan Jiesong, Zhao Jianhua, Zheng Guoliang

Department of Computer Science and Technology
Nanjing University, Nanjing, P.R.China

Abstract

In this paper, we use UML sequence diagrams as scenario-based specifications, and give the solution to runtime verification of Java programs for the safety consistency and the mandatory consistency. Our work can be used to detect

- the program bugs resulting from the wrong temporal orders of message interactions among objects, and
- the incomplete UML interaction models constructed in reverse engineering for the legacy systems,

Our work also leads to an automatic testing tool.

Outline

- UML sequence diagrams
- Scenario-based specifications
  (Safety consisteny, Mandatory consistency)
- Runtime verification process
- Program instrumenting and consistency checking
- Tool prototype and case studies
- Related work and conclusion

UML sequence diagrams

An UML sequence diagram describes an interaction, which is a set of messages exchanged among objects within a collaboration to effect a desired operation or result. Its focus is on the temporal order of the message interactions.

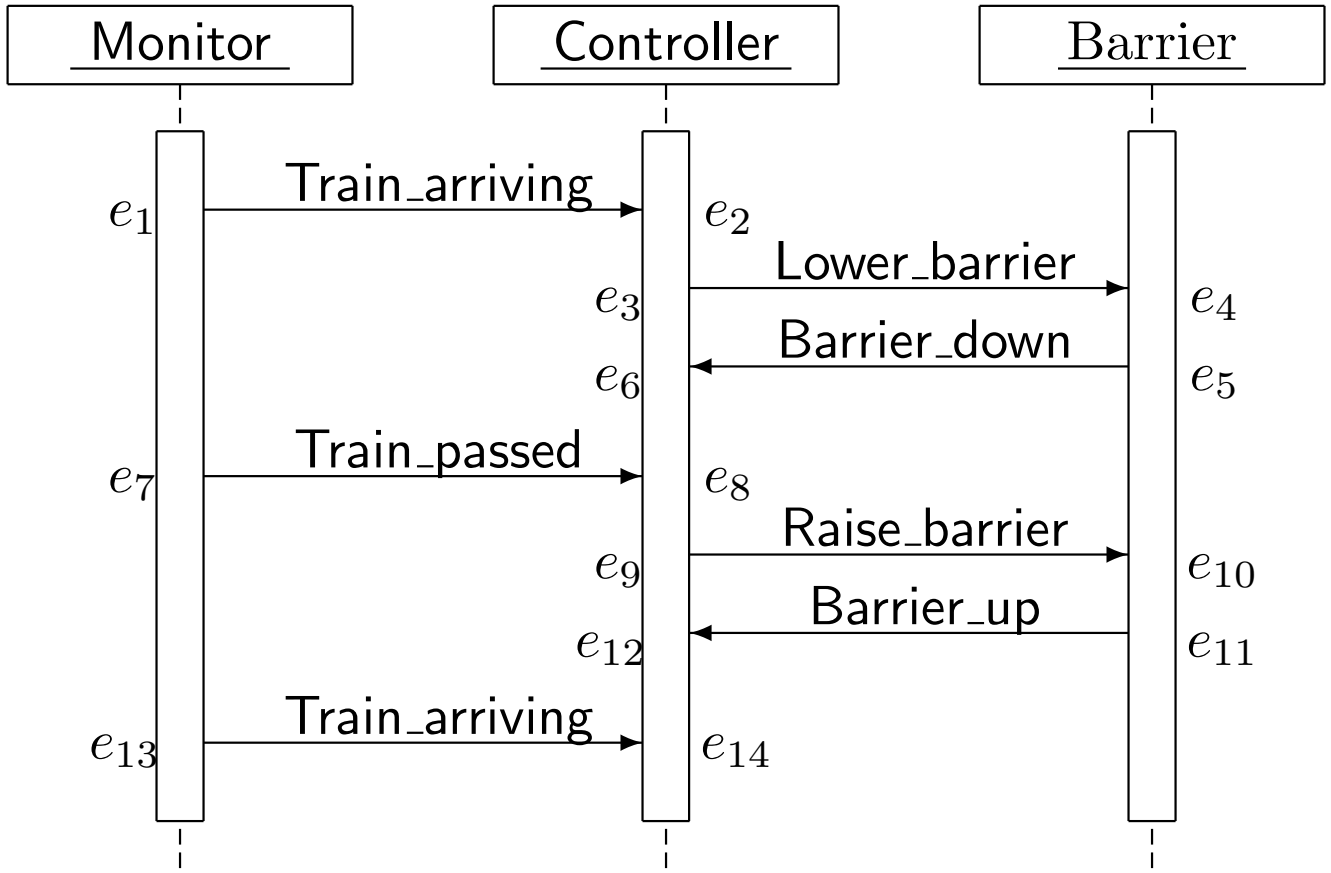


Figure 1: A simple UML sequence diagram describing the railroad crossing system

UML sequence diagrams

A sequence diagram is a tuple $D = (O, E, M, L, V)$ where

- O is a finite set of objects. E is a finite set of events corresponding to sending or receiving a message.
- M is a finite set of messages. Each message in M is of the form (e, g, e') where $e, e' \in E$ corresponds to sending and receiving the message respectively, and g is the message name which is a character string.
- $L : E \to O$ is a labelling function which maps each event $e \in E$ to an object $L(e) \in O$ which is the sender (receiver) while e corresponds to sending (receiving) a message.
- V is a finite set whose elements are a pair (e, e') where $e, e' \in E$ and e precedes e', which is corresponding to a visual order.

UML sequence diagrams

We use *event sequences* to represent the *traces* of sequence diagrams, which describe the temporal order of the message interactions. For any sequence diagram $D = (O, E, M, L, V)$, an event sequence

$$e_0 \hat{} e_1 \hat{} \ldots \hat{} e_m$$

is a *trace* of D if and only if the following condition holds:

- all events in E occur in the sequence, and each event occurs only once, i.e. $\{e_0, e_1, \ldots, e_m\} = E$ and $e_i \neq e_j$ for any i, j $(0 \leq i < j \leq m)$; and
- $e_0, e_1, \ldots, e_m$ satisfy the visual order defined by V, i.e. for any e_i $(0 \leq i \leq m)$ and e_j $(0 \leq j \leq m)$, if $(e_i, e_j) \in V$, then $0 \leq i < j \leq m$.

The problem we concern

The problem we concern is to check to if the program execution traces are consistent with the traces of the given sequence diagrams.

Scenario-Based Specifications

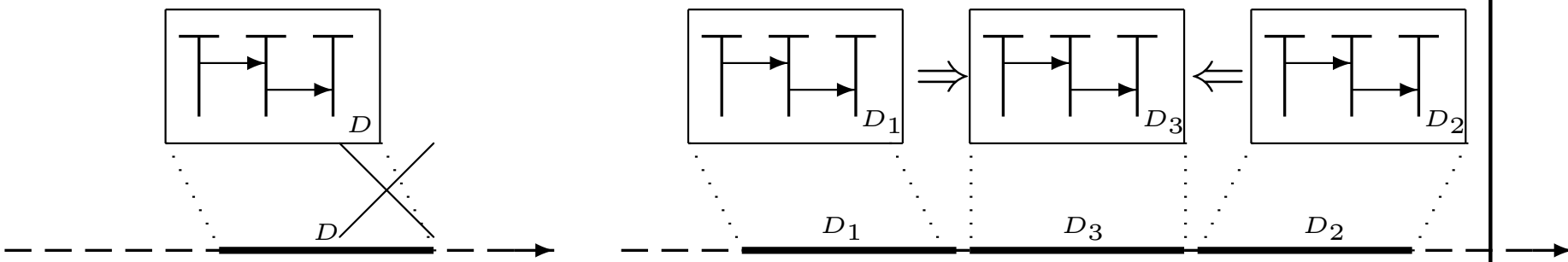
- Safety consistency specifications
  Safety consistency specifications require that any forbidden scenario described by a given sequence diagram never happens during the execution of a program.
- Mandatory consistency specifications

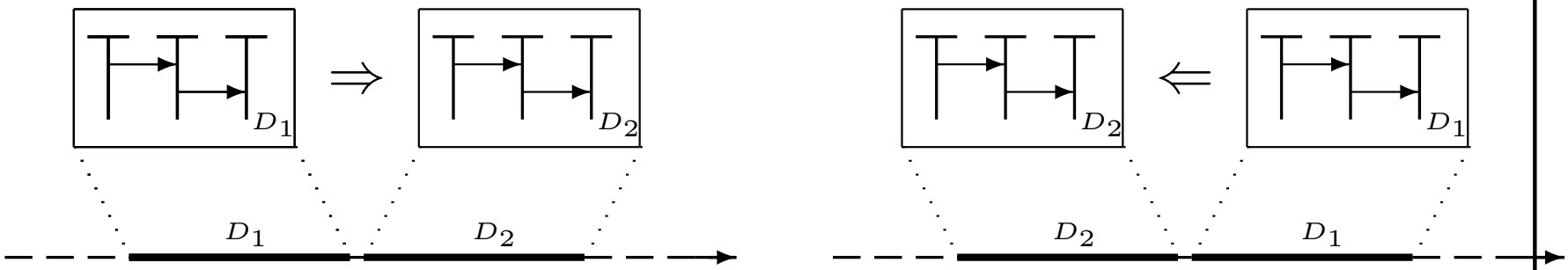
Mandatory Consistency Specifications

Mandatory consistency specifications requires that if a reference scenario described by the given sequence diagrams occurs during the execution of a program, it must immediately adhere to a scenario described by the other given sequence diagram.

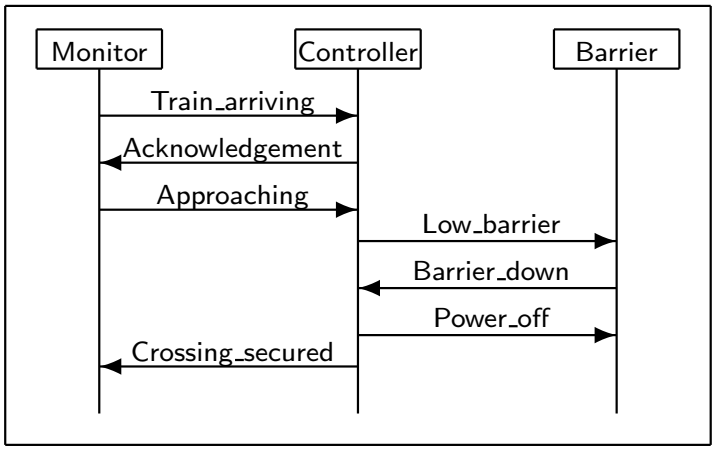
- Forward mandatory consistency specifications
- Backward mandatory consistency specifications
- Bidirectional mandatory consistency specifications

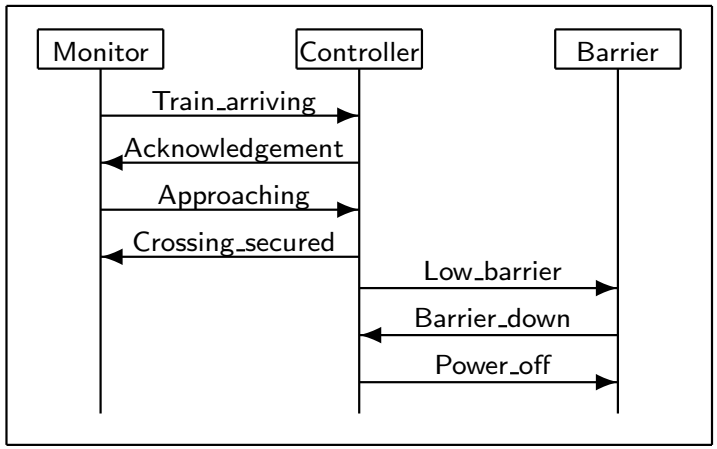
Scenario-based specifications



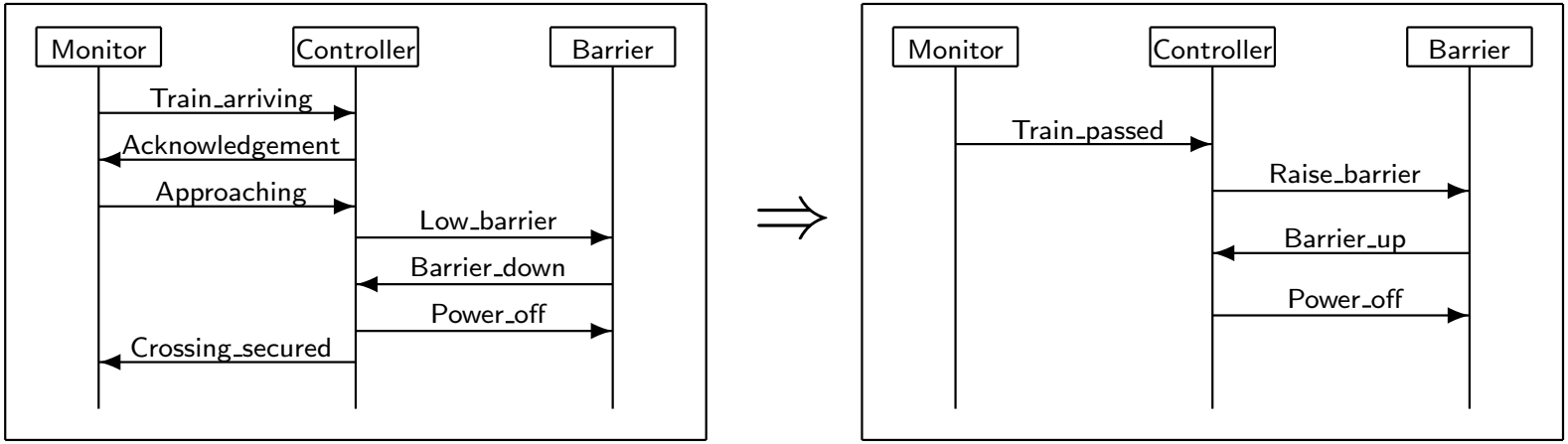


Safety consistency specifications for RCS

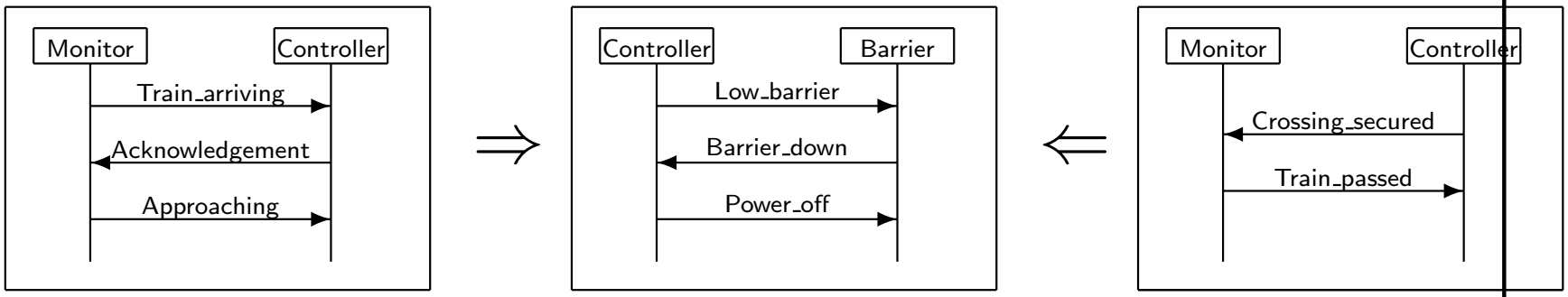




Forward mandatory consistency specification for RCS



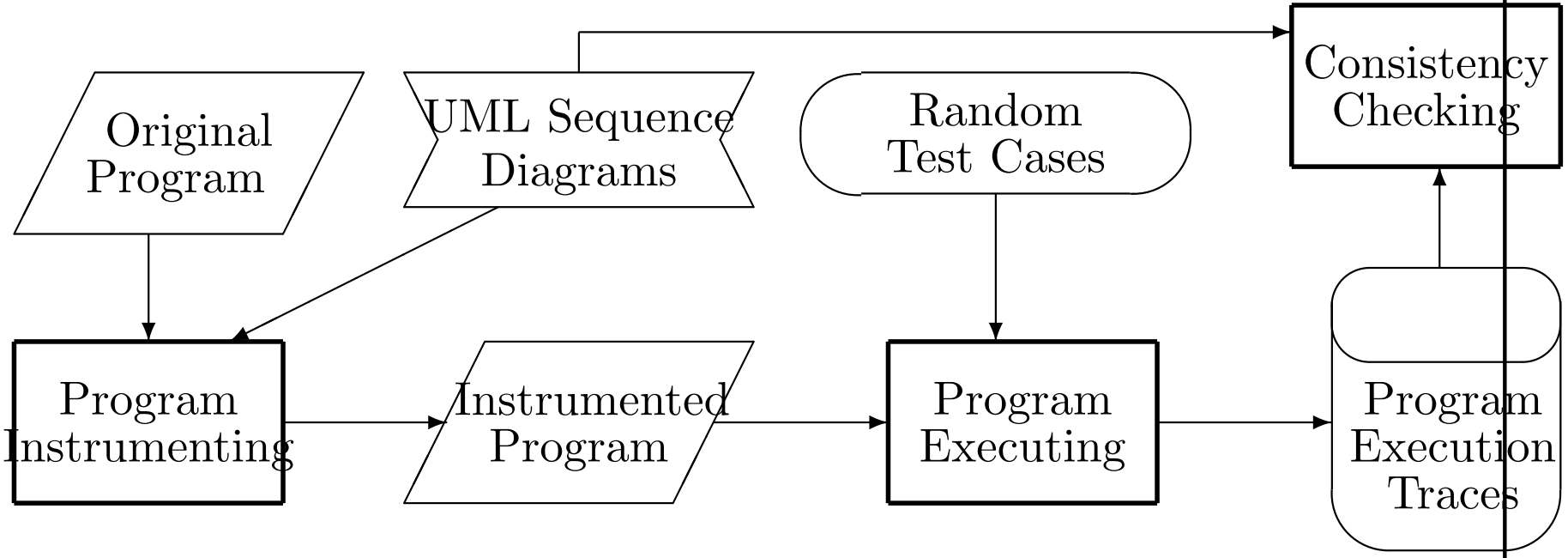
Bidirectional mandatory consistency specification for R



Runtime verification

- Instrument the program under verification so as to gather the program execution traces related to a given scenario-based specification.
- Drive the instrumented program by random test cases so as to generate the program execution traces.
- Check if the collected program execution traces satisfy the given specification.

Runtime verification process



Program instrumenting

- For a Java program under verification, we need to insert some statements into its source code for gathering the program execution traces.
- Since the scenario-based specifications we consider are represented by the sequence diagrams, the program execution traces we gather are a sequence of events corresponding to sending and receiving messages.

Program instrumenting

In a Java program,

- a method call is corresponding to a message sending event, and
- the first statement execution in a method is corresponding to a message receiving event.

Thus we insert the statements for gathering the information around each related method call and in the beginning of each related method definition.

Program instrumenting

The function of the inserted code segments is as follows.

- When a sending or receiving event for a message in a program happens, the information we need to log include the message, its sender or receiver, and the class which the sender or receiver belongs to.
- Since an object may send or receive the same message many times, we need to pair a sending event and its corresponding receiving event for the same message.

Consistency checking

For a given scenarion-based specification, consistency checking is

- to match the program execution traces and the traces of the given sequence diagrams in a scenario-based specification, and
- to check if the collected program execution traces satisfy the given specification.

Consistency checking

Since the different objects with the same class may occur in a program execution trace, for a given sequence diagram $D = (O, E, M, L, V)$ there may be multiple object compositions corresponding to O in the program execution trace.

Thus, when consistency checking we should consider the scenarios generated by those object compositions respectively.

Support tool

With the work presented in this paper, we aim to develop an automatic support tool for testing, which may help us to reduce the testing cost.

- The tool can help us to detect the inconsistency between the behavior implemented by the program and the expected behavior specified by the scenario-based specifications.
- The tool may proceed in a fully automatic fashion, and we can drive it after we leave our office in the evening, and see the results in the next morning.

Tool prototype

We have implement a prototype of this kind of tool. The tool accepts a Java program under verification, instruments the program according to the given scenario-based specifications, drives the instrumented program to execute on a set of random test cases, and reports the errors which result from the inconsistency with the specification.

Case studies

- Automated teller machine simulation system
- Microwave oven simulation system (17 classes, 113 methods)
- Official retirement insurance system (17 classes, 241 methods)

In these case studies, in addition to the bugs embedded manually, by the tool we did find out several inconsistent cases resulting from the wrong temporal orders of message interactions or the incomplete sequence diagrams we use as the specification.

Since the algorithms for program instrumenting and consistency checking are simple and efficient, we think there is no particular obstacle to scale our approach to larger systems.

Related work

- Model checking for Java programs
- Live sequence charts
- UML model based testing
- Runtime verification of Java Programs for deadlocks and data races

Conclusion

- We give the solution to runtime verification of Java programs driven by UML interaction models, which focus on the temporal order of message interactions among objects.
- The underlying approach and ideas in our work are more general and may also be applied to the runtime verification of the other object-oriented programs.