Available online at www.sciencedirect.com
SCIENCE DIRECT®
Microprocessors and Microsystems 27 (2003) 523–531
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

Generating test sequences using symbolic execution for event-driven real-time systems

Nam Hee Lee*, Sung Deok Cha

Division of Computer Science and AITrc, Department of Electrical Engineering and Computer Science, KAIST, 373-1, Gusung-Dong, Yusung-Gu, Daejon, South Korea

Received 11 November 2002; accepted 18 April 2003

Abstract

Real-time software, often used to control event-driven process control systems, is usually structured as a set of concurrent and interacting tasks. Therefore, output values of real-time software depend not only on the input values but also on internal and nondeterministic execution patterns caused by task synchronization. In order to test real-time software effectively, one must generate test cases which include information on both the event sequences and the times at which various events occur. However, previous research on real-time software testing focused on generating the latter information. Our paper describes a method of generating test sequences from a Modechart specification using symbolic execution technique. Based on the notion of symbolic system configurations and the equivalence definitions between them, we demonstrate, using the railroad crossing system, how to construct a time-annotated symbolic execution tree and generate test sequences according to the selected coverage criteria.

Keywords: Real-time system testing; Symbolic execution; Modechart

1. Introduction

Dependable real-time software must produce functionally correct results within the specified time intervals. Validation of real-time software can be particularly difficult since it often consists of cyclic and interacting tasks, exhibiting a large number of nondeterministic execution patterns for a given input set. Formal verification and testing is well-known and complementary software quality assurance approaches. Although formal verification made impressive technical advances to the degree that they are used on large and complex industrial projects [1,2], it does not eliminate the need for testing. It is our strong belief that testing will always remain an essential and indispensable component of any software quality assurance program.

Many testing techniques have been proposed, but relatively few results are provided for real-time software. Input values alone are sufficient as test cases for sequential software, and input sequences which can exercise a sequence of synchronization are necessary for concurrent software. However, for typical event-driven real-time software, repeated runs using the same input values and sequences do not necessarily follow the identical execution paths and produce the same results. Therefore, both the event sequences and the time of each event occurrence must be included in test cases for real-time software.

There are only a few works on the testing of the temporal behavior of real-time systems [3–5]. In Ref. [3], a technique for generating test cases from TRIO specification is introduced. It extends the classical temporal logic to deal explicitly with time measures, and TRIO formulas can be automatically checked for satisfiability or validity. During the interpretation of a formula F specifying a property of a system, behaviors of the system compatible with F are generated: they are called *histories*. These histories are used as test cases. In Ref. [4], a system is modelled with a formally defined SA/SD-RT notation and translated into the time reachability tree for representing the behavior of the system. Each path from the root of the tree to its leaves represents a potential test case.

In Ref. [5], a technique for testing timing constraints of real-time systems is presented. A constraint graph is used for describing the various timing constraints the system

* Corresponding author.
E-mail address: nhlee@salmosa.kaist.ac.kr (N.H. Lee).

0141-933/03/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0141-9331(03)00102-9

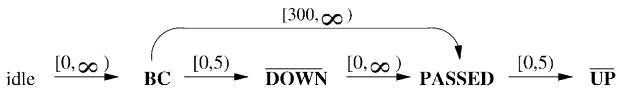

Fig. 1. A constraint graph for railroad crossing.

must satisfy. Timing constraints among various events are shown on the edges of the graph. For example, the constraint graph shown in Fig. 1 indicates that the system is initially in the idle state and that the input event BC, indicating the detection of a train approaching the crossing, may occur at an arbitrary and unknown time $[0, \infty)$. It also shows that the system is required to generate the output event $\overline{DOWN}$ within 5 time units following the receipt of the BC. Similarly, at an arbitrary time in the future, an input event PASSED is expected to occur. However, the occurrences of the events BC and PASSED must be separated by at least 300 time units. Finally, it is a system requirement to generate the output event $\overline{UP}$ within 5 time units following the occurrence of the input event PASSED.

Clarke [5] treated the time interval as another input domain and used traditional domain testing technique. While constraint graph-based approach is useful in generating test cases for real-time software, Clarke did not address how constraint graphs can be automatically generated from a formal specification. This paper bridges the gap by illustrating how such task can be accomplished on a Modechart specification [6] (Fig. 2).

Some authors have proposed coverage criteria to the problem of testing real-time software [3–5,7]. In Ref. [7], test adequacy criteria based on coverage measures of Petri nets topology for concurrent and real-time systems are presented. Specifically, these criteria are based on firing or transition coverage over Petri nets. In Ref. [3], since the actual test case is a subset of the allowed traces of the system, some heuristic techniques are defined to select a subset of all possible test cases for a given specification. These criteria are based on the constructs of the TRIO

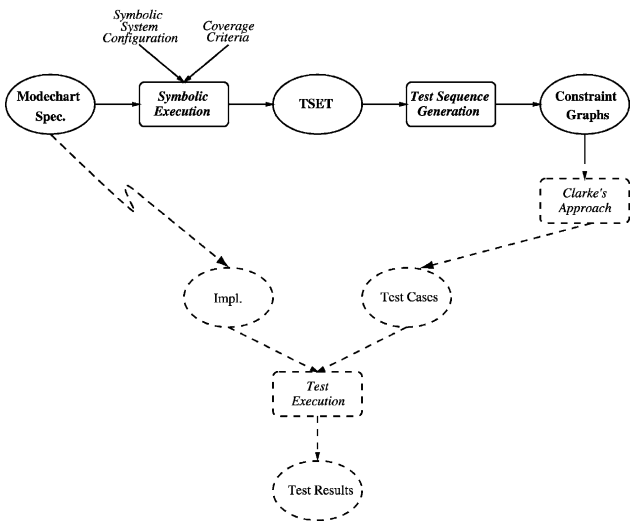

Fig. 2. Our approach for generating test sequences using symbolic execution.

specifications. Ref. [4] describes how to restrict test cases according to different coverage criteria because the time reachability graph would become large even for small-scale systems.

In this paper, we propose coverage criteria designed to test Modechart specification and discuss how symbolic execution technique can be used to generate test cases. We choose Modechart because it provides a rich set of visual constructs with which one can represent various modes the system components can be in and timing constraints among them. We use symbolic execution technique, which treats the time as symbolic values, for generating test sequences. The notion of symbolic system configuration is defined, and a time-annotated symbolic execution tree (TSET) is generated by symbolically executing Modechart specification. Finally, the test sequences are generated from TSET based on the selected coverage criteria and represented as the constraint graph.

The rest of our paper is organized as follows. Modechart specification language is briefly introduced in Section 2. Some coverage criteria of event-driven real-time systems are defined in Section 3, and procedures for constructing TSET are explained in Section 4. We demonstrate, using the railroad crossing system, how to generate constraint graphs. Section 5 concludes the paper.

2. Modechart

Modechart is a visual language devised to specify the behavior of real-time systems [6]. Modechart constructs include *modes*, which are analogous to states in Statecharts [8]; *actions*, which assign values to data variables and require at least one time unit to complete; and *events*, which are instantaneous. Events can be classified into *external*, *mode entry*, *mode exit*, *start*, and *stop* events. External events represent changes in the system environment, mode entry and exit events mark entry into or exit from a mode, and start and stop events mark the start and stop of an action. Modechart borrows compact Statecharts notations for representing concurrent states and provides constructs to denote various timing requirements such as delays, deadlines, and time intervals. In addition, Modechart uses a discrete time model: its delay and deadlines are represented as non-negative integers. Modechart formalism is supported by a software toolset including an editor, a simulator, a verifier, and a code generator [9].

In Modechart, modes may be composed in either serial or parallel manner. Modechart's notion of serial and parallel modes corresponds to OR and AND composition in Statecharts. If M is a *serial* mode with child modes M_1 and M_2, at any given time the system can be in exactly one of M_1 and M_2. If M is a *parallel* mode with child modes M_1 and M_2, then when the system is in M, it is simultaneously in both modes M_1 and M_2. Mode transition, indicating a change