Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER Electronic Notes in Theoretical Computer Science 113 (2005) 65–83
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Enforcing Concurrent Temporal Behaviors

Doron Peled and Hongyang Qu

Department of Computer Science
The University of Warwick
Coventry, CV4 7AL, UK

Abstract

The outcome of verifying software is often a 'counterexample', i.e., a listing of the actions and states of a behavior not satisfying the specification. In order to understand the reason for the failure it is often required to test such an execution against the actual code. In this way we also find out whether we have a genuine error or a "false negative". Due to nondeterminism in concurrent code, recovering an erroneous behavior on the actual program is not guaranteed even if no abstraction was made and we start the execution with the prescribed initial state. Testers are faced with a similar problem when they have to show that a suspicious scenario can actually be executed. Such a scenario may involve some intricate scheduling and thus be illusive to demonstrate. We describe here a program transformation that translates a program in such a way that it can be verified and then reverse transformed for testing a suspicious behavior. Since the transformation implies changes to the original code, we strive to minimize its effect on the original program.

Keywords: Behavior monitoring, Concurrency, Counterexample analysis, Model Checking, Nondeterminism, Temporal Logic, Testing.

1 Introduction

Verification is used to pinpoint the existence of errors in software. The outcome of the verification process is often given by listing the sequence of atomic actions and global states comprising a behavior not satisfying the specification, called a *counterexample.* Since it is often not the code itself that is being verified, but rather a model of it, there is a non negligible likelihood of encountering a 'false negative'. That is, an execution that does not conform with a behavior of the actual program. Since false negatives occur frequently, an execution suspicious of being faulty needs to be carefully checked. Unfortunately, executions of concurrent programs may be quite long and complicated when manually analyzed. Testers face a similar problem when they are required to show that some suspicious behavior actually occurs during some run of the code. Given a behavior that appears under

1571-0661/$ – see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.entcs.2004.01.034

some uncommon scheduling, it may be very hard for a tester to enforce the tested program to execute it.

We are interested here in causing the checked code to behave according to a given suspicious execution, which may be the result of a verification or testing effort. We want to be able to reconstruct and inspect this behavior in the context of the tested or verified code. Due to nondeterminism associated with concurrently executing events, we are not guaranteed to recover the given execution without enforcing some modification to the checked software. We therefore concentrate on minimizing the effect of the changes to the original code. We suggest a simple and automatic transformation that can be applied to the code in order to recover the suspicious behavior. Our method gives the verification engineer or tester a tool for checking and demonstrating the existence of the error in the code. We impose the following constraints on the suggested software transformation:

- Minimize the changes to the software. We want to deviate as little as possible from the original program.
- Enforce the required execution exactly when choosing an appropriate initial state. Other executions are still available when this initial state is not selected.
- Any concurrency or independence between executed actions is preserved. We are reconstructing a concurrent execution, rather than a completely synchronized one in which one action is executed at a time. Hence our solution is *distributed* rather than centralized.
- Preserve the checked property. Not all the execution sequences with the same concurrent structure as the original counterexample necessarily satisfy the same temporal properties.
- Apply the construction to a finite representation of infinite execution, i.e., an ultimately periodic sequence.

Although our transformation will be demonstrated for a given (Pascal-like) syntax, it is language-independent. We discuss some language and implementation issues in Section 7.

Our architecture is shown in Figure 1. In order to perform the verification, the program is translated into a finite set of atomic actions. This translation may also include an abstraction that simplifies the verified model. Another component of the translation is a dependency relation, which includes pairs of actions that cannot be executed concurrently, e.g., due to the use of a shared variable or a message queue. Finally, a third component of the translation is an annotated code, which has pointers to various locations in the text, specifically the beginning and the end of the text of the code related to atomically executed actions. These pointers are useful in adding new instructions (or changing the existing ones) in a way that will enforce executing the suspicious behavior.