Verification and Testing of Concurrent Systems with Action Races

Andreas Ulrich[a], Alex Petrenko[b]
[a] Siemens AG, CT SE 1, München, Germany, <andreas.ulrich@mchp.siemens.de>
[b] CRIM, Montréal, Canada, <apetrenk@crim.ca>

1.0 Development and Test of Software Systems – An Overview

The production process of a software system comprises the following phases that might be repeated several times (spiral process model of software): analysis of system requirements, architectural and functional design of the system, implementation, and testing [1].

Only in recent time, formal description techniques (FDTs) obtain some acceptance in the requirement and design phases of software systems. The high complexity of software systems and a lack of appropriate modeling techniques for different design aspects (e.g. models for dynamic and static aspects, models for data base operations or human computer interactions etc.) largely contribute to this fact. That is why even when FDTs are used they reflect only some aspects of the system's properties.

A currently much used requirement and design language is the Unified Modeling Language (UML) that actually comprises a set of different notations, e.g. use cases, sequence diagrams, state charts, activity diagrams, and others [2]. A drawback of the current UML version is its, in some respects, informal semantics that renders the development of tools to provide automation in the software production process difficult. However improvements in this respect are expected in the near future.

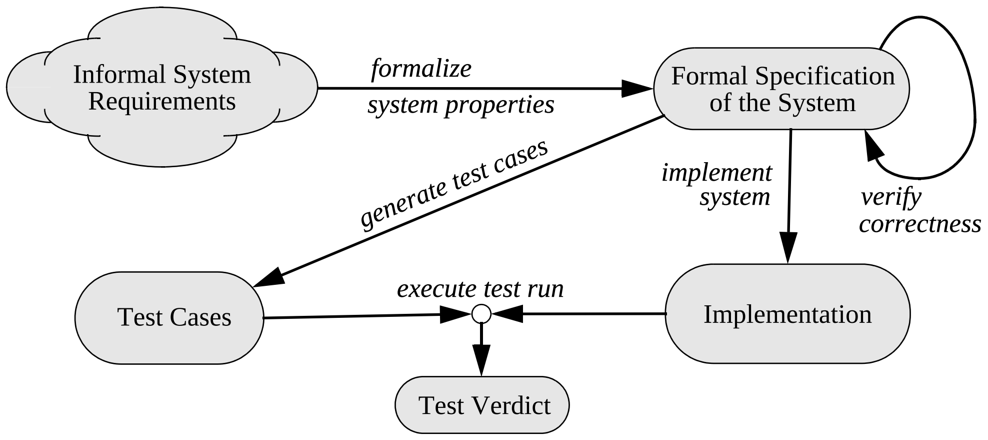

FIGURE 1. Design and Test Process of Software Systems

Figure 1 depicts the typical design and testing approach for software systems using FDTs. First, system requirements are formalized to avoid ambiguity in the following phases. The obtained formal description of the system specifies the behavior of the system at the interface to its environment, e.g. at the graphical user interface (GUI), and is subject to multiple refinement steps to obtain a detailed model of the system as a starting point for the implementation.

Each refinement step can be verified against standard properties (liveness, safety) and other user-defined properties. When the designer is convinced with the behavior of her/his model, the implementation of the system is started. The implementation process can be supported by some semi-automatic tools that generate parts of system, e.g. CASE (computer-aided software engineering) tools like Rational Rose, Together, or Rhapsody that all use UML as modeling language. However, still large portions of the system are implemented manually. Therefore, testing the implementation against the original requirements (*conformance tests*) is an important part in the software production process to ensure the final quality of the system. This test method is propagated by an own ISO standard [3] and is vitally important to systems that rely on interoperability with other ones, e.g. telecommunication appliances.

The goal of testing is to show conformance of the implementation of the system with its requirements. Testing differs from verification in the way that it deals with the real system instead with the system's model only. The testing process itself distinguishes several activities:

- *Test derivation*: derives test cases from the system specification; can be done automatically by tools if formal specifications are used;
- *Test execution*: executes test cases in a controlled test environment (test architecture); can be well automated by tools in most cases;
- *Test result evaluation*: checks the result obtained after a test run and assigns a test verdict *fail*, *pass*, or *inconclusive* to the test run of a certain test case; can be automated.

The following table compares testing with verification as two main validation techniques to ensure system quality [4].

Verification	**Testing**
• performed on the (partially known) model	• performed on the concrete system
• proves properties	• shows errors (deviation from requirements)
• uses rigid state exploration as main technique	• does selective state exploration
Verification is only as good as the validity of the model on which it is based!	**Testing can show only the presence of errors, not their absence!**

Though verification can help to discover errors of the system early in the development process, its results can be only as good as the model is. If, for example, the model contains missing or wrong properties, verification might produce false results. Since testing deals always with the real system, such modeling faults can be detected. Furthermore testing can reveal even errors that were introduced during the implementation phase that are completely out of context for model-based verification.

On the other side, both verification and testing use similar techniques for state exploration in the model or in the concrete system to prove properties or to discover errors. Although verification explores the state space systematically, the quality of the obtained results depends very much on the completeness of the model. Testing again is more restrictive in covering the state space since test execution is a very time and cost sensitive activity. Therefore, test selection criteria (*fault coverage criteria*) are used that promise to discover certain types of errors [5].

2.0 Test Execution for Concurrent Systems

Testing concurrent systems is impeded by a number of problems. Besides state space explosion that hampers the derivation of test cases, nondeterminism during test execution is another main concern. Nondeterminism makes the repeatability of test runs hard and should therefore be avoided in a chosen test architecture. It exists in three different facets: (1) the environment of the system, i.e. the tester, cannot predict the output after sending an input to the system (internal nondeterminism); (2) the execution order of concurrent actions is arbitrary and cannot be controlled by a tester; (3) external and internal actions of the system are executed with different speeds leading to action races.

All three types of nondeterminisms must be mastered during test execution. The remaining part of the presentation will concentrate on nondeterminism due to action races. It is a summary of a paper published in [6].

2.1 Action Races as a Property of Concurrent Systems

Races are a form of nondeterminism that is often inherent to concurrent systems. Depending on the execution order of actions, the global state of the system reached after a race is ambiguous. In general, races are a source of software faults that are difficult to detect during test execution. They require repeated test runs of the same test case with varying execution speeds of test events. As an alternative to testing, possible races can be analyzed in a formal specification of the concurrent system within the design phase. However, general verification methods do not pay particular attention to race conditions and other types of nondeterminism since they evaluate all possible execution sequences of the system in a systematic manner. Therefore, analysis techniques to detect race conditions are needed. Work on race analysis started first in the context of sequential circuits [7].

Intuitively, an action race condition arises when in a given global state several actions are enabled. A race condition leads to a race when, as a result, the system may reach different destination (*stable*) states from where only external actions are enabled. Depending on the types of actions creating race conditions, we distinguish three types of race conditions:

- internal actions race conditions or i-race conditions;
- external actions race conditions or e-race conditions;
- mixed (internal and external) race conditions or m-race conditions.

Autonomous (closed) systems may have only i-race conditions. A given (open) concurrent system may or may not have a particular type of race conditions if it is put into different environments. Accordingly, we define three basic types of environments depending on its ability to offer either single or several concurrent actions at a time when the system is in a stable or in any (stable or transient) state.

- An environment that submits a single external action at a time only when the concurrent system is in a stable state is called a *sequential-slow environment*. Here only i-race conditions can occur.
- An environment that may submit external actions sequentially even before the concurrent system has reached the next stable state is called a *sequential-fast environment*. M-race conditions (in addition to i-race conditions) occur in a system within such an environment.
- Finally, if the environment can offer simultaneously several external actions starting from a stable state, it is called a *concurrent environment*. Additionally, e-race conditions may arise within such an environment.

In this presentation, we restrict the discussion to testing in sequential-slow environments [6]. We assume that the model of a concurrent system is given as a set of communicating finite state machines (FSMs) communicating synchronously with each other. The global behavior of a concurrent system can thus be represented by the composite machines obtained as a synchronous product from all FSMs in the model.

A sequential-slow environment waits until a concurrent system completes the execution of internal actions before it offers a next external action, i.e., it prioritizes the execution of internal actions over external ones. The behavior of a concurrent system that is controlled by a sequential-slow environment can be represented by a *stable composite machine* (SCM), a proper submachine of a composite machine. The number of states in a SCM may reach that of a composite machine in the worst case situation, is however much lower in typical examples.

I-race conditions exist in such a system if different internal actions are enabled in a global state that lead to different destination states, i.e., the SCM is nondeterministic. This type of action races is completely out of control of the environment. If a given specification has races within a sequential-slow environment, then we may conclude that either there exists a design error or the specification is, in fact, an abstraction of a future design and has to be refined further.

2.2 Steady Testing

The sequential-slow environment is a very much used assumption in the definition of an appropriate test architecture since it reflects common observations for GUI-based applications: a system is most of the time waiting for new inputs from the environment. Testing under the sequential-slow environment assumption is called *steady testing*. Test cases can be derived from the stable composite machine of the system.

Deterministic test cases exists only when i-race conditions in the specification do not lead to races, i.e., the SCM is deterministic. Furthermore, the tester must guarantee that the system under test always returns into a stable state before the next external action of the test case is applied. This behavior is achieved when a *delay* is inserted after each action of the test case.

Note that test cases are derived from the largely reduced SCM instead from the composite machine. This leads to tests of only limited fault detection capability. In fact, some faults of the system, even when they are present in the model, might never be detected using steady testing because states, where such faults can be observed, are not reachable. It follows that steady tests must be supplemented by tests using less stringent assumptions, e.g. sequential-fast or concurrent environments. Then other types of races might occur that must be tackled by the tester.

3.0 Conclusions

The first part of the presentation was an outline of today’s software production process. In particular the interrelation between verification and testing as two different activities that are both justified to coexist was highlighted. Then the presentation continued with the special issue of action races as an inherent property of concurrent systems. Such races occur when internal and external actions of the system are enabled in certain global states that result in different end states. Most likely, races cannot be completely avoided in the design of a concurrent system. In spite of this, races have to be identified to take preventive measures in the implementation of the system. The issue of races is of interest too when it comes to testing the concurrent system.

Tests using stringent environment assumptions, e.g. sequential-slow environments, have their attractions because they avoid most of the possible nondeterminism contained in a system, i.e., such tests are more likely to be repeatable with the same test results. On the other side, they detect only a limited number of faults and must therefore be supplemented with further tests. This necessity arises in particular from stress testing the system or when the sequential-slow environment assumption cannot be guaranteed in reality, e.g. for embedded systems. Since race conditions can be analyzed already at the specification level if a model exists that is sufficiently specified, verification methods can be proposed such that similar tests at implementation level become less important.

4.0 References

[1] C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering*; Prentice-Hall, 1991; ISBN 0-13-818204-3.

[2] M. Fowler: *UML Distilled*; Addison-Wesley Longman, 1997; ISBN 0-201-32563-2.

[3] D. Hogrefe, St. Heymer, J. Tretmans: *Report on the standardization project "Formal Methods in Conformance Testing"*; in: Baumgarten, et.al. (eds.): Testing of Communication Systems (IWTCS '96), Darmstadt; Chapman & Hall, 1996; pp. 289–298.

[4] J. Tretmans: *Specification Based Testing with Formal Methods: From Theory via Tools to Applications*; in: A. Fantechi (ed.): FORTE / PSTV 2000 Tutorial Notes; Pisa, Italy, Oct. 10 2000; http://fmt.cs.utwente.nl/publications/tretmans.pap.html.

[5] A. Petrenko: *Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography*; in: F. Cassez, C. Jard, B. Rozoy, M. Ryan (eds.): Proceedings of Summer School MOVEP'2k Modelling and Verification of Parallel Processes; Nantes, July 2000.

[6] A. Petrenko, A. Ulrich: *Verification and Testing of Concurrent Systems with Action Races*; in: H. Ural, R. L. Propbert, G. v. Bochmann: Testing of Communicating Systems (TestCom'00); Ottawa; Kluwer Academic Publishers, 2000; pp. 261–280.

[7] J. A. Brzozowski, C.-J. Seger: *Asynchronous Circuits*; Springer-Verlag, 1994.