Methods for Increasing Software Testability Extended abstract

Birgitta Lindström, Jonas Mellin, and Sten Andler

Testing is the act of executing a system or application, in order to show the presence of faults. The confidence in the system can thereby increase as faults are removed. It can, however, never be used to prove that the system is correct. The reason for this is that it is generally infeasible to conduct exhaustive testing, since the number of test cases to test all possible outcomes is too high. It is a well-known fact in software engineering that the cost for testing of software often is 50% or more of the development costs. Hence, methods to improve testability, i.e., to reduce the effort required for testing, have a very large potential to decrease development costs.

In this paper, controllability and observability, (Schütz 1993), are considered necessary prerequisites for testability. Without the ability to control test execution, and the ability to observe important outcomes such as state transitions, it is not feasible to reach a high level of testability. A system, which lacks controllability and observability, is always difficult to test. However, testability also depends on the number of possible execution orders (i.e., interleavings of threads). An often neglected circumstance which is especially important for distributed or real-time systems. The reason for this dependency is that observability and controllability only apply to test cases that are actually executed. Given a certain amount of test effort, the test coverage decreases with an increase in the number of distinct execution orders and hence, test cases. Achieving higher test coverage (i.e., executing a higher ratio of all possible test cases) increases the required test effort, since it requires designing, executing, and analyzing more test cases (Birgisson, Mellin and Andler 1999).

As we have shown that testability is dependent of the number of distinct execution orders, it is a natural step to investigate the execution environment and its impact on the number of execution orders. Which execution orders are allowed depends to a large degree on processor scheduling and concurrency control policies. In a survey and analysis of current methods for improving testability (Lindström 2000), several such methods are investigated and their properties with respect to processor scheduling and concurrency control analyzed. Especially, their impact on the number of distinct execution orders is discussed. The survey reveals that (*i*) there are few methods which explicitly address software testability, and (*ii*) methods that concern the execution en-

vironment for real-time systems require or favor a time-triggered design.

Kopetz (1991) points out that testability depends on the architecture and must be considered during the design phase. The first comparison of time-triggered and event-triggered systems with respect to testability is made by Schütz (1993). Schütz shows that the effort to test an event-triggered real-time system is inherently higher than that of a time-triggered real-time system. This is due to the dynamic nature of event-triggered systems, which makes the number of distinct execution orders high in comparision to time-triggered systems.

A time-triggered design is, however, not always suitable. The reasons to choose an event-triggered design include the need for a flexible system in unpredictable environments. Usually there is a need for graceful degradation. Therefore, it is an interesting issue to investigate which properties of the execution environment in event-triggered systems that lead to high or low levels of testability. Some of these properties are identified by Lindström (2000) and others remain as open problems. These properties are used to define categories, which form a basis of taxonomy for testability.

It is important to note here that we have no intention to discriminate between good and bad solutions based on testability alone. We are, however, interested in the impact on testability from different properties of the execution environment, as design is always a matter of trade-off decisions. The important thing is that these decisions should be based on as much information about the consequences of the choices as possible.

The scheduling properties that are investigated by Lindström (2000) are preemption policy, task priority policy, and observation policy. It is shown that there is a significant influence on testability from the choice of scheduling properties. One example is the priority policy, which has two dimensions of complexity that do have an impact on the test effort:

1. Whether the task priority is unique during the execution of the task,
2. Whether the priority ordering between two tasks can switch during task execution (e.g., priority ceiling protocol)

A priority policy that guarantees unique priorities during execution does not increase the test effort, since for each given set of tasks we will have exactly one distinct execution order. However, if the priorities are non-unique, the number of potential execution orders for a set of k tasks equals $k!$ given a non-preemptive scheduling policy. The proof is trivial, consider the case when all k tasks are assigned the same priority. In this worst case, the execution order is totally arbitrary which means that all permutations of k are possible. In a more average case, we might have a distribution of the tasks over levels of priorities. Suppose we have three levels of priorities, then for each set of tasks we will have $x!*y!*z!$ distinct execution orders, where x, y, and z are the number of tasks in the set with priority level 1, 2, and 3 respectively. The question of whether priorities are guaranteed to be unique is of significant importance for the system testability. It separates the systems into two different degrees of testability where the number of test cases either grows

with combinations of the number of tasks or permutations of the number of tasks.

A priority policy that allows the priority order between tasks to be changed arbitrarily during the execution has a negative influence on testability. The change can be regarded as an event that changes the execution order for the set of current tasks. If these changes are arbitrary, all execution orders are possible for a given set of tasks. Hence, this is also a property that separates systems into two different levels of testability.

Similar results have been shown for several other execution environment properties. The conclusion is that there is an obvious impact on testability from the execution environment. Further investigations are necessary to identify other properties that are important for testability. It is also of central importance to determine the magnitude of influence from the different properties on testability and any interrelationships between them that might affect their influence. Of course, some combinations of properties may not be meaningful.

The completion of the initiated research into properties that affect testability will lead to a taxonomy of testability. The benefit of such a taxonomy is that it enables designers of event-triggered systems to regard testability when making informed trade-off decisions.

References

Birgisson, R., Mellin, J. and Andler, S. (1999). Bounds on Test Effort for Event-Triggered Real-Time Systems, *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*.

Kopetz, H. (1991). The Design of Real-Time Systems, *Software Enginering Journal* **6**(3): 72–82.

Lindström, B. (2000). *Methods for increasing software testability*, Master's thesis, HS-IDA-MD-00-017, University of Skövde.

Schütz, W. (1993). *The Testability of Distributed Real-Time Systems*, Kluwer Academic Publishers.