Deterministic State Space Exploration for Concurrent Embedded Software

Piet Cordemans, Jeroen Boydens and Eric Steegmans

Abstract - Concurrent embedded software is prone to data races, race conditions and deadlock issues. Moreover, the effect of its non-deterministic behavior is likely underestimated; therefore test coverage will be incomplete. In our approach, concurrency is isolated from the test environment. More specifically, a colored petri net model represents the platform-independent concurrency model, which delivers execution traces for tests to cover the state space completely in a deterministic manner. A proof of concept application indicates the feasibility of this approach.

Keywords – Concurrent embedded software, testing, petri nets

I. INTRODUCTION

Nowadays, embedded software does not only display concurrent behavior when dealing with interrupts. Rather, modern embedded systems incorporate Real-Time Operating Systems or multicore hardware targets, which introduce concurrency in order to take full advantage of the CPU cycles of the system. Unfortunately, this also leads to a number of concurrency problems, which are difficult to test, because of their non-deterministic characteristics.

In this paper we present a novel approach to enhance conventional unit tests, effectively iterating them through a number of concurrency problems, which are difficult to reproduce certain concurrency-related bugs a specific conventional non-deterministic schedulers. Moreover, to introduce concurrency in order to take full advantage of the computational power, state space exploration is effectively limited by the embedded target hardware.

This paper is organized in six sections. In the introduction, the problem is stated and the concept of state space exploration is introduced. Next, the premises, principles and models of the methodology are described. Then an application for a generic Actor system is provided. Finally, related and future work is indicated.

A. Problem statement

Inherent to its nature, concurrent embedded software is subject to data races, race conditions and other issues, such as live- or deadlock. This is a consequence of using conventional non-deterministic schedulers. Moreover, to reproduce certain concurrency-related bugs a specific ordering of events is required, which has a very low probability of occurring. Nevertheless, the effects of a bug in embedded software can still have disastrous effect on the system.

In this respect, conventional unit tests are regarded as an insignificant attempt to detect concurrency issues, since only a single set of thread interleavings is tested. Furthermore, due to its non-deterministic execution, should a bug manifest during a test run, there is no opportunity to reproduce the events which led to the failing assertion.

Alternatively, static analysis of concurrent software is proposed as a valid strategy in order to detect concurrency issues. This analysis is based on a lockset [1], happens-before relationship [4], or hybrid model [10]. Yet, these models have limitations as they suffer from excessive false positive results or typically are platform-specific and expensive to implement. Consequently for embedded software, this limitation is aggravated, considering the differences between embedded targets.

B. State space exploration

The state space can be modeled as a directed graph [6], in which each node represents a reachable state and each arc corresponds to a transition condition. However, the state space of a concurrent system expands exponentially when the number of atomic operations increases. This phenomenon is called state space explosion and is the main limitation when dealing with state spaces of large and complex systems. In order to deal with state space explosion, only a subset of reachable states can be considered.

Exploring the state space or a subset of the state space refers to executing software following a path of arcs, which connect the nodes. Similar to sequential test coverage, path coverage of the state space can be expressed as the number of paths covered in relation to the maximum number of coverable paths. However, ensuring the properties of state spaces, three criteria can be defined. The least comprehensive is state coverage, in which all states are visited. A broader criterion is arc coverage, since the number of arcs is at least the number of states. The all-inclusive criterion is the permutation of possible paths covering all arcs. Similar to state space explosion, the computational load of exploration rises exponentially for the more comprehensive criteria. Considering the required computational power, state space exploration is effectively limited by the embedded target hardware.

II. METHODOLOGY

In our framework, as depicted in figure 1, conventional automated unit tests are employed to explore the state space according to a number of traces, which are derived from a Colored Petri Net (CPN) model [6]. Modeling the CPN from source and test code is the first step in a static analysis phase. From the CPN model it is possible to conduct a test-based state space exploration.
In this section, the premises on which the methodology is based are stated first. Then, the principle is discussed. Next, a CPN model for tests and Actor-based implementations are described. Furthermore, traces and test coverage are discussed. Finally, a system to test deterministically is proposed.

A. Premises

The purpose of the test suite in the framework is to detect race conditions, based on the assertions of a conventional automated test suite. This test-based state space exploration approach is based on two premises.

Initially, a test suite needs to be provided with assertions, which cover the sequential behavior of the code. Furthermore, all these tests should pass to begin with. Moreover, test assertions must be able to fail when the specified behavior is not respected.

Secondly, in order to reduce the state space, the code under test should not deliberately share data in an unguarded manner across concurrent threads. Ideally, state is shared through immutable messages, as in message passing or the Actor model.

B. Separation of concerns

As concurrency is modeled in a CPN, it allows constructing the state space and discovering a relevant set of paths during static analysis. These paths can be used as input in a testing framework, which deterministically issues test runs according to the traces in the state space. This principle isolates non-deterministic behavior during test runs, while guaranteeing state space coverage.

More specifically, the static analysis constructs a model of concurrency, which lacks the common problems of this type of tools. On the one hand, it is platform-independent, since it does not require symbolic execution. On the other hand, the static analysis does not provide any information on the correctness of execution, thus eliminating the possibility of introducing false positive results.

Furthermore, conventional tests run in a deterministic fashion, as they iterate over a selection of traces of all possible paths in the state space. This increases coverage, even when dealing with traces with a low probability of occurrence. Moreover, certain coverage properties can be enforced, which is not possible with heuristic techniques.

Although tests cannot prove the absence of bugs, non-determinism reduced their effectiveness while dealing with concurrency. However, by introducing deterministic test traces, the probability of finding race conditions is only limited by assertion coverage, as in sequential tests, and the state space path coverage criterion.

By separating the concurrency model from execution, the non-deterministic behavior is isolated, while testing retains its deterministic property.

C. Colored Petri Nets

In order to model the concurrent behavior of both tests and system under test, a suitable modeling language has to be chosen. Petri Nets [5] provide a formal semantics to model concurrency, which can be visually represented. Furthermore, algorithms have already been defined to construct the state space from a Petri net.

Principally, a Petri Net consists of tokens, which traverse across a set of places, according to transitions, which in turn connects the places reciprocally. Colored Petri Nets introduce a type system to regular Petri Nets, allowing behavioral modeling of the system in a more concise manner. As it has been proven that a CPN can always be transformed in a more convoluted Petri Net, CPNs retain the desired characteristics to formally express concurrency and analyze these properties.

CPNs provide the input to construct the state space of the test. However, in order to reduce the problem of state space explosion, the CPN has to be constructed with a minimal set of reachable places, transition bindings and token values. Furthermore, abstracting the token values with classes, leads to a large state space, as the type system is in essence the representation of a more convoluted Petri Net. However, all input values of a test are known. This information can be provided to the tokens, effectively reducing the state space considerably. Moreover, this observation is generally applicable to the complete model. Unlike system models, which typically lead to large state spaces, test-specific CPN models represent a manageable subset of these state spaces, averting the state space explosion problem.

D. Trace generation

Concluding the static analysis phase, paths are selected to explore the state space. These traces are chosen based on the state space coverage criteria, in order to guarantee test coverage regardless of non-deterministic scheduling.

Algorithms can be defined to construct the traces which fulfill one of the three criteria, i.e. state, arc and path coverage. However, only the path coverage criterion is completely deterministic. Namely, considering arc and state coverage the effective traces to cover all arcs and states respectively might differ each time the analysis is run. Nevertheless, test execution remains deterministic as the state space and respective traces are fixed during test execution.

Arc and path coverage are relevant to consider, since race conditions might manifest in the sequential ordering of
states. Specifically, the states only represent the concurrent behavior, as stated in the separation of concerns. Therefore operations changing the effective memory might still interfere, due to their persistent property across states in the state space. Consequently, race conditions can only be excluded with a test when the traces cover all paths in the state space and the test assertions cover all affected variables.

E. Deterministic testing

Finally, the static analysis delivers traces as input to the deterministic testing framework, as shown in figure 2.

![Figure 2: Static analysis schedules N tests while providing the necessary traces to a marshaller.](image)

In order to make this system generic, two components need to be influenced in a non-intrusive manner. On the one hand, the test runner must execute a single test N times, as each trace represents a particular test run. On the other hand, the scheduler must schedule its tasks according to the specifications of the current test trace.

A marshaller is introduced to intercept all messages and resend them in the order, in compliance with the trace input. Furthermore to guarantee the correct trace execution, the marshaller requests the state from the code under test. This concept requires only that all messages sent are rerouted to the marshaller, while state retrieval is based on the absence or presence of the message in its respective queue.

Fundamental to this approach is, except for the static analysis, that all components are executed on the embedded target. This guarantees that the results will not contain any false positives.

III. Application

A proof of concept application is implemented to illustrate the approach of deterministically running tests, according to a trace in the state space. In order to avoid interference of undesired data races, the application is built according to the Actor model. An Actor is a reactive concurrent object, which acts in response to messages to execute its behavior. If messages are immutable, then an Actor-based system can be considered data race free, since state is not shared otherwise.

In the application an Actor framework is used, which provides a standard implementation of the Actor model. However, the framework differs on the implementation of the mailboxes, which have FIFO queuing behavior. Figure 3 is the partial CPN of a test based on the transfer of an amount of money between two accounts represented by Actors.

![Figure 3: CPN example of a test and Actor.](image)

In this model, messages are modeled as tokens traversing through Actors and their respective mailboxes. Furthermore, Actor and mailbox state are modeled to guard the critical section of the Actors, so they are unable to process multiple messages simultaneously.

Figure 4 indicates the state space, which can be constructed from the CPN model and it annotates the minimal set of traces, according to the state coverage criterion.

![Figure 4: State space of the CPN example, with the minimal set of traces for state coverage.](image)

Algorithms to construct the state space and discover the set of traces fulfilling one of the coverage criteria have their foundation in directed graph theory. This concludes the static analysis phase and delivers the traces used as input for the marshaller.

As a proof of concept, the marshaller is implemented, as an Actor in order to plug into the Actor message passing system. This approach, as shown in figure 5, is non-intrusive, as it does not require an adaption to either the scheduler or test framework. All messages sent need to be rerouted to the marshaller. In turn the marshaller will queue the message in a mailbox, which shadows the destination mailbox of the actor.

Internally the shadow mailboxes are used as a buffer to prevent destination Actors to non-deterministically read from their mailbox. Rather the current state of the test is indicated in a state matrix, while the current trace selects the next state from a reference state matrix, which was built from the state space. With this information an internal scheduler can send the required message to its effective destination. Required feedback is obtained from the actors by requesting their internal state. This operation is a safeguard as the non-deterministic scheduler is not required.
to immediately schedule the actor with a message in its mailbox.

IV. RELATED WORK

A similar tool based on state space exploration is CHESS [8] for .NET. Typically the number of thread interleavings is limited in order to deal with the state space explosion problem. Another set of tools is based on graphs, namely context-aware communication graphs. An example of this tools is Bugaboo [7]. Heuristic methods use patterns, as in CorTests [3], or random testing techniques such as the adaptive fuzzing tool QuickCheck [1]. Model-based techniques can be categorized in tools based on a specific model. The most primitive model, lockset, checks whether all shared data is properly locked. Tools are for instance Eraser [9] or Goldilocks [2]. More advanced tools are based on a happens-before relationship with examples such as Velodrome [4]. Finally, a hybrid approach combines both models, for instance in Racetrack [10].

V. FUTURE WORK

The future work for this approach consists of three items. Most importantly is the automation of the approach, in which the CPN model should be extracted automatically from source code. As long as the model is manually crafted, the technique is limited in applicability for larger test cases and quite cost intensive.

Another item of interest would be to apply CPN modeling on other models of concurrency than the Actor model.

Finally, the implementation of the static analysis algorithms to generate the state space and discover the traces could be optimized in order to reduce the number of test runs and find race conditions earlier on.

VI. CONCLUSION

Test-specific state space exploration allows deterministically running a conventional test suite in order to detect race conditions. This approach is based on CPN modeling techniques to separate the non-deterministic concurrency model from the effective test execution. However state space coverage remains guaranteed, as a set of traces is constructed, according to the required state space coverage criterion. Moreover the test-specific model reduces the state space considerably, thus avoiding the state space explosion problem. After the static analysis, a marshall component wraps the non-deterministic scheduler and ensures that the test traces are executed deterministically.

In this paper we described the implementation of this approach in an Actor-based software system.

REFERENCES