Symbolic System Time in Distributed Systems Testing

Oscar Soria Dustmann, Raimondas Sasnauskas, and Klaus Wehrle
Communication and Distributed Systems (ComSys)
RWTH Aachen University, Germany
firstname.lastname@comsys.rwth-aachen.de

***Abstract*—We propose an extension of symbolic execution of distributed systems to test software parts related to timing. Currently, the execution model is limited to symbolic input for individual nodes, not capturing the important class of timing errors resulting from varying network conditions.**

In this paper, we introduce symbolic system time in order to systematically find timing-related bugs in distributed systems. Instead of executing time events at a concrete time, we execute them at a set of times and analyse possible event interleavings on demand. We detail on the resulting problem space, discuss possible algorithmic optimisations, and highlight our future research directions.

I. Introduction

Symbolic execution [1] has been established as a viable idea to test off-line systems [2], [3]. Such test setups can typically be divided into two phases: First, specify some symbolic input to feed into the system. Second, run the system on that symbolic input. Consequently, testing the system's behaviour for all possible inputs results in high code coverage and thus in a high probability to detect errors.

SDE (Symbolic Distributed Execution) applies symbolic execution to distributed systems [4], [5], such as WSNs (Wireless Sensor Networks). Its representation of interaction in networks by means of discrete events is a helpful idiom for testing as it enables state space reduction. However, the execution of interactive systems is not solely dependent on the actual data that is provided as input but also on the time required for inter-node communication. Thus, we propose a generalised symbolic input-model for distributed discrete event systems consisting of the following categories: (1) *External stimuli* and (2) *synchronisation latency*.

While external stimuli relate to data that is of potential interest, synchronisation latency is a result of the underlying medium's performance. Existing SDE-capable tools are limited to modelling symbolic input of the first category. Although this includes network failure semantics to some degree (e. g., introducing symbolic packet drops), it precludes the ability to test code that is sensitive to varying network conditions. Since this is typically low level network interface code, bugs in that portion of the system might be especially costly in e. g., a WSN, if single nodes in the deployed system stop responding.

II. Symbolic Network Conditions

We propose research into the realisation of the input model presented in the previous section. To be able to test systems under symbolic network conditions, we introduce an element of uncertainty for the delivery of data packets. This is effectively impossible with current SDE implementations; Receiving a packet is realised on discrete event systems by scheduling an event in the future, depending on the packet delivery time. The underlying event scheduling mechanism will dispatch the associated handler once the system reaches the respective event time. If the event time is uncertain, the event appears in a status where it was already executed and where it is not yet. A naïve resolution to this apparent contradiction is to follow both possibilities for every discrete time step, time steps being subject to the granularity of the system time. For instance, a packet that is assumed to have a delay of up to $100\,\mathrm{ms}$, would cause a million execution states to be forked off on a node with a time granularity that corresponds to a $10\,\mathrm{MHz}$ clock.

A. Problem Subdivision

While this approach would certainly produce correct results and reside completely in userspace model-code, it is infeasible for intensively communicating networks. We intend to defer scheduling decisions to the symbolic execution engine and introduce the notion of symbolic event time. This means that an event is no longer tied to a concrete time, but instead a set of times. We see two core challenges in building such a system:

1) Execute event handlers of the tested code on a set of virtual times instead of a concrete virtual time.
2) Find a strict total ordering of events with sets of virtual times.

1) Event Execution: Execution of the event handler for a given event must be done in a fashion that allows us to pretend it was executed on any concrete time in the set of admissible event times. This will cause the regular symbolic execution of the subsequent code to branch into separate execution paths, if and only if necessary.

To implement this approach one must identify the location of the system time, in order to modify it. However, this approach will result in branches that create truly inequivalent, and therefore non-redundant, execution states.

2) Event Ordering: Discrete Event Scheduling is a well studied and understood procedure: For a distributed system, executed (or simulated) by a single host, the scheduler simply executes the event with the smallest time stamp, while considering events of the whole network. As trivial as this is for

classical DES (Discrete Event Simulators) the application in our domain is problematic as the generalisation of event times to sets of event times causes us to loose the canonic ordering of the natural numbers as strict total ordering of events.

Consider two events e_1 and e_2 with the respective non-trivial time sets T_1 and T_2. If there are *no* times $t_i \in T_i,\ t_j \in T_j$ with $t_j \leq t_i$ for a given valuation of $\{i, j\} = \{1, 2\}$, we can safely execute e_i before e_j without violating the causality constraints imposed by the tested application (i. e., the future must not precede the past). Otherwise, in the presence of such times t_i, t_j for both possible valuations of (i, j), we are not allowed to execute e_1 before e_2 nor are we allowed to execute e_2 before e_1 because both would violate causality constraints.

However, in a real system run, one of these events would be executed first, in another system run, the other might be executed first. This means that our current execution state is too general to reflect both paths the execution could take. Therefore, we branch the execution. For one path a we choose e_1 to be executed first, with all timestamps that are admissible, according to T_1, while we restrict the allowed timestamps of e_2 to values greater than or equal to the values of T_1. For the other path, b, we proceed analogously to a, only with e_1 and e_2 exchanged.

This kind of forced event sorting is known as *Interval Branching* and was studied for system-simulation [6]. Although the basic problems of Interval Branching and Symbolic Time are quite similar, their respective domains allow for different solutions and optimisations. While the former is a Parallel Discrete Event Simulator, running a system model on several LPs (Logical Processes), the latter relies on a symbolic execution engine of actual, unmodified code, which is run in a single process.

B. Possible Optimisations

While Interval Branching can be expected to produce reasonable state spaces, there might be additional considerable improvements. SDE reduces the number of execution states by keeping transitive relationship information and branching packet receivers only if there exist competing, non-sending states. This global view allows packet-interception and inspection, which results in additional state space reduction if the same packet is sent by all relevant states. With symbolic packet arrival times, packet equivalency depends on the intersection of arrival times, which may be overlapping, yet unequal. There are several valid possibilities to optimise this situation and it will be subject to evaluation to decide which is most efficient.

Since events on separate nodes are independent, per se, interleaving times require no state branching in such a case. However, in this situation one event e_1 can be chosen to be run prior to the other event e_2. But event e_2 might schedule an event e_3 on e_1's node. Since the execution time intervals of e_1 and e_2 overlap, e_3 is allowed to be scheduled on a time interval that overlaps with e_1. Therefore, e_3 would have been eligible to be run prior to e_1, due to the Interval Branching mechanism. In order to resolve this, a state must be preserved, where e_1 has not been executed. This would then allow an execution where e_3 is scheduled prior to e_1. While snapshotting can be used to implement this rollback, the execution of the other equally legit branch where e_1 indeed does precede e_3 has been run incorrectly because the time restrictions imposed by event e_3 were unknown when e_1 was run. Thus, e_1 would have to be rerun with adjusted time constraints on the snapshot. It depends on the execution framework, whether this work repetition can be avoided by retroactively narrowing the set of times at which e_1 has been run. Such retroactive tampering would constitute an optimistic execution scheme that entails no rollback penalty.

III. Outlook

We see this research direction not only as an effort to amend symbolic packet delivery time capabilities to SDE but as a paradigm shift and a different approach to symbolic testing of distributed systems. The limited input model of current SDE forces the user to invest time into identifying crucial sensory input and hoping that a few symbolically injected packet drops will trigger a possible bug. Our proposed input model is more generic, in the sense that it allows for push-button tools for testing of protocol behaviour.

However, a generalised input model yields a more thorough coverage at the price of a higher workload for the execution engine and therefore a longer runtime and possibly a larger memory footprint. For instance, an overzealous attempt to test all possible cases, might result in the choice of large packet delivery intervals, producing a high number of overlapping events and therefore severe state explosion. This presents us with an additional source of unusability.

We believe that a hybrid system that infers rough latency estimates from an actually deployed system or a concrete emulation, and weakens these to add symbolic uncertainty could be used to efficiently run fully automated tests of networked systems. Further applications could entail incremental searcher implementations for continuous testing, discovering even remote corner case bugs.

References

[1] J. C. King, "Symbolic execution and program testing," in *Commun. ACM*, vol. 19. ACM, July 1976, pp. 385–394.

[2] D. Dunbar, C. Cadar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008, pp. 209–224.

[3] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. ACM, 2008.

[4] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ser. IPSN '10. ACM, 2010, pp. 186–196.

[5] R. Sasnauskas, O. Soria Dustmann, B. L. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski, "Scalable symbolic execution of distributed systems," in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ser. ICDCS '11. IEEE, 2011, pp. 333–342.

[6] P. Peschlow, P. Martini, and J. Liu, "Interval branching," in *22nd Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '08. IEEE, 2008, pp. 99–108.