Erschienen in: 2014 International SPIN Symposium on Model Checking of Software (SPIN) : Proceedings, July 21-23, 2014, San Jose, CA, USA / Neha Rungta ... (Hrsg.). - New York, NY : ACM, 2014. - S. 117-120. - ISBN 978-1-4503-2452-6

SpinCause: A Tool for Causality Checking

Florian Leitner-Fischer
University of Konstanz, Germany
Konstanz, Germany
florian.leitner@uni-konstanz.de

Stefan Leue
University of Konstanz, Germany
Konstanz, Germany
stefan.leue@uni-konstanz.de

ABSTRACT

In this paper we present the SpinCause tool for causality checking of Promela and PRISM models. We give an overview of the capabilities of SpinCause and briefly sketch how the causality checking algorithms are integrated into the state-space exploration algorithms used for model checking. In addition we compare the runtime and memory needed for causality checking with the different state-space exploration algorithms and two newly proposed iterative causality checking approaches.

Categories and Subject Descriptors

D.2.4 [**Software /Program Verification**]: Model Checking; D.4.5 [**Reliability**]: Verification

General Terms

Verification

Keywords

Model Checking, Causality, Fault Localization

1. INTRODUCTION

Model checking [6] is an automated technique to check whether a model of a system violates a formalized property. In case the property is violated, the model checker returns a counterexample, which consists of a system execution trace leading to the property violation. While a counterexample helps in retracing the system execution leading to the property violation, it does not identify causes of the property violation and represents merely one possible execution of the system.

In recent work [16] we have proposed a method, complementing model checking, called *causality checking* which aims at providing insights into why a property was violated during model checking. Causality checking uses an adaption of the *actual cause* definition by Halpern and Pearl [11] in order to algorithmically computed the causal events for a reachability property violation. We adapt the actual cause definition from [11] so that it can be applied to model checking. Since the systems that we aim to analyze are concurrent systems, the order in which the events occur also needs to be considered as a causal factor for the property violation, since for instance, one execution trace might entail a race condition and lead to a property violation while another execution trace consisting of the same events might not violate the property. We extend the adapted actual cause definition in order to take the ordering of events into account as a causal factor. The causal event orderings are captured by the event order logic that we have proposed in [16].

Causality checking computes the event combinations causing a property violation, together with the order in which the events have to occur in order to be causal.

In this paper we present the SpinCause[1] tool for causality checking of Promela [12] and PRISM [14] models. SpinCause is based on the SpinJa [7] tool-set a Java reimplementation of the Spin [12] model checker. We give an overview of the capabilities of SpinCause and briefly sketch how the causality checking algorithms are integrated into the state-space exploration algorithms of SpinJa. We then compare the runtime and memory needed for causality checking using different state-space exploration algorithms.

2. THE SPINCAUSE TOOL

SpinCause comprises the following features:

- Causality checking of Promela models as proposed in [16].
- Causality checking of PRISM models via the PRISM to Promela translation algorithm proposed in [17].
- Computation of the probabilities of causal event combinations in PRISM models as proposed in [17].
- Representation of causal event orderings with the event order logic (EOL) that we have proposed in [16].
- Visualization of the computed causal relationships with fault trees [18], which are a method used in engineering practice to visualize the causal relationship between events and requirement violations.

The causality checking approach proposed in [16] can be integrated into both of the standard state-space exploration algorithms used in explicit state model checking, namely depth-first search (DFS) and breadth-first search (BFS). Whenever a bad or a good execution is found by the search algorithm it is added to a special data-structure that is used to compute the causality relationships. In order to efficiently

[1] http://se.uni-konstanz.de/research1/tools/spincause

store the execution traces, we use two prefix tree [8] data-structures, one prefix tree that stores the actions representing the events of the execution traces and one prefix tree that stores the states of the execution traces. In order to make causality decisions on-the-fly we have devised a data-structure called subset graph. We can decide whether an execution trace is causal as soon as we add it to the subset graph.

If DFS or BFS encounter a state that is already in the state-space and hence all successors of this duplicate state have already been explored, the successors are not explored for a second time. When DFS encounters a duplicate state, it is possible that the new trace to the duplicate state is shorter or has a different event order than the already known execution traces leading to the duplicate state. Hence the new execution trace is needed to ensure the minimality of the causal event combinations and to be able to detect all orderings. BFS explores the state-space following an exploration order that leads to a monotonically increasing length of the execution traces, consequently, the new execution trace found by BFS leading to the duplicate state either has the same length as the already known execution trace leading to the duplicate state, or the new execution trace is longer than the already known execution trace. If the new execution trace has the same length, the events on the trace have an order that is different from the one in the already known execution trace. Hence the new execution trace is needed to be able to detect all orderings. We have implemented a method called prefix matching that ensures that, all execution traces are found, by replacing an old prefix leading to a duplicate state with the new prefix and adding the resulting trace to the sub-set graph. For an in-depth discussion of the integration of causality checking into DFS and BFS we refer to [16].

In addition to the integration into DFS and BFS we propose in Section 2.1 and Section 2.2 two iterative approaches that reduce the runtime and memory consumption needed for causality checking and have not been previously published.

2.1 Iterative Approach

In the standard causality checking approach the bad and good traces found during state-space exploration are added to the data-structure used for causality checking and stored if necessary. Especially due to the storage of the potentially large number of good traces, this is not memory efficient. We will now propose an iterative version of the approach that consumes less memory. The iterative algorithm constitutes of two consecutive executed state-space explorations with BFS. In the first state-space exploration, we limit the causality checking to identify the minimal causal event combinations for a property violation and in the second state-space exploration, we focus on finding the causal event orderings for the previously identified causal event combinations and check whether the non-occurrence of some event is causal. The iterative approach leverages the fact that the length of the bad traces found by BFS is monotonically increasing, hence the iterative approach is not implemented for DFS, because this would be very memory inefficient.

2.2 Iterative Approach with Parallel BFS

In order to further optimize the runtime of the iterative approach we extended the parallel breadth-first algorithm already implemented in the SpinJa model checker to support causality checking. The parallelization of the BFS algorithm is achieved by executing a predefined number of BFS threads with a shared queue, state-space, and sub-set graph for causality checking. Consequently each parallel BFS thread retrieves a state from the shared queue and adds the successor states to the shared queue and checks whether the property is violated in one of the successor states.

3. EXPERIMENTAL EVALUATION

We evaluate the causality checking approach using four case studies from academia and industry. We compare the runtime and memory consumption of the standard causality checking approach, the iterative approach, and the iterative approach with parallel breadth-first search and summarize our results. For all experiments partial-order reduction was disabled and we computed all possible counterexamples with the -c0 option of SpinJa. The following experiments were performed on a PC with two Intel Xeon Processor (3.60 Ghz; 4 cores) and 144 GBs of RAM. Due to space restrictions we present here only the results of the causality checking for the airbag system.

Airbag System [1]. The architecture of the airbag system consists of two acceleration sensors, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag. We are interested in computing the causal events for an inadvertent ignition of the airbag. The Promela model of the airbag system consists of 2,952 states and 25,340 transitions. While there are a total of 912 bad states, the causality checker result comprises only 5 causal event combinations. Obviously, a manual analysis of this large number of traces in order to determine causal factors would be very laborious. Figure 1 shows the fault tree generated by the SpinCause tool. The event order logic (EOL) formula returned by the causality checker is Ψ = (FASICShortage)∨(FETStuckHigh ⟑ ASICStuckHigh)∨ (MicroControllerFailure ⟑ enableFET ⟑ armFASIC ⟑ fireFASIC)∨(FETStuckHigh ⟑ MicroControllerFailure ⟑ armFASIC ⟑ fireFASIC) ∨ (MicroControllerFailure ⟑ enableFET ⟑ FASICStuckHigh). There is only one single fault that can lead to an inadvertent deployment, namely *FASICShortage*, which is represented by the EOL formula FASICShortage. The combination of the basic events *FETStuckHigh* and *FASICStuckHigh* only leads to an inadvertent deployment of the airbag if the basic event *FETStuckHigh* occurs prior to the basic event *FASICStuckHigh*, which is represented by the EOL formula FETStuckHigh ⟑ FASICStuckHigh. The basic event *MicroControllerFailure* can lead to an inadvertent deployment if it is followed by the following sequence of basic events: *enableFET*, *armFASIC*, and *fireFASIC*. This sequence is represented by the EOL formula MicroControllerFailure ⟑ enableFET ⟑ armFASIC ⟑ fireFASIC. If the basic event *FETStuckHigh* occurs prior to the *MicroControllerFailure* the sequence in which *armFASIC* and *fireFASIC* occur after the *MicroControllerFailure* event suffices to lead to the top level event. This sequence is represented by the EOL formula FETStuckHigh ⟑ MicroControllerFailure ⟑ armFASIC ⟑ fireFASIC. If the basic event *FASICStuckHigh* occurs after *MicroControllerFailure* and *enableFET* this also leads to a sequence leading to an inadvertent deployment. It is represented by the EOL formula MicroControllerFailure ⟑ enableFET ⟑ FASICStuckHigh .

Embedded Control System [13]. Various failure modes can

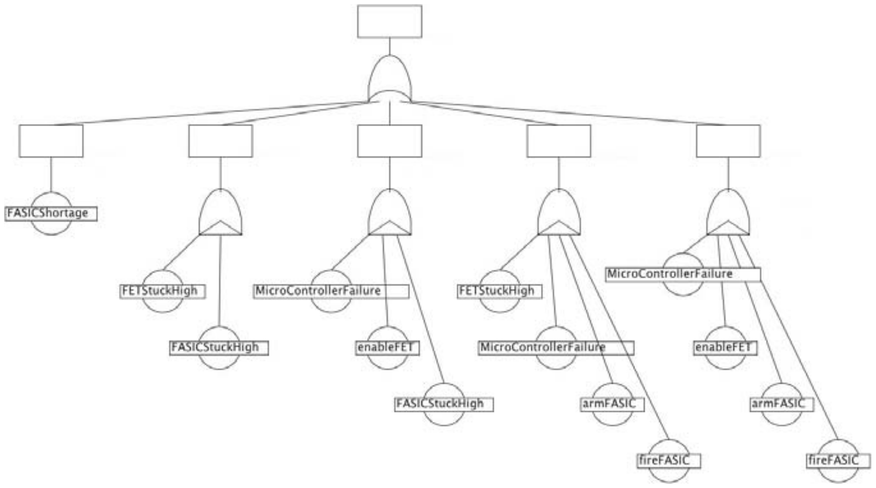

Figure 1: Fault tree of the airbag system

lead to a shutdown of the system. We are interested in computing the causal events for the event "system shut down". We set the constant *MAX_COUNT*, which represents the maximum number of processing failures that are tolerated by the main processor, to a value of 5. SpinCause automatically translates the PRISM model to a Promela model which comprises 6,013 states and 25,340 transitions and contains a total of 83 bad states. Note that for comparability reasons we only computed the causal events without a probability computation.

Train Odometer Controller [4]. The Promela model of the train odometer comprises 11,722 states, 14,049 transition, and 1368 bad states.

Airport surveillance radar (ASR) [2]. The ASR system was modeled in the SysML from which a Promela model was automatically derived by the QuantUM tool[15]. We analyze two variants of the ASR system, one with one data-processing channel and one with two redundant data-processing channels. The one channel variant comprises 1,230,516 states and 7,492,866 transitions and the redundant two channel variant comprises 46,389,412 states and 326,412,170 transitions. The one channel variant contains 1,018,624 bad states and the two channel variant contains 15,206,400 bad states.

Discussion. We will now compare and discuss the runtime and memory consumption of the proposed causality checking approach. The runtime and memory needed for model checking of the case studies with DFS and BFS and the runtime and memory needed for causality checking including model checking with DFS and BFS and the iterative causality checking approaches are given in Table 1. The runtime and memory consumption for causality checking with DFS of both variants of the ASR case study and with BFS of the ASR case study with two channels can not be given because the algorithm ran out of memory (oom.) prior to the completion of the causality checking. The following trends can be identified:

- If no causality checking is done, DFS and BFS have approximately the same runtime and memory consumption. The causality checking adds a runtime and memory penalty, but the experiments show that causality checking is applicable to industrial size Promela models. In addition causality checking provides valuable insight as to why the hazard occurs, which is very tedious or even impossible to determine if standard model checking and manual counterexample analysis is used, due to the amount of counterexamples generated.
- In the first implementation published in [16] we did not use prefix trees in order to store the states and actions. For the airbag model, for instance, the old implementation consumed 3.5 GBs of memory for causality checking using BFS which is reduced to 18.51 MBs by the iterative approach with parallel BFS using prefix trees.
- When performing causality checking, BFS outperforms DFS for large models in terms of both runtime and memory consumption. BFS outperforms DFS because if BFS is used, we can safely rely on the assumption that when a bad trace is found all shorter bad traces already have been found. This assumption assures that the minimality constraint imposed by the causality conditions defined in [16] holds as soon as a causal event combination is found. If DFS is used, no assumptions on the length of the bad trace can be made and thus a much larger number of traces needs to be stored.
- For very small models, like the embedded model, both the runtime and the memory consumption for the iterative approaches are higher than for the standard approach. This is due to the fact that the state-space is explored twice. With increasing size of the model, the iterative approaches outperform the standard approach with respect to runtime and memory consumption.
- With increasing size of the model the iterative approach with parallel breadth-first search outperforms the iterative approach with the standard BFS in terms of runtime and memory. Note that even though for the parallel BFS approach the different BFS threads need to be managed, there is no overhead introduced by the parallelization of BFS. With the iterative approaches it is possible to analyze the 2 channel variant of the ASR case study which was not possible with the standard causality checking approaches because they ran out of memory.

Table 1: Runtime and memory needed for model checking of the case studies with DFS and BFS and for causality checking with DFS and BFS and the iterative causality checking approaches.

	Model Checking				Causality Checking							
	DFS		BFS		DFS		BFS		Iterative Approach with standard BFS		Iterative Approach with parallel BFS	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
Airbag	0.17	9.23	0.18	9.06	0.86	165.52	1.24	21.19	1.55	18.53	1.59	18.51
Embedded	0.05	8.94	0.06	8.76	0.13	19.95	0.16	9.43	0.75	17.99	0.75	17.99
Train Odo.	0.26	9.79	0.27	9.62	15.06	2280.86	2.59	63.36	1.59	19.21	1.44	19.11
ASR 1 Chan.	44.95	8,467	51,66	8,466	oom.	oom.	750,31	24,663	65.31	16,661	34.04	325.14
ASR 2 Chan.	503.92	14,669	659.26	14,668	oom.	oom.	oom.	oom.	1,342	27,759	1,328	12,415

4. RELATED WORK

Work documented in [3] uses the Halpern and Pearl approach to explain counterexamples in CTL model checking by determining causality. However, this approach considers only variable-value changes on single counterexamples. In [9] a formal framework for reasoning about global contract violations is presented. This approach decides whether a prefix of a local trace is the cause for a global property violation or not, thus focusing on individual traces instead of a set of traces. Work by Groce et al. described in [10] establishes causality based on counterfactual reasoning by computing distance metrics between execution traces. All of the above mentioned approaches work only on single counterexamples. To the best of our knowledge we are not aware of any other causality checking algorithm identifying all causal events, that can be integrated with explicit state-space exploration algorithms, and which works on-the-fly. Chockler et al. define in [5] a coverage measure for model checking based on the notion of causality, where as in our approach we aim at computing the causal events for a property violation.

5. CONCLUSION

The SpinCause tool implements the causality checking approaches proposed in [16, 17] and offers causality analysis for Promela and PRISM models. We have demonstrated that causality checking can be applied to industrial sized models and have shown that the iterative causality checking approaches lead to a reduction of the consumed runtime and memory.

6. REFERENCES

[1] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In *Proc. of QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society, 2009.

[2] A. Beer, U. Kühne, F. Leitner-Fischer, S. Leue, and R. Prem. Analysis of an Airport Surveillance Radar using the QuantUM approach. Technical Report soft-12-01, Chair for Software Engineering, University of Konstanz, 2012.

[3] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.

[4] E. Böde, T. Peikenkamp, J. Rakow, and S. Wischmeyer. Model Based Importance Analysis for Minimal Cut Sets. In *Proc. of ATVA 2008*, volume 5311 of *LNCS*. Springer, 2008.

[5] H. Chockler, J. Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? *ACM Transactions on Computational Logic*, 9(3), 2008.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking (3rd ed.)*. The MIT Press, 2001.

[7] M. de Jonge and T. Ruys. The spinja model checker. In *Model Checking Software*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer, 2010.

[8] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[9] G. Gössler, D. L. Métayer, and J.-B. Raclet. Causality analysis in contract violation. In *Runtime Verification*, volume 6418 of *LNCS*, pages 270–284. Springer, 2010.

[10] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3), 2006.

[11] J. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 2005.

[12] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addision–Wesley, 2003.

[13] M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427–1434, 2006.

[14] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[15] F. Leitner-Fischer and S. Leue. QuantUM: Quantitative safety analysis of UML models. In *Proceedings Ninth Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, volume 57 of *EPTCS*, pages 16–30, 2011.

[16] F. Leitner-Fischer and S. Leue. Causality checking for complex system models. In *Proc. 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI2013)*, LNCS. Springer, 2013.

[17] F. Leitner-Fischer and S. Leue. On the synergy of probabilistic causality computation and causality checking. In *In Model Checking Software - Proceedings of International SPIN Symposium on Model Checking of Software. Stony Brook, NY, USA*, volume 7976 of *LNCS*, pages 246–263. Springer, 2013.

[18] U.S. Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.