

Towards Causality Checking for Complex System Models

Florian Leitner-Fischer and Stefan Leue

florian.leitner@uni-konstanz.de, stefan.leue@uni-konstanz.de

Abstract: With the increasing growth of the size and complexity of modern safety-critical systems, the demand for model based engineering methods that both help in architecting such systems and to assess their safety and correctness becomes increasingly obvious. Causality checking is an automated method for formal causality analysis of system models and system execution traces. In this paper we report on work in progress towards an on-the-fly approach for causality checking of system models. We also sketch how this approach can be applied in model-based system analysis when assessing the system's functional correctness.

1 Introduction

With the increasing growth of the size and complexity of modern safety-critical systems, the demand for model based engineering methods that both help in architecting such systems and to assess their safety and correctness becomes increasingly obvious. Due to the size of the systems traditional techniques like reviews and testing, on the one hand, and manual fault tree analysis or failure mode and effect analysis on the other hand, can only be applied to limited parts of the system. The main reason for this limitation lies in the vast amount of time and resources that is consumed by manually executing those techniques. In order to be able to completely assess the correctness and safety of these systems automated or at least computer-aided techniques are needed.

Model Checking [CGP01] is an established technique for the automated verification of system correctness. If a model of the system and a formalized property is given to the model checker, it automatically checks whether the property is fulfilled or not. In case the property is violated, the model checker returns a counterexample, which consists of a system execution leading to the property violation. While a counterexample helps in retracing the system execution leading to the property violation, it does not identify the causes of the property violation. Only if we look at a large number of counterexamples for a given property and if we are thus able to identify commonalities among them we might be able to gain insight into what combination of events may have caused the property violation. Complex system models can have very large numbers of counterexamples, each of which may be very long if the model features causes for nondeterminism, such as for instance concurrency. A manual analysis of those sets of counterexamples in order to determine fault causalities is therefore impossible, which calls for algorithmic tool support.

In [LFL11a, LFL11b] we presented a tool based approach called QuantUM that allows for

the specification of dependability characteristics and requirements directly within a system or software architecture model that is specified in the Unified Modeling Language (UML). Furthermore the annotated models are automatically translated into the input language of the probabilistic model checker PRISM [HKNP06] and subsequently analyzed. Our tool DiPro [ALFLS11] for probabilistic counterexample computation is integrated into the QuantUM tool and allows for efficient counterexample computation. In [KLFL11] we extend this approach by an automated mapping of the counterexamples to fault trees via causality computation. Furthermore, the system executions represented by the fault tree can then be represented by a UML sequence diagram that allows for result interpretation on the level of the UML model. This approach and tool chain has recently been extended to support models in the SysML language. While the QuantUM approach has been successfully applied to several academic and industrial case studies, two possible enhancements became visible: 1. For the time being the causality computation requires the enumeration of all bad executions, that is all paths in the counterexample, and all good executions, that is all paths not in the counterexample. While it was possible for all case studies so far to compute all executions prior to causality computation, we think that an on-the-fly algorithm for causality checking would be much more efficient. 2. So far, the focus of the QuantUM approach was to enable quantitative analysis of UML models. Apart from that, a pure qualitative analysis of reachability properties with a functional model checker could be of interest. Hence we aim at adding an automatic translation of QuantUM models into Promela the input language of the model checker SPIN. The expected benefit of the qualitative analysis, is that a qualitative model checker like SPIN usually has a much better scalability as probabilistic model checkers like PRISM.

Since it is beyond the scope of this paper to address the above mentioned enhancements in full detail, we will focus here on our ideas and work in progress towards an automated causality checking for system models. The automated translation of QuantUM annotated models to Promela will not be discussed in this paper.

The remainder of this paper is structured as follows. In Section 2 we discuss how causality relationships can be formally established within system models. An on-the-fly approach for causality computation that can be integrated with depth-first search and breadth-first search algorithms used for model checking is presented in Section 3. In Section 4 we sketch how this approach can be applied in model-based system analysis when assessing the system's functional correctness. Related work is discussed in Section 5. We conclude in Section 6.

2 Causality Reasoning in System Models

Our goal is to identify the events that caused a system to reach a certain undesired state, for instance such a state could represent a hazard or a potential unsafe state of the system. We use the model checker to check whether there are system executions that lead to such an undesired state. One of the most common approaches on causality reasoning is *counterfactual* reasoning and the related *alternative world* semantics of Lewis [Lew01, Col04]. The counterfactual argument is widely used as the foundation for identifying faults in program

debugging [Zel09, GCKS06]. The "naive" counterfactual causality criterion according to Lewis is as follows: event A is causal for the occurrence of event B if and only if, were A not to happen, B would not occur. The testing of this condition hinges upon the availability of alternative worlds. A causality can be inferred if there is a world in which A and B occur, whereas in an alternative world neither A nor B occurs. In our setting we can think of each possible system execution representing an alternative world.

In [KLFL11] we use an adaption of the *structural equation model (SEM)* by Halpern and Pearl [HP05] in order to automatically compute fault trees out of a complete set of counterexamples, that is the bad executions where the property is violated, and the corresponding good executions of the system where the property is not violated. The underlying SEM extends the counterfactual reasoning approach by Lewis and encompasses the notion of causes being logical combinations of events as well as a distinction of relevant and irrelevant causes. Additionally we extended the SEM by a so called event order logic that allows for considering the order of the events in causality computation and thus enables the usage of this adapted model in the context of concurrent system models.

In order to illustrate the definitions, we will demonstrate them on a small example of a railroad crossing. In this example, a train can approach the crossing (Ta) and cross the crossing (Tc) and finally leave the crossing (Tl). Whenever a train is approaching, the gate should close (Gc) and will open when the train has left the crossing (Go) but it might also be the case that the gate fails (Gf). The car approaches the crossing (Ca) and crosses the crossing (Cc) if the gate is open and finally leaves the crossing (Cl). We are interested in finding those events that lead to a state where both the car and the train are in the crossing which would lead to an accident.

We now sketch here an informal definition of our adaption of the SEM that is tailored for the use in the context of model based systems analysis. In the following we use the event order logic that is also defined in [KLFL11]. Here it suffices to say that besides other operators, we also have defined a sub-set (\sqsubset) and ordered sub-set ($\dot{\sqsubset}$) operator that express sub-set relationships of ordered event logic formula. If the sub-set operator holds for two event ordered logic formulas, the first event logic formula represents a sub execution of the second one. For instance the execution of our example " Ta, Gf " is an ordered sub execution of the execution " Ta, Ca, Gf ". An ordered causal formula in the adapted SEM is an order constraint boolean conjunction ψ of boolean variables representing the occurrence of events. The variable associated with an event is true in case that event has occurred. Intuitively you can think of ψ being an ordered boolean conjunction of the events that cause the failure. We use the \wedge operator to indicate that two events are an ordered conjunction. Note that these kind of ordered boolean conjunction can be used to represent system executions. The execution trace " Ta, Ca, Gf " consequently can be represented by the ordered boolean conjunction $\psi = Ta \wedge Ca \wedge Gf$. The set of all variables is partitioned into the set U of *exogenous* variables and the set V of *endogenous* variables. Exogenous variables represent facts that we do not consider to be causal factors for the effect that we analyze, even though we need to have a formal representation for them so as to encode the "context" ([HP05]) in which we perform causal analysis. Endogenous variables represent all events that we consider to have a meaningful, potentially causal effect. The set $X \subseteq V$ contains all events that we expect jointly to be a candidate cause, and the boolean conjunction of

these variables forms a causal formula ψ .

It should be pointed out that [EL02] contains a careful analysis of the complexity of computing causality in the SEM. Most notable is the result that even for an SEM with only binary variables computing causal relationships between variables is NP-complete. Thus we compute an over-approximation here. Instead of identifying single events that cause the effect, we compute the causal events together with the events that are part of the causal process. The causal process comprises all variables that mediate between X and the effect that ψ is causing. Those variables are not root causes, but they contribute to rippling the causal effect through the system until reaching the final effect.

Omitting a complete formalization, we assume that there is an actual world and an alternate world. In the actual world, there is a function val_1 that assigns values to variables. In the alternate world, there is a function val_2 assigning potentially different values to the variables. The valuations are given through the system executions. For all events that occur on a system execution the variables representing those events are set to true, all other variables are set to false. The system execution "Ta,Ca,Gf" for instance corresponds a valuation that sets the variables representing the events Ta, Ca and Gf to true and all other variables to false. Furthermore, we assume the existence of a function $order_1$ assigning an order to the occurrence of the events M in the actual world, as well as a function $order_2$ which assigns a potentially different order in the alternate world. The order functions are also given through the order of the events of the corresponding system execution. The example execution "Ta,Ca,Gf" corresponds to the order Ta before Ca before Gf. Note that in our setting each "world" represents an execution trace that can either be a bad execution leading to the effect or a good execution that does not lead to the effect. Hence the alternate worlds that need to be considered by the following tests are limited to execution traces that are actually possible in the system. An event order logic formula ψ is considered a cause for an event represented by the event order logic formula φ , if the following conditions are satisfied:

AC1: Both ψ and φ are true in an execution trace, assuming the context defined by the variables in U , given a valuation $val_1(V)$ and an order $order_1(V)$. In our example this condition is true for each ψ that leads to a possible crash (φ). The execution "Ta,Ca,Gf,Cc,Tc" which is represented by $\psi = Ta \wedge Ca \wedge Gf \wedge Cc \wedge Tc$ leads to a possible crash, because the car (Cc) and the train (Tc) are both in the crossing at the same time.

AC2: The set of endogenous events V is partitioned into sets Z and W , where the events in Z are involved in the causal process and the events in W are not involved in the causal process. It is assumed that $X \subseteq Z$ and that there exist valuations $val_2(X)$ and $val_2(W)$ and orders $order_2(X)$ and $order_2(W)$ such that:

1. Changing the values of the variables in X and W from val_1 to val_2 and the order of the variables in X and W from $order_1$ to $order_2$ changes φ from true to false. Note that changing the values of the variables in X and W is analog to adding or removing events from the execution trace. Changing the order means, that the order of the events on the execution trace is changed. These changes create sub- or super-executions with respect to the unordered subset operator. If we take for example the execution "Ta,Ca,Gf,Cc,Tc" which is represented by $\psi = Ta \wedge Ca \wedge Gf \wedge Cc \wedge Tc$ and

remove the event Tc the value of φ changes from true to false, because then only the car is in the crossing but the train isn't.

2. Setting the values of the variables in W from val_1 to val_2 and the order of the variables in W from $order_1$ to $order_2$ should have no effect on φ as long as the values of the variables in X are kept at the values defined by val_1 , and the order as defined by $order_1$, even if all the variables in an arbitrary subset of $Z \setminus X$ are set to their value according to val_1 and $order_1$.

This test is problematic in the context of system executions, since the variable values in Z can be affected by the change of the variables in W . For instance, when a newly introduced event in W preempts the occurrence of an event in Z the value of the variable in Z representing this event would be changed. But the condition requires to set the variable values of Z to their original values and to overwrite the new assignment. Because of this it is possible that an execution which is not possible in the system is considered. In order to prevent this, we adopt the condition to the following: Since the value of all variables in W is false, changing the values of variables in W corresponds to adding events to the execution. Hence we search for good execution traces that are super-executions, that is the current execution is a sub-set of the said good execution, of the execution we want to check. If no such execution exists the test is passed. If adding an event by changing the values of variables in W has an effect on φ , this means that the added event does prevent the effect from happening. As a consequence, we can say that the events in X are only causal if the event we have added through the change of W does not occur. Or in other words, the non occurrence of the event is also causal. Hence, we have to reflect that in the ordered conjunction of events by adding the negation of that event. If for instance we add the event car left crossing (Cl) to our example execution "Ta,Ca,Gf,Cc,Tc" between the Cc and Tc events the car and train are not in the crossing at the same time and φ changes from true to false. In order to reflect this in ψ we add the negation of the Cl event and get $\psi = Ta \wedge Ca \wedge Gf \wedge Cc \wedge \neg Cc \wedge Tc$.

AC3: The set of variables X is minimal: no subset of X satisfies conditions AC1 and AC2. Minimality ensures that only those elements of the conjunction that are essential for changing φ in AC2(1) are considered part of the cause; inessential elements are pruned. Of course, in our example there are system executions where first a number of trains and cars cross the crossing without an accident followed by a sequence of events that cause an accident. But obviously we are interested in the smallest ordered conjunctions of events possible.

If a formula ψ meets the above described conditions, the occurrence of the events included in ψ is causal for φ . However, condition AC2 does not imply that the order of the occurring events is causal. We introduce the following condition to express that the order of the variables occurring in ψ , or an arbitrary subset of these variables, has an influence on the causality of φ :

OC1: Let $Y \subseteq X$. Changing the order $order_1(Y)$ of the variables in Y to an arbitrary order $order_2(Y)$, while keeping the variables in $X \setminus Y$ at $order_1$, changes φ from true to false. Since the function assigning the order of the events is defined by system execution traces, this corresponds to show that for all bad executions containing X the events in Y

occur always in $order_1(Y)$. If for a subset of X OC1 is not satisfied, the order of the events in this subset has no influence on the causality of φ . In our example, the order of the events Gf, Cc, Tc is causal, because only if the gate fails (Gf) before the car and the train are entering the crossing an accident will be caused, otherwise the gate will close in time to stop the car before the crossing.

3 On-The-Fly Causality Checking

For the off-line causality computation in [KLFL11] all good and bad execution traces need to be computed and stored on disk prior to the causality checking. In this section we outline an algorithm and a corresponding data structure that can be integrated in a depth-first search or breadth-first search algorithm. The resulting algorithm allows for on-the-fly causality computation.

In order to compute causality relationships, it is still necessary to compute good and bad execution traces. If depth-first search or breadth-first search is used for model checking good and bad executions can easily be retrieved. Note that we only want to use reachability properties and hence only need to consider finite execution fragments [BK08]. Bad executions are all those executions where a property violation is detected. If depth-first search is used it is sufficient to print the search stack in order to retrieve the bad execution. When using breadth-first search an additional data-structure is needed that stores the bad execution. There are two categories of good executions that need to be distinguished. The first and obvious one is when the search reaches a state that has no successor states, without finding a property violation on the trace to this state. The second category of good executions is what we call *so-far good executions*. The so-far good executions are executions where there has not yet been a property violation so far. In other words, these executions could be prefixes of either bad or good executions.

Note that the tests defined in Section 2 require to find or establish sub- or super-set relationships between the different system executions. For AC2(1) we need to find good sub executions of the execution under test, since we remove events from X . In AC2(2) we need to find good super-executions where events in W have been added. And in AC(3) we have to show that there is no bad sub execution that fulfills AC1, AC2(1) and AC2(2). Hence inferring causality in the context of a system model can be seen as the problem of finding sub- or super-executions and determining whether the executions are bad or good executions. Based on the relationships of those sub- and super-executions causality relationships can be derived. We use this observation to devise an algorithm for efficient on-the-fly causality checking.

For the efficient storage of the execution traces we have devised a data-structure called sub-set graph. This data-structure enables us to make causality decisions on-the-fly. The sub-set graph is structured into levels where each level corresponds to the length of the execution traces on that level. An execution trace with length one is stored on level one, an execution trace with length two is stored on level two, and an execution trace with length n is stored on the n -th level. Each node represents exactly one execution trace. Figure 1

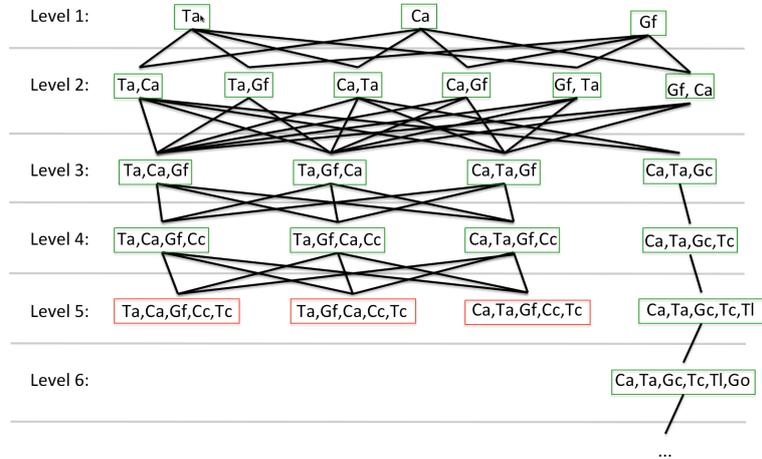


Figure 1: Sub-set-graph of the railroad crossing example.

shows a part of the sub-set graph for the railroad crossing example. The execution traces on adjoining levels are connected by edges indicating sub-set relationships between execution trace t and execution trace t' : For simplicity reasons the edges between executions on the same level are not displayed in the figure. The nodes representing the execution traces are colored in green, red, orange or black according to the following rules:

1. green if it represents a good execution trace and all nodes on the level below that are connected with the node are colored green.
2. red if it represents a bad execution trace and all nodes on the level below that are connected with the node are colored green.
3. orange if it represents a bad execution trace and at least one node on the level below (that is connected with the node) is colored red.
4. black if it represents a good execution trace, but at least one of the nodes on the level below that are connected with the node are colored red.

We now sketch how the sub-set graph can be used to derive causality relationships. Due to space restrictions we omit the proofs in this paper. Note that all red execution traces fulfill AC1, AC3, and AC2(1) and are thus strong candidates to be causal. AC1 is fulfilled by definition, since the events that occur on the path are being considered as ψ and for the execution trace being red the effect φ also has to occur. AC3 is fulfilled, since by definition an execution trace is only colored red, if all its sub-sets are colored green which means there is no sub-set where the effect occurs. AC2(1) is fulfilled because all execution traces on the level below represent execution traces where events from the red execution trace have been removed, that corresponds to setting X to val_2 . Since it is required by the definition that all the sub-sets are green, the outcome of φ can be changed by changing variables in

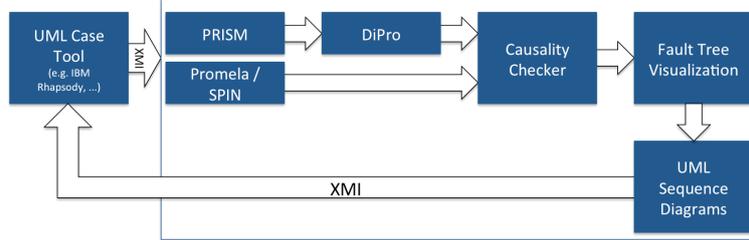


Figure 2: Overview of the QuantUM tool chain.

X , which is sufficient to satisfy AC2(1). Finally we have to test AC2(2) for all red execution traces. For this test all direct black successors (level+1) need to be considered. If the test AC2(2) is fulfilled the execution represents an ordered causal conjunction of events. If the paths are added according to their length, for instance by using breadth first search, the algorithm can stop when the last level that was inserted only consisted out of orange nodes. In our example in sub-set graph in Figure 1 the execution traces Ta, Ca, Gf, Cc, Tc and Ta, Gf, Ca, Cc, Tc and Ca, Ta, Gf, Cc, Tc are marked red. After identifying all red execution traces and checking AC2(2) we need to check OC1. Due to the structure of the sub-set graph, it is sufficient to check for each red execution trace whether there exists a red execution trace on the same level for which the unordered \subseteq relationship holds.

4 Causality Checking of System Models

We are currently implementing the afore mentioned algorithm, together with an automatic translation of QuantUM annotated UML or SysML models to Promela. The resulting tool chain is depicted in Fig. 2. The UML or SysML model is annotated using the QuantUM profile in a UML Case tool, for example IBM Rational Rhapsody. Subsequently the annotated model is exported to an XMI file, which is then imported by the QuantUM tool. In the QuantUM tool the imported model can then either be translated to a PRISM model for the probabilistic analysis or to Promela for qualitative analysis and causality checking. For the state-space exploration in the causality checking module, we use SpinJa [dJR10] the Java implementation of the model checker Spin. If the probabilistic analysis is selected, the integrated version of our DiPro tool can be used to compute probabilistic counterexamples that can then be used for causality checking. The computed causality relationships can then be viewed in the fault tree visualization. Finally, the system executions represented by the fault tree can be exported to an UML sequence diagram in the XMI format, that can then be imported in the UML CASE tool. The QuantUM tool chain enables to integrate causality checking into a model-based engineering development process, because it is fully automated and all inputs for the analysis can be specified on the level of the UML or SysML model. Furthermore, all analysis results are lifted on the level of the UML and SysML model. In the context of model-based engineering we have identified the following

application scenarios for the QuantUM tool:

- Proof of concept of system architectures: fast and automated model-based reachability analysis of particular good or bad states.
- Evaluation of architecture alternatives: evaluation of alternatives based on reachability analysis, automatically generated fault trees and failure mode and effects analysis and based on probabilistic analysis.
- Evaluation of different failure rates for components: with the probabilistic analysis the impact of different failure-rates of components on the rate of a system hazard can be evaluated automatically.
- System Debugging: the causality checking algorithm pinpoints the events that are causal for a certain hazard or mal-function of the system.

5 Related Work

Work documented in [BBDC⁺09] uses the Halpern and Pearl approach to explain counterexamples in functional CTL model checking by determining causality. However, this approach considers only functional counterexamples that consist of single execution sequences. In [GMR10] Goessler et al. present a formal framework for reasoning about contract violations. In order to derive causality the notion of precedence established by Lamport clocks [Lam78] is used. While this captures a partial order of the observed contract violations there is no evidence whether this partial order has an impact on causality or not. Based on Lewis' counterfactual test Groce et al. [GCKS06] establish causality by computing distance metrics between execution traces. The delta between the counterexample and the most similar good execution is identified as causal for the bad behavior. For all the above mentioned approaches it is necessary to compute the counterexamples prior to the causality analysis whereas our approach works on-the-fly.

6 Conclusions

We have discussed how causality relationships can be established in system executions. Furthermore we have shown how our approach can be extended to work on-the-fly and how it can be integrated in model checking algorithms. Finally we have demonstrate application scenarios for causality checking of system models and have sketched the tool chain that we are currently implementing. In future work we plan to fully automated the translation of UML and SysML models that are annotated with the QuantUM profile to the Promela language. A detailed experimental evaluation comparing the on-the-fly algorithm with the existing algorithms for causality checking based on already computed sets of counterexamples is planned as well. Furthermore, we plan to extend the QuantUM tool chain with support for the AADL architecture description language.

References

- [ALFLS11] Husain Aljazzar, Florian Leitner-Fischer, Stefan Leue, and Dimitar Simeonov. DiPro - A Tool for Probabilistic Counterexample Generation. In *Model Checking Software*, LNCS. Springer, 2011.
- [BBDC⁺09] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining Counterexamples Using Causality. In *Proceedings of CAV 2009*, LNCS. Springer, 2009.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking (3rd ed.)*. The MIT Press, 2001.
- [Col04] John Collins, editor. *Causation and Counterfactuals*. MIT Press, 2004.
- [dJR10] Marc de Jonge and Theo Ruys. The SpinJa Model Checker. In *Model Checking Software*, LNCS. Springer, 2010.
- [EL02] Thomas Eiter and Thomas Lukasiewicz. Complexity results for structure-based causality. *Artificial Intelligence*, 2002.
- [GCKS06] 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.
- [GMR10] Gregor Gössler, Daniel Le Métayer, and Jean-Baptiste Raclet. Causality Analysis in Contract Violation. In *Runtime Verification*, LNCS. Springer Verlag, 2010.
- [HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proceedings of TACAS 2006*, LNCS. Springer, 2006.
- [HP05] J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 2005.
- [KLFL11] Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. From Probabilistic Counterexamples via Causality to Fault Trees. In *Proceedings of the Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011*, LNCS. Springer, 2011.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.
- [Lew01] David Lewis. *Counterfactuals*. Wiley-Blackwell, 2001.
- [LFL11a] F. Leitner-Fischer and S. Leue. Quantitative Analysis of UML Models. In *Proceedings of Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2011)*. Dagstuhl, Germany., 2011.
- [LFL11b] Florian Leitner-Fischer and Stefan Leue. QuantUM: Quantitative Safety Analysis of UML Models. In *Proceedings of QAPL 2011*, volume 57 of *EPTCS*, 2011.
- [Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.