Using LTL Rewriting to Improve the Performance of Model-Checker Based Test-Case Generation

Gordon Fraser and Franz Wotawa*
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria
{fraser,wotawa}@ist.tugraz.at

ABSTRACT

Model-checkers have recently been suggested for automated software test-case generation. Several works have presented methods that create efficient test-suites using model-checkers. Ease of use and complete automation are major advantages of such approaches. However, the use of a model-checker comes at the price of potential performance problems. If the model used for test-case generation is complex, then model-checker based approaches can be very slow, or even not applicable at all. In this paper, we identify that unnecessary, redundant calls to the model-checker are one of the causes of bad performance. To overcome this problem, we suggest the use of temporal logic rewriting techniques, which originate from runtime verification research. This achieves a significant increase in the performance, and improves the applicability of model-checker based test-case generation approaches in general. At the same time, the suggested techniques achieve a reduction of the resulting test-suite sizes without degradation of the fault sensitivity. This helps to reduce the costs of the test-case execution.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Performance, Algorithms, Experimentation

*The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsfrderungsgesellschaft mbH. (SFG), the city of Vienna in terms of the center for innovation and technology (ZIT), the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-809446.

A-MOST'07 July 9, 2007, London, UK

Keywords

Automated software testing, LTL rewriting, test-case generation with model-checkers

1. INTRODUCTION

Recently, the use of model-checkers for the automated generation of test-cases has seen increased attention. Counter examples produced by model-checkers are interpreted as test-cases. The approach is straight forward, fully automated and achieves good results.

The main drawback lies within the performance limitations of model-checkers. The state space explosion problem severely limits the applicability of model-checker based testing approaches. It is therefore necessary to find ways to improve the performance and applicability. Another disadvantage of model-checker based approaches is the structure of resulting test-suites. Often, test-suites consisting of large numbers of very short test-cases are created. In contrast, if longer test-cases are created, they often share long identical prefixes. This adversely affects the time and costs of the execution of a resulting test-suite.

While ultimately the size of the model used for test-case generation determines the applicability of any model-checker based technique, there are other factors that contribute to possible bad performance. For example, sometimes large numbers of duplicate test-cases are created. In this paper we identify sources of redundancy that contribute to possible bad performance during test-case creation and execution, and describe an approach based on temporal logic formula rewriting that can be used to reduce the number of model-checker queries significantly. Consequently, the overall time it takes to create a complete test-suite is reduced. In detail, the contributions of this paper are as follows:

- We show how temporal logic formula rewriting can be used to efficiently avoid the creation of redundant test-cases.
- In addition to improving the performance of test-case generation by avoiding unnecessary calls to the model-checker, we show that the rewriting can also be used to extend test-cases such that the resulting number of test-cases and their overall length is reduced. This can be seen as an improvement of the performance of the test-case execution.
- We suggest different algorithms to efficiently generate test-suites using model-checkers and rewriting.

- Finally, we use an example model for a detailed evaluation of the presented ideas in order to show their feasibility.

This paper is organized as follows: First, Section 2 gives an overview of model-checker based testing and introduces all necessary concepts. Then, Section 3 describes how formula rewriting can be incorporated into the test-case generation in order to improve the overall performance. The effects of this combined approach are empirically analyzed in detail in Section 4. Finally, Section 5 discusses the results and concludes the paper.

2. PRELIMINARIES

In this section, the idea of testing with model-checkers is recalled. The available approaches are reviewed, and those suitable for an optimization based on formula rewriting are identified. Then, the necessary theoretical background of testing with model-checkers and properties written in Linear Temporal Logic [27] (LTL) is introduced.

2.1 Testing with Model-Checkers

A model-checker is a tool intended for formal verification. It takes as input an automaton based model of an application and a temporal logic formula. Then, the entire state space of the model is effectively explored in order to determine whether the model and the formula are consistent. If the model-checker determines that the property does not hold on the model, then it returns a trace (counter example), which is a sequence of states leading from the initial state to some state such that the property violation is illustrated. Although this is not strictly correct, we will call a property that does not hold on a model *inconsistent* in this paper.

Each state of a counter example fully describes the values of all variables of the model. The values of input variables (i.e., such variables that are provided by the environment to the system under test) can be used as test data. Similarly, the values of output variables that reflect the behavior of a model upon a certain input can be used as test oracle. Test-case generation is automated by systematically introducing inconsistency to a model that is assumed to be correct.

Two main categories of approaches can be distinguished: The first category uses intentionally inconsistent properties to force the model-checker to create counter examples: *Trap property* based approaches to test-case generation [7, 12, 14, 19, 28] express the items that make up a coverage criterion as properties that claim these items cannot be reached. For example, a trap property might claim that a certain state or transition is never reached. When checking a model against a trap property the model-checker returns a counter example illustrating how the trap property is violated. For example, it shows how the state or transition described by the trap property is reached. This counter example can be used as a test-case. Further related approaches that make use of the model-checker similarly are those based on mutation of properties that "reflect" the transition relation [5] or traps based on requirement properties [23].

The second category of test-case generation approaches uses fault injection to change the model [2, 3, 10, 26]. Here, the model-checker is used to examine the effects of injected faults on specification properties, or to illustrate the differences between changed models and the original model.

In this paper we focus on the first category of approaches. The many different types of trap properties suggested in recent years show how flexible this kind of approach is. The size and thoroughness of a resulting test-suite can be greatly varied by using different trap properties. As reported by Devaraj et al. [13] and Heimdahl et al. [21], simple coverage criteria and trap properties might result in test-cases of poor quality, so more complex coverage criteria are preferable. Such criteria, however, usually lead to more complex trap properties. The performance of the model-checker decreases as property complexity increases. A high number of trap properties also has a negative impact on the overall performance. Another drawback results from the fact that model-checkers are not originally designed to be used for test-case generation. Each trap property can result in a counter example. Often, the same counter example is created several times for different properties. Similarly, a counter example might be subsumed by another, longer counter example. If the number of trap properties is large and/or the model complexity is high, then the creation of each such counter example wastes valuable time. Ideally, creating redundant traces should therefore be avoided in the first place.

The above consideration is related to the idea of test-suite reduction [15], also referred to as test-suite minimization. Test-suite minimization describes the problem of finding a minimal subset of a test-suite that is necessary to fulfill a given set of requirements (e.g., a coverage criterion). Several experiments [20, 22, 30] have shown that minimization has a negative impact on the fault sensitivity. This raises the question of whether avoiding the creation of test-cases for trap properties that are already covered has a similar effect.

Hamon et al. [14] take a slightly different approach to simply model-checking all trap properties directly. They integrate the test-case generation into the model-checker SAL, such that each counter example iteratively extends the previous one. While this approach avoids the creation of duplicate test-cases it can still create test-cases for trap properties that are already covered. Although this can be positive with regard to the overall fault sensitivity, the model-checker is called more often than would be necessary with regard to the used coverage criterion or trap properties, thus potentially impairing the performance.

2.2 Theoretical Background

Model-checkers and temporal logics use Kripke structures as model formalism:

DEFINITION 1 (KRIPKE STRUCTURE). *A Kripke structure K is a tuple $K = (S, S_0, T, L)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, $T \subseteq S \times S$ is the total transition relation, and $L : S \to 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state. AP is the countable set of atomic propositions.*

If the model-checker determines that a model K violates a property ϕ then it returns a trace that illustrates the property violation as a counter example. The trace is a finite prefix of an execution sequence of the model (path):

DEFINITION 2 (PATH). *A path $\pi := \langle s_0, s_1, \ldots, s_n \rangle$ of Kripke structure K is a finite or infinite sequence such that $\forall\, 0 \leq i < n : (s_i, s_{i+1}) \in T$ for K.*

Properties are specified using temporal logics. In this paper, we use future time Linear Temporal Logic (LTL) [27].

An LTL formula consists of atomic propositions, Boolean operators and temporal operators. The operator "$\bigcirc$" refers to the *next* state. E.g., "$\bigcirc\ a$" expresses that a has to be true in the next state. "$\mathcal{U}$" is the *until* operator, where "$a\ \mathcal{U}\ b$" means that a has to hold from the current state up to a state where b is true. "$\Box$" is the *always* operator, stating that a condition has to hold at all states of a trace, and "$\Diamond$" is the *eventually* operator that requires a certain condition to eventually hold at some time in the future. The syntax of LTL is given as follows, with $a \in \text{AP}$:

$$\begin{aligned}\phi \quad ::= \quad & true \mid false \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \\ & \phi_1 \rightarrow \phi_2 \mid \phi_1 \equiv \phi_2 \mid \phi_1\ \mathcal{U}\ \phi_2 \mid \bigcirc\phi \mid \Box\phi \mid \Diamond\phi\end{aligned}$$

The semantics of LTL is expressed for infinite traces of a Kripke structure, where $K, \pi \models \phi$ means that path π of Kripke structure K satisfies the LTL formula ϕ. π_i denotes the suffix of the path π starting from the i-th state, and $\pi(i)$ denotes the i-th state of the trace π, with $i \in \mathbb{N}_0$. The initial state of a trace is $\pi(0)$.

$$K, \pi \models true \quad \text{for all } \pi \tag{1}$$
$$K, \pi \not\models false \quad \text{for all } \pi \tag{2}$$
$$K, \pi \models a \quad \text{iff} \quad a \in L(\pi(0)) \tag{3}$$
$$K, \pi \models \neg\phi \quad \text{iff} \quad K, \pi \not\models \phi \tag{4}$$
$$K, \pi \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \wedge K, \pi \models \phi_2 \tag{5}$$
$$K, \pi \models \phi_1 \vee \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \vee K, \pi \models \phi_2 \tag{6}$$
$$K, \pi \models \phi_1 \rightarrow \phi_2 \quad \text{iff} \quad K, \pi \not\models \phi_1 \vee K, \pi \models \phi_2 \tag{7}$$
$$K, \pi \models \phi_1 \equiv \phi_2 \quad \text{iff} \quad K, \pi \models \phi_1 \text{ iff } K, \pi \models \phi_2 \tag{8}$$
$$K, \pi \models \phi_1\ \mathcal{U}\ \phi_2 \quad \text{iff} \quad \exists i \in \mathbb{N}_0 : K, \pi_i \models \phi_2 \wedge \forall\, 0 \leq j < i : K, \pi_j \models \phi_1 \tag{9}$$
$$K, \pi \models \bigcirc\phi \quad \text{iff} \quad K, \pi_1 \models \phi \tag{10}$$
$$K, \pi \models \Box\phi \quad \text{iff} \quad \forall j \in \mathbb{N}_0 : K, \pi_j \models \phi \tag{11}$$
$$K, \pi \models \Diamond\phi \quad \text{iff} \quad \exists j \in \mathbb{N}_0 : K, \pi_j \models \phi \tag{12}$$

A trap property ϕ is a property that is supposed to be violated by a correct model, resulting in a trace t such that $K, t \not\models \phi$. For example, in order to create a state coverage test-suite, a trap property for each possible state a of every variable x is needed, claiming that the value is not taken: $\Box\neg(x = a)$. A counter example to such an example trap property is any trace that contains a state where $x = a$. This example is a safety property, but trap properties can use the full power of LTL; for example, they can be defined over transitions or sequences of transitions.

Although a trap property is supposed to force creation of a counter example, it is not per definition inconsistent with the model. For example, not all mutants of properties "reflecting" the transition relation as used by Black [5] are necessarily inconsistent with a given model.

Counter examples resulting from trap properties are used as test-cases. A test-case t is a finite prefix of a path π. We consider such test-cases where the expected correct output is included. This kind of test-cases is referred to as *passing* or *positive* test-cases. The result of the test-case generation is a *test-suite*, a set of test-cases. Test-cases created by model-checkers are deterministic, which means that in their basic form they do not handle non-deterministic behavior of the system under test.

Definition 3 (Test-Case). *A test-case t is a finite prefix of a path π of Kripke structure K.*

The number of transitions a test-case consists of is referred to as its *length*. For example, test-case $t := \langle s_0, s_1, ..., s_n \rangle$ has a length of $length(t) = n$.

Definition 4 (Test-Suite). *A test-suite TS is a finite set of n test-cases. The* size *of TS is n. The overall* length *of a test-suite TS is the sum of the lengths of its test-cases t_i: $length(TS) = \sum_{i=1}^{n} length(t_i)$.*

A test-suite created from a set of trap properties automatically achieves maximum coverage of the criterion used to create the traps. Infeasible trap properties are handled implicitly, because they do not result in counter examples. A trap property ϕ is covered, if there exists a test-case t such that $K, t \not\models \phi$. Coverage is also used as an estimate for the test-suite quality:

Definition 5 (Test Coverage). *The coverage C of a test-suite TS with regard to a coverage criterion represented by a set of trap properties $\mathcal{P}$ is defined as the ratio of covered properties to the number of properties in total:*

$$C = \frac{1}{|\mathcal{P}|} \cdot |\{x | x \in \mathcal{P} \wedge covered(x, TS)\}|$$

The predicate $covered(a, TS)$ is true if there exists a test-case $t \in TS$ such that t covers a, i.e., $K, t \not\models a$.

3. ADVANCED TEST-CASE GENERATION WITH LTL REWRITING

The straight forward approach to generating test-cases with a model-checker and trap properties is to simply model-check all trap properties sequentially. This possibly results in duplicate test-cases, or test-cases that are subsumed by other test-cases. We refer to such test-cases as *redundant*. The creation of redundant test-cases unnecessarily consumes time. If model-checking a single property is costly due to the complexity of the model or the property, then the time wasted can be significant. This can be avoided by determining whether a property is already covered by a previous test-case, and only if not so calling the model-checker.

Theoretically, a model-checker could be used to determine whether a test-case is already covered. Ammann and Black [1] present a straight forward approach to represent test-cases as SMV models and then simply model-check the test-case model against a trap property in order to determine whether it is covered. Intuitively, the state space of a test-case model is significantly smaller than that of the full model. The use of regular model-checker techniques and tools, however, is not the optimal solution with regard to the performance. For example, consider a complex model with a large number of trap properties and also a large number of test-cases created up to a certain point in the test-case generation process. During the test-case generation, each test-case would have to be converted to a model. Then, the model-checker would have to be called for this model, in order to check all remaining trap properties. Repeating this each time a test-case is created would result in a large number of model-checker calls, which would be inefficient. Clearly, a more efficient solution is necessary under such circumstances. Markey and Schnoebelen [25] analyze the problem of model-checking paths and show that there are more efficient solutions than checking Kripke structures.

Runtime verification is commonly based upon determination of whether a finite path satisfies a temporal logic

property. In contrast to model-checking it does not use an explicit model, but only execution traces. For example, the NASA runtime verification system Java PathExplorer (JPaX) [16] uses monitoring algorithms for LTL. Properties are rewritten using the states of a trace. That way, violation of a property can be efficiently detected during runtime or during analysis after the execution. This idea is also useful for test-case generation. If the rewriting is applied to the trap properties after creating a test-case, then all trap properties that are already covered can be efficiently detected before calling the model-checker on them.

This section therefore presents an approach that uses LTL rewriting in order to detect already covered trap properties efficiently, and thus increase the overall performance of the test-case generation process.

3.1 LTL Rewriting

An efficient method to monitor LTL formulae is to rewrite them using the states of an execution trace. The rewriting approach we present here is based on work by Havelund and Rosu [17]. Their implementation uses a rewriting engine that is capable of 3 million rewritings per second, which shows that rewriting is an efficient and fast approach. There are approaches that try to further optimize this approach, e.g., [4, 18, 29].

In the domain of runtime verification, one important aspect is the optimization with regard to space demands. Long execution runs can create very long execution traces and lead to space problems. This problem does not exist in the domain of model-checker based test-case generation, as the creation of the traces is the overall objective. In order for counter examples to serve as usable test-cases, their size always has to be within bounds. Therefore, space constraints do not have to be considered when choosing an algorithm for LTL monitoring.

In runtime verification, LTL rewriting is used to decide when a fault has occurred. In the context of test-case generation, the rewriting can be used to determine whether it is necessary to create a trace from a trap property *before* actually calling the model-checker. If there exists a test-case that already covers the trap property, then there is no need to create another test-case for this trap. This is achieved by evaluating a formula using the value assignments of a state, and by rewriting temporal operators. If a trace violates a property, then at a violating state the formula is rewritten to a contradiction, i.e., it is false.

Monitoring LTL properties for runtime verification is generally based on finite trace semantics that are different from the infinite trace semantics presented in Section 2.2. Finite trace semantics consider only finite traces, therefore special treatment of the last state of a finite trace is necessary. For example, one possibility is to assume that the last state is repeated after the end of the trace. Another possibility is to define that no proposition holds after the last state. For example, this changes the meaning of the $\Box$ operator. In the context of model-checker based test-case generation we do not need to consider this. It is only of interest, whether a trap property is violated somewhere along a test-case. If it is not violated at the end of a finite trace, satisfaction is not of interest. The only conclusion that needs to be drawn is that it is not yet covered.

The rewriting of property ϕ with state s is recursively defined below, where $a \in AP$ denotes an atomic proposition, ϕ denotes a temporal logic formula, and $s \in S$ for Kripke structure $K = (S, s_0, T, L)$. $\phi\{s\}$ denotes that state s is applied to the formula ϕ. Application of a state to a formula determines, whether the propositions valid in that state have an effect on the formula. The parts of the formula that refer to the present state are instantiated according to $L(s)$, while affected temporal operators are rewritten according to the rules. The rewriting in Definition 6 differs from that given by Havelund and Rosu [17] in order to reflect the different semantics applied; the final state of a trace is not treated specially.

Definition 6 (State Rewriting).

$$
\begin{aligned}
(\Box\,\phi)\{s\} &= \phi\{s\} \wedge \Box\,\phi && (13)\\
(\bigcirc\,\phi)\{s\} &= \phi && (14)\\
(\Diamond\,\phi)\{s\} &= \phi\{s\} \vee \Diamond(\phi) && (15)\\
(\phi_1\ \mathcal{U}\ \phi_2)\{s\} &= \phi_2\{s\} \vee (\phi_1\{s\} \wedge (\phi_1\ \mathcal{U}\ \phi_2)) && (16)\\
(\phi_1 \wedge \phi_2)\{s\} &= \phi_1\{s\} \wedge \phi_2\{s\} && (17)\\
(\phi_1 \vee \phi_2)\{s\} &= \phi_1\{s\} \vee \phi_2\{s\} && (18)\\
(\phi_1 \rightarrow \phi_2)\{s\} &= \phi_1\{s\} \rightarrow \phi_2\{s\} && (19)\\
(\phi_1 \equiv \phi_2)\{s\} &= \phi_1\{s\} \equiv \phi_2\{s\} && (20)\\
(\neg\phi)\{s\} &= \neg(\phi\{s\}) && (21)\\
a\{s\} &= a \textit{ if } a \notin L(s) \textit{ else } true && (22)
\end{aligned}
$$

As a simple example, consider a trap property that forces the creation of a test-case which contains a transition from a state where x is true and y is false to any state where x is true. To achieve this, the property claims that such a transition does not exist:

$$\phi := \Box\,((x \wedge \neg y) \rightarrow \bigcirc \neg x)$$

A previously checked trap property might have resulted in the following test-case: $t := \langle (x, y), (x, \neg y), (x, \neg y), (\neg x, \neg y) \rangle$. Obviously, ϕ is covered by this test-case as the transition from the second to the third state is the one described by ϕ. In order to detect this, ϕ is rewritten using the states of the test-case sequentially. Application of the first state (x, y) of t is performed as follows:

$$
\begin{aligned}
\phi\{x,y\} &= \Box\,((x \wedge \neg y) \rightarrow \bigcirc \neg x)\{x,y\}\\
&= ((x \wedge \neg y) \rightarrow \bigcirc \neg x)\{x,y\} \wedge \phi)\\
&= ((x \wedge \neg y)\{x,y\} \rightarrow (\bigcirc \neg x)\{x,y\}) \wedge \phi\\
&= ((x\{x,y\} \wedge (\neg y)\{x,y\}) \rightarrow (\neg x)) \wedge \phi\\
&= ((true \wedge false) \rightarrow (\neg x)) \wedge \phi\\
&\text{Which can be simplified to:}\\
&= ((false \rightarrow (\neg x)) \wedge \phi\\
&= true \wedge \phi\\
\phi_1 &= \phi
\end{aligned}
$$

Rewriting with the first state does not change ϕ. The second

state, however, does affect ϕ:

$$\begin{aligned}\phi_1\{x,\neg y\} &= \Box\,((x \wedge \neg y) \rightarrow \bigcirc \neg x)\{x,\neg y\}\\ &= ((x \wedge \neg y) \rightarrow \bigcirc \neg x)\{x,\neg y\} \wedge \phi)\\ &= ((x \wedge \neg y)\{x,\neg y\} \rightarrow (\bigcirc \neg x)\{x,\neg y\}) \wedge \phi\\ &= ((x\{x,\neg y\} \wedge (\neg y)\{x,\neg y\}) \rightarrow (\neg x)) \wedge \phi\\ &= ((true \wedge true) \rightarrow (\neg x)) \wedge \phi\\ \phi_2 &= \neg x \wedge \phi\end{aligned}$$

The third state $(x, \neg y)$ is now applied to ϕ_2:

$$\begin{aligned}\phi_2\{x,\neg y\} &= (\neg x \wedge \phi)\{x,\neg y\}\\ &= ((\neg x)\{x,\neg y\} \wedge \phi\{x,\neg y\})\\ &= (false \wedge \phi\{x,\neg y\})\\ \phi_3 &= false\end{aligned}$$

After rewriting with the third state a contradiction results, therefore it can be concluded that ϕ is covered by t. Hence, there is no need to call the model-checker with the trap property ϕ.

3.2 Test-Case Generation

The basic approach to automated test-case generation with model-checkers is to sequentially call the model-checker with one trap property after the other. Integrating the formula rewriting is therefore easy. Either all remaining properties are checked after creating a test-case, or each property is checked against all previous test-cases before calling the model-checker. The simple algorithm MON in Listing 1 shows the latter possibility. The worst case scenario is that of n trap properties, where each property results in a unique test-case that only covers the property used for its creation. For an average test-case length of l, this means that the rewriting procedure would be called $\sum_{k=1}^{n}(k-1) * l$ times. The maximum number of calls to the model-checker is n, with and without the use of rewriting. Obviously, in order to improve the overall performance, rewriting a property has to be significantly faster than model-checking a property.

Due to the use of the rewriting method the order in which trap properties are selected has an influence on the results with regard to both the performance and the test-suite size. Consider two trap properties ϕ_1 and ϕ_2, resulting in test-cases t_1 and t_2. Now assume that t_1 only covers ϕ_1, while t_2 covers both ϕ_1 and ϕ_2. If ϕ_1 is chosen first, then the model-checker is called for both properties, resulting in t_1 and t_2. Here, t_1 can even be a prefix of t_2, in which case it would be completely redundant. In contrast, if ϕ_2 were chosen first, then the resulting t_2 would cover ϕ_2 and ϕ_1, thus avoiding that the model-checker is called with ϕ_1 in the first place. In Listing 1, the choice of the next trap property is non-deterministic.

Even if the formula rewriting does not show that a trap property is covered, the result of the rewriting can be useful. If the transformation of a property with a test-case results in a formula that is different from the original, this is an indication that the trace affects the property, although it does not yet cover it.

For example, assume a trap property that requires a test-case such that there is a state where x is true, and upon which a state where y is true follows. To achieve this, the

```
function covered(trap, traces)
begin
  for trace in traces do
  begin
    trap' = trap
    for s in trace do
    begin
      trap' = trap' {s}
      if trap' == False then
        return True
      fi
    end
  end

  return False
end

function CreateTestCases_MON(Model M, Traps T)
begin
  traces = []
  for each trap in T do
  begin
    if !covered(trap, traces) then
      traces.append(createTrace(M, trap))
    fi
  end
  return traces
end
```

Listing 1: Algorithm MON: Test-case generation with monitoring by rewriting

trap property expresses that whenever x is true, $\neg y$ follows: $\phi := \Box\, x \rightarrow \bigcirc \neg y$. Assume further a test-case of the shape $t := \langle(\neg x, \neg y), (\neg x, y), (x, \neg y)\rangle$, i.e., the test-case ends with a state where x is true. This test-case could simply be extended with one state where y is true in order to also cover ϕ. Even though t does not cover ϕ, the transformation changes the property to $\neg y \wedge (\Box\, x \rightarrow \bigcirc \neg y)$. The fact that the property changed can be seen as an indication that the test-case can be extended. In the example, only one additional transition is needed to cover ϕ, while a new test-case to cover ϕ starting in the initial state is likely to be longer. In general, the extension sequence of the existing test-case is likely to be shorter than a distinct test-case for the property, as there is no prefix necessary to reach a relevant state, and part of the temporal logic formula already is achieved.

In order to use rewritten properties as trap properties it is necessary to place them within a next-operator, such that the model-checker creates at least one new transition:

$$\phi' = \bigcirc(\phi\{s\})$$

The final state of the trace that is extended serves as the initial state of the new model, therefore the next operator is necessary in order to avoid duplicate evaluation of that state.

The algorithm $EXT1$ in Listing 2 shows how this can be incorporated into the test-case generation. Again, a trap property is checked against the previous test-cases using rewriting. If the trap property is not covered, then the results of the rewriting process are compared to the original

trap property. Any rewritten trap property that differs from the original property suggests that the according test-case is related and can be extended. If there are several test-cases suitable for extension, then one of the test-cases has to be chosen. In Listing 2 this is the second non-deterministic choice besides the choice of the next trap property. The function *extendTrace* calls the model-checker to create a new counter example beginning with the final state of the trace that is to be extended. The actual implementation of this function depends on the model-checker that is used. If the model-checker does not support to explicitly set the initial state, a possible alternative is to rewrite the initial state in the model source file. After changing the initial state, the model-checker is called with the trap property, resulting in a counter example. This new counter example is appended to the previous trace.

```
function CreateTestCases_EXT1(Model M, Traps T)
begin
  traces = []
  for each trap in T do
  begin
    if !covered(trap, traces) then
      if exists trace t : trap{t} ≠ trap then
        extendTrace(t, M, trap)
      else
        traces.append(createTrace(M, t))
      fi
    fi
  end
  return traces
end
```

Listing 2: Algorithm $EXT1$: Extending test-cases with affected trap properties

Finally, the monitoring idea can also be integrated into a test-case generation approach similar to the idea presented by Hamon et al. [14]. In the algorithm $EXT2$ shown in Listing 3 trap properties are used to extend a test-case until it reaches a certain maximum length MAX. If MAX is reached, then a new test-case is started in the initial state of the model.

As with the other algorithms, the choice of the next trap property has an influence on the results. It is possible to use the rewriting to guide this choice. In contrast to the previous two algorithms, $EXT2$ applies the transformation to all remaining trap properties after a test-case extension. All trap properties that are already covered are removed. If a trap property is changed by the transformation, the changed version is stored. It is only necessary to use the extension for the rewriting instead of the whole test-case, after the test-case is extended.

The advantage of this approach is that all trap properties affected by the current test-case are identified. By preferring changed trap properties over unchanged ones, the overall test-suite length can be reduced. If no trap property is affected, Listing 3 prefers those traps properties that were affected earlier during creation of the current test-case, or else chooses one of the remaining trap properties. If a new test-case is started, then the rewritten traps properties have to be reset to their original versions.

```
function CreateTestCases_EXT2(Model M, Traps T)
begin
  traces = [], current_trace = []
  while not empty(T) do
  begin
    trap = trap affected by previous rewriting,
        or random trap
    if length(current_trace) < MAX then
        extendTrace(current_trace, M, trap)
    else
        traces.append(current_trace)
        reset rewritten traps
        current_trace = createTrace(M, trap)
    fi

    for each trap in T do
    begin
      if covered(trap, current_trace) then
        remove trap
      else if trap changed by rewriting then
        save rewritten trap
      fi
    end
  end
  return traces
end
```

Listing 3: Algorithm $EXT2$: Extending up to maximal depth

4. EMPIRICAL EVALUATION

This section describes our prototype implementation of the presented techniques as well as the setup, environment and results of a set of experiments conducted with the prototype.

4.1 Experiment Setup

Our prototype implementation was written with the programming language Python[1]. The LTL rewriting was implemented on top of abstract syntax trees generated by the parser generator Antlr[2]. Clearly, this is not a high performance solution, and the achieved results should therefore be improvable by using more efficient tools and methods. All non-deterministic choices are implemented such that trap properties are chosen sequentially in the order they are created by or provided to the prototype. Version 2.4.1 of the open source model-checker NuSMV [8] is used in our experiments. NuSMV provides symbolic BDD-based model-checking and SAT-based bounded model-checking. In our experiments, the symbolic model-checker was used. All experiments were conducted on a PC with Intel Core Duo T2400 processor and 1GB RAM. For the experiments, the two different maximum depth values 20 and 50 were chosen for $EXT2$. This is supposed to illustrate the effects the choice of the maximum depth has on the performance and quality of the results.

As an example model, a windscreen wiper controller provided by Magna Steyr is used. The model was created manually from a Matlab Stateflow model. The system has

[1]http://www.python.org
[2]http://www.antlr.org

four Boolean and one 16 bit integer input variables, three Boolean and one 8 bit integer output variables, and one Boolean, two enumerated and one 8 bit integer internal variables. The system controls the windscreen heating, speed of the windscreen wiper and provides water for cleaning upon user request. NuSMV reports a total of $2^{44.8727}$ states, 93 BDD variables and 174762 BDD nodes after model encoding. The time it takes to check a single property not only depends on the model, but also on the property itself. For the example model and trap properties, checking one property takes between 2 and 3 seconds in average. The size of the model is not yet problematic for a model-checker based approach, but it is big enough to make performance changes visible while conveniently allowing an extensive set of experiments to be conducted within realistic time.

Table 1: Coverage criteria and resulting trap properties.

Coverage Criterion	Shorthand	Traps
Transition	T	89
Condition	C	320
Transition Pair	TP	6298
Reflection	R	5116
Property	P	345

Trap properties were created automatically for different criteria. Transition and condition coverage are simple criteria based on the NuSMV model. Transition coverage requires each transition relation of the NuSMV model to be covered, while condition coverage tests the effects of each atomic condition in a transition guard expression. Consider the follow excerpt of a NuSMV transition relation:

```
next(var) := case
  condition1 & condition2: next_value;
  ...
esac;
```

A transition coverage trap property for this transition would be $\square\,(condition1 \wedge condition2 \rightarrow \bigcirc \neg next\_value)$. There are two condition coverage traps for this transition relation: $\square\,(\neg condition1 \wedge condition2 \rightarrow \bigcirc next\_value)$ and $\square\,(condition1 \wedge \neg condition2 \rightarrow \bigcirc next\_value)$. Transition pair coverage requires all possible pairs of transitions to be covered. For example, the transition $condition1 \rightarrow value1$ and $condition2 \rightarrow value2$ are combined to the following trap property: $\square\,((condition1 \rightarrow \bigcirc value1) \wedge (\bigcirc(condition2 \rightarrow \bigcirc value2)))$.

In addition to these coverage criteria, we implemented the approach described by Black [5]. Here, trap properties are generated by representing the transition relation of the model as properties (*reflection*) and then applying mutation to the resulting properties. The following mutation operators were used (see [6] for details): STA (replace atomic propositions with true/false), SNO (negate atomic propositions), MCO (remove atomic propositions), LRO, RRO, ARO (logical, relational and arithmetical operator replacement, respectively). This approach subsumes the presented Transition and Condition coverage criteria. In the tables of this paper, we refer to this kind of trap properties as 'Reflection'.

Finally, a set of trap properties was written for the property coverage criterion introduced by Tan et al. [23]. This coverage criterion creates traps from requirement properties, and results in interesting (i.e., showing non-vacuous satisfaction) test-cases for the requirement properties. For this, 30 requirement properties from an informal requirements specification were manually formalized using LTL.

Table 1 lists the numbers of trap properties created for the presented criteria. In our experiments only trap properties that result in counter examples were used. The different algorithms were executed using these sets of trap properties. The time the creation takes is measured as well as aspects of the resulting test-suites. As the order in which trap properties are chosen during the test-case creation can influence the results, we repeated the test-case creation with ten different random orderings for each set of trap properties, and the results stated in the tables below are averaged.

Besides the performance of the different algorithms, it is of major interest to examine the effects on the quality of the resulting test-suites. Therefore, the coverage of each test-suite is measured for all the presented coverage criteria. In addition, the mutation score is measured with regard to the model and to an implementation. A mutant results from a single syntactic modification of a model or program. The mutation score of a test-suite is the ratio of mutants that can be distinguished from the original to the number of mutants in total. A mutant is detected if the execution leads to different results than expected. For this, a test-case can be symbolically executed against a model or a model mutant by converting it to a verifiable model. The transition relations of all variables are given such that they depend on a special state counting variable, as suggested by Ammann and Black [1]. This test-case model can be combined with a mutant model, where the values of the test-case serve as inputs to the mutant model. Symbolic execution is performed by querying the model-checker whether the output values of mutant model and test-case model differ at some point. It is also conceivable to implement this symbolic execution using rewriting techniques.

For the model-based mutation score, the original model was mutated using the same mutation operators as described above for the trap property creation. The resulting mutants were analyzed in order to eliminate equivalent mutants. This is done with a variant of the test-case generation approach proposed by Okun et al. [26]. The original model and a mutant model are combined so that they share the same input variables, and the model-checker is then queried whether there exists a trace such that the output values of model and mutant differ. Therefore, an equivalent mutant is detected if no counter example is returned. This method produced a total of 3303 syntactically valid, non-equivalent mutants. In addition, a Java implementation of the system was written in order to calculate a mutation score using actual execution. Java was chosen for this in order to make use of MuJava [24] for the creation of mutants. MuJava created 218 syntactically valid mutants.

4.2 Results

In the tables below we refer to straight forward test-case creation by sequentially calling the model-checker with all trap properties as *Normal*. Table 2 lists the numbers of test-cases created for each set of trap properties and method. The number of unique test-cases is determined by removing redundant test-cases (i.e., duplicate tests and such tests

that are prefixes of other, longer test-cases and therefore subsumed). While on average 75% of the test-cases created without monitoring (*Normal*) are redundant, this ratio is significantly improved with all presented methods. MON creates almost no redundant test-cases. In average there are 0.3% redundant test-cases for all criteria except the property coverage criterion, which results in 14.76% redundant test-cases. Redundancy can occur with MON if an existing test-case is a prefix of a counter example for another trap property, but does not fully cover it. Therefore, the order in which trap properties are selected has an influence on the amount of redundant test-cases. Theoretically, $EXT1$ can also create such redundant test-cases, as the rewriting cannot detect that an existing test-case is a prefix of another test-case in all situations. This only occurred a few times, and only for the property coverage set of trap properties, where the maximum number of redundant test-cases was 4 out of 89. Except for that, $EXT1$ and $EXT2$ created no duplicate or subsumed traces at all.

All the considered algorithms create smaller test-suites than the straight forward (*Normal*) approach. For $EXT1$ and $EXT2$ this was to be expected, because these approaches are intended to create fewer but longer test-cases. The fact that also MON creates significantly less test-cases indicates that the straight forward approach creates test-suites that contain considerable redundancy, which is discussed below.

Table 2: Average number of unique test-cases.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	31	20.3	16.3	6.3	3.3
C	78	46.5	39.7	15.7	8.0
TP	261	171.5	67.6	53.1	25.2
R	277	187.7	144.3	51.0	22.3
P	197	142.0	81.0	50.0	21.6

Table 3: Average test-case length.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	8	8.8	12.2	26.6	50.9
C	9	10.9	12.7	27.3	53.9
TP	11	11.0	25.2	28.8	57.8
R	8	8.8	11.3	25.0	54.2
P	9	11.0	15.9	25.4	55.5

Table 4: Total test-suite length after removing duplicate/redundant test-cases.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	259	186.9	207.1	170.0	168.5
C	748	522.6	518.6	436.1	433.5
TP	2972	2009.4	1731.2	1556.2	1523.2
R	2469	1717.1	1699.8	1290.2	1226.8
P	1934	1566.4	1332.5	1289.5	1209.9

Table 3 lists the average test-case lengths, and Table 4 lists the overall test-suite lengths. The length of a test-suite is calculated as the sum of the lengths of its unique test-cases. The length of a test-case equals the number of its transitions. As expected, the tables show that the test-cases created using extension of other test-cases are significantly longer than those where test-cases are not extended. At the same time, all methods produce test-suites with a total length smaller than that of test-suites created with the *Normal* method. Interestingly, the overall lengths of $EXT1$ test-suites are sometimes bigger than those of MON. The reason for this is that symbolic model-checking does not necessarily return the shortest counter examples. If there are only few trap properties, then this can result in greater overall lengths. This does not seem to be a problem in general, as the effect is only observable for the quite simplistic transition coverage test-suites. A possible alternative to overcome this problem would be the use of a bounded model-checker, which is guaranteed to find the shortest counter example. If all trap properties are of a similar structure, it is also conceivable to simplify the rewriting. For example, if all trap properties are of the type $\square(x \rightarrow \bigcirc \neg y)$, then it would be sufficient to use $\neg y$ as rewritten trap for trace generation instead of $\neg y \wedge \square(x \rightarrow \bigcirc \neg y)$, after a state where x is true. In order to keep our approach independent of the type of trap properties used, we do not consider such a modification to the rewriting technique used in this paper. This, however, could potentially be interesting further research.

Table 5: Standard deviation of test-suite lengths.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	0	16.7	22.3	19.0	16.7
C	0	17.6	26.9	38.9	30.4
TP	0	48.4	34.5	54.6	60.4
R	0	40.7	61.7	53.5	73.7
P	0	29.2	79.7	65.4	65.0

Table 6: Redundancy.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	31.25%	26.91%	11.7%	2.99%	0.94%
C	30.84%	26.89%	16.02%	3.96%	1.48%
TP	33.38%	31.19%	5.68%	4.25%	1.62%
R	49.25%	46.29%	20.94%	4.75%	1.64%
P	21.52%	16.43%	9.42%	4.20%	1.47%

The influence of the order in which trap properties are selected has been pointed out several times in this paper. As an example of the effects of these choices, Table 5 lists the standard deviation of the total test-suite lengths. The total test-suite length was chosen because it is representative of the performance and the quality. The number of transitions a test-suite consists of reflects the actual savings compared to the original, and is also proportional to the performance improvements. The table shows that the deviation is not significant. In general, the deviation in the total test-suite length is significantly smaller than the achieved reduction compared to the normal test-suite. Although only the small subset of 10 different orderings out of the set of possible permutations was used, we can safely conclude that simply using trap properties in the order they are generated or passed to the test-case generation process is feasible. Still, some kind of heuristic to guide the selection of trap properties could be useful to further improve the performance.

In [11] we introduced a redundancy measurement for test-

suites created with model-checkers. The redundancy value represents the amount of common prefixes. Test-suites with high redundancy values are less efficient at detecting faults as the test-cases traverse the same passages repeatedly and unnecessarily. Table 6 shows the redundancy values for all test-suites. The amount of redundancy saved by MON is proportional to the savings in the test-suite size. $EXT1$ results in significantly less redundancy. In general, the redundancy seems to be correlated to the ratio of the number of test-cases to the average test-case length. Therefore, the redundancy of test-cases created with $EXT2$ and a maximum length of 20 contain more redundancy than those with a maximum length of 50.

Table 7: Creation time.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	3m35s	59s	55s	49s	49s
C	12m32s	2m01s	2m02s	2m15s	2m12s
TP	247m54s	25m17s	20m00s	41m24s	39m17s
R	218m35s	13m01s	10m46s	12m51s	12m34s
P	13m20s	7m08s	7m11s	10m06s	9m45s

Table 8: Average number of model-checker calls.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	89	20.3	19.4	18.6	18.4
C	314	46.5	44.4	44.3	44.0
TP	6298	171.8	165.2	143.8	152.1
R	5410	189.9	179.1	170.9	168.8
P	342	154.3	126.7	117.6	113.4

Table 7 shows the total time consumed for test-case generation for each approach and test-suite. For all algorithms, the savings are significant. This performance improvement is caused by the reduced number of actual calls to the model-checker, as listed in Table 8. The overhead added by the large amount of rewritings is negligible, as long as the model complexity makes the model-checking process costly enough, and the number of trap properties and test-cases is within bounds. Our prototype is comparatively slow with regard to rewriting and could be optimized. The average test-case length seems to be related to the number of model-checker calls; the longer a test-case, the more trap properties it covers. Therefore, the $EXT2$ algorithm with a maximum depth of 50 performs the least model-checker calls in most cases.

Table 9: Coverage: Transition coverage test-suites.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	100%	100%	100%	100%	100%
C	71.66%	71.66%	72.93%	73.25%	73.25%
TP	18.75%	18.75%	27.42%	34.77%	37.66%
R	88.87%	88.87%	89.45%	89.72%	89.74%
P	30.72%	30.72%	30.14%	31.30%	31.01%

In order for the presented algorithms to be feasible, it is important that the coverage with regard to the criterion used for test-case generation is not negatively affected. Therefore, the coverage of all test-suites is measured with regard to the criterion used for creation as well as all other criteria. Tables 9, 10, 11, 12 and 13 list the results of the coverage analysis. For each set of trap properties used for test-case creation there is one table. Only such trap properties that result in counter examples are used, therefore a normal test-suite achieves 100% coverage of the criterion used for creation. As expected, the tables show that this coverage value is not affected by any of the alternative algorithms. In contrast, there is a slight variation with regard to the coverage of criteria not used for creation. MON has a minimal negative impact on the coverage in some cases. Monitoring avoids the creation of test-cases where the according trap property is already covered. However, when called on an already covered trap property, the model-checker might return a different trace than the one that already covers the trap property. Such traces are not created when monitoring the trap properties. This has no effect on the coverage criterion used to create test-cases but explains the small possible degradation of coverage of other criteria in some cases.

Table 10: Coverage: Condition coverage test-suites.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	100%	100%	100%	100%	100%
C	100%	100%	100%	100%	100%
TP	25.93%	25.64%	34.90%	44.86%	42.66%
R	93.59%	92.07%	92.27%	93.12%	92.66%
P	38.26%	37.39%	37.39%	38.26%	37.68%

Table 11: Coverage: Transition-Pair coverage test-suites.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	100%	100%	100%	100%	100%
C	85.99%	85.99%	85.67%	85.03%	84.39%
TP	100%	100%	100%	100%	100%
R	93.18%	93.18%	93.18%	93.60%	93.31%
P	36.81%	36.81%	36.81%	37.10%	37.10%

While there are still some cases where both $EXT1$ and $EXT2$ achieve lower coverage (e.g., coverage of the transition pair traps by the reflection test-suites), in the majority of cases the coverage is about the same or higher than that of normal test-case generation. The effects on the coverage are generally difficult to predict as they depend very much on the type of trap properties, redundancy of the test-cases, how well the test-cases can be extended and many other factors. From our experiments we conclude that the presented approaches can safely be applied without significant negative effects on the coverage with regard to the model.

Finally, the mutation score is measured as an indicator for

Table 12: Coverage: Reflection coverage test-suites.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	100%	100%	100%	100%	100%
C	96.82%	94.59%	94.90%	96.18%	96.82%
TP	60.78%	31.55%	30.41%	50.71%	57.05%
R	100%	100%	100%	100%	100%
P	41.74%	41.16%	42.90%	44.64%	44.06%

Table 13: Coverage: Property coverage test-suites.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	93.26%	93.26%	93.26%	100%	93.26%
C	92.36%	92.36%	91.72%	95.54%	92.68%
TP	40.95%	40.66%	46.52%	51.78%	52.21%
R	94.07%	93.96%	93.64%	96.47%	93.86%
P	100%	100%	100%	100%	100%

Table 14: Mutation scores using model mutants.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	59.52%	59.52%	64.64%	69.48%	72.69%
C	72.42%	70.30%	74.63%	84.80%	84.50%
TP	90.40%	90.37%	90.31%	88.19%	89.92%
R	78.84%	76.96%	86.92%	92.40%	93.73%
P	79.35%	79.35%	81.83%	86.71%	85.17%

the fault sensitivity of the test-suites. Table 14 shows the results using the model mutants. The results are similar to those with regard to coverage: MON performs marginally worse than *Normal* test-suites, while in most cases $EXT1$ and $EXT2$ achieve higher scores. Only the extended transition pair test-suites perform slightly worse than the original test-suite. These results are also reflected in Table 15, which lists the mutation scores calculated with the Java mutants.

5. CONCLUSION

In this paper, we have presented an approach that integrates LTL rewriting known from runtime verification into model-checker based test-case generation. If a sufficiently simple model is used for test-case generation, then a straight forward approach of model checking all trap properties is applicable without problems. If, however, the model size increases to a point where the verification of a single property takes significant time, then the applicability of a straight forward approach declines. Although a model is usually more abstract than the actual program it represents, the model size can still be significant. For instance, automatic conversion (e.g., Matlab Stateflow to SMV) can result in complex models.

The integration of LTL monitoring techniques results in a significant performance increase in such a scenario. This is achieved by avoiding unnecessary calls to the model-checker. Already covered trap properties are detected using the faster method of LTL rewriting instead of model-checking. The results of this rewriting also help to extend test-cases instead of creating only distinctive test-cases. As a consequence, the overall number and length of test-cases in a test-suite are reduced. At the same time, the quality of the test-suites is not adversely affected in general, and even enhanced in many cases. An increased fault sensitivity with a smaller size is achieved as the test-suite redundancy is reduced.

Table 15: Mutation scores using implementation mutants.

Crit.	Normal	MON	EXT1	$EXT2_{20}$	$EXT2_{50}$
T	76.15%	76.15%	78.90%	82.11%	81.19%
C	80.28%	79.36%	83.49%	84.40%	84.86%
TP	86,70%	86,70%	86,70%	86.24%	86.24%
R	82.57%	80.73%	86.24%	87.61%	88,07%
P	77,06%	77,06%	77,06%	85,78%	83,49%

The presented algorithms apply to all approaches where a single model is checked against multiple trap properties in order to create test-cases. There are also approaches that use mutation of the model instead of trap properties. The rewriting cannot directly be applied to those methods. This is considered for further research. The presented rewriting is restricted to LTL, which is sufficient for trap properties in most cases. Even if CTL [9] is sometimes used in the literature, this is only using the 'all paths' quantifier (ACTL), and such a subset that allows linear counter examples. The resulting trap properties could therefore also be represented using LTL.

There are several non-deterministic choices in the presented algorithms. The experiments have shown that a random choice achieves very good results, but further optimizations are conceivable. Possible future research therefore includes the search for suitable heuristics to guide these choices.

Model-checkers are not originally intended for test-case generation. Therefore, they are clearly not optimized for this task. It is necessary to identify areas where drawbacks result from this fact. The introduction of rewriting techniques to model-checker based test-case generation improves the applicability. Even though the performance increase using LTL rewriting can be significant, this does enable the use of model-checker based test-case generation for models of deliberate complexity. Often, a model can cause the model-checker to take too long to check even a single property. In such a case, abstraction seems to be the only possibility to allow test-case generation.

6. REFERENCES

[1] P. Ammann and P. E. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 239–248, Washington, DC, USA, 1999. IEEE Computer Society.

[2] P. Ammann, W. Ding, and D. Xu. Using a Model Checker to Test Safety Properties. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS 2001)*, pages 212–221, Skovde, Sweden, 2001. IEEE.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, 1998.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *PADTAD'04, Parallel and Distributed Systems: Testing and Debugging*, 2004.

[5] P. E. Black. Modeling and Marshaling: Making Tests From Model Checker Counterexamples. In *Proc. of the 19th Digital Avionics Systems Conference*, pages 1.B.3–1–1.B.3–6 vol.1, 2000.

[6] P. E. Black, V. Okun, and Y. Yesha. Mutation Operators for Specifications. In *Proceedings of the*

Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00), 2000.

[7] J. R. Callahan, S. M. Easterbrook, and T. L. Montgomery. Generating Test Oracles Via Model Checking. Technical report, NASA/WVU Software Research Lab, 1998.

[8] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A New Symbolic Model Verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[9] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[10] G. Fraser and F. Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006.

[11] G. Fraser and F. Wotawa. Redundancy based test-suite reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2007. To Appear.

[12] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests From Requirements Specifications. In *ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687, pages 146–162, Toulouse, France, September 1999. Springer.

[13] D. L. George Devaraj, Mats P. E. Heimdahl. Condition Based Coverage Criteria: To use or not to use that is the question. To be published, 2005.

[14] G. Hamon, L. de Moura, and J. Rushby. Generating Efficient Test Sets with a Model Checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 261–270, 2004.

[15] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[16] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[17] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.

[18] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004.

[19] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-Generating Test Sequences using Model Checkers: A Case Study. In *Third International International Workshop on Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 42–59. Springer Verlag, October 2003.

[20] M. P. E. Heimdahl and G. Devaraj. Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In *ASE*, pages 176–185. IEEE Computer Society, 2004.

[21] M. P. E. Heimdahl, G. Devaraj, and R. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *HASE*, pages 178–186. IEEE Computer Society, 2004.

[22] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–209, 2003.

[23] I. L. Li Tan, Oleg Sokolsky. Specification-based testing with linear temporal logic. In *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI'04)*, pages 493–498, 2004.

[24] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.

[25] N. Markey and P. Schnoebelen. Model checking a path (preliminary report). In *Proc. Concurrency Theory (CONCUR'2003), Marseille, France*, volume 2761 of *Lect.Notes Comp. Sci*, pages 251–265. Springer, Aug. 2003.

[26] V. Okun, P. E. Black, and Y. Yesha. Testing with Model Checker: Insuring Fault Visibility. In N. E. Mastorakis and P. Ekel, editors, *Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, pages 1351–1356, 2003.

[27] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA*, pages 46–57. IEEE, 1977.

[28] S. Rayadurgam and M. P. E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91, Washington, DC, April 2001. IEEE Computer Society.

[29] G. Rosu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engg.*, 12(2):151–197, 2005.

[30] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 34, Washington, DC, USA, 1998. IEEE Computer Society.