Property-Driven Benchmark Generation

Bernhard Steffen[1], Malte Isberner[1,2], Stefan Naujokat[1], Tiziana Margaria[3], and Maren Geske[1]

[1] Dortmund University of Technology, Chair for Programming Systems
Dortmund, D-44227, Germany
{steffen,malte.isberner,stefan.naujokat,maren.geske}@cs.tu-dortmund.de

[2] Carnegie Mellon University
Moffett Field, CA, USA

[3] Universität Potsdam, Chair Service and Software Engineering,
Potsdam, D-14482, Germany
margaria@cs.uni-potsdam.de

Abstract. We present a systematic approach to the automatic generation of platform-independent benchmarks of tailored complexity for evaluating verification tools for reactive systems. Key to this approach is a tool chain that essentially transforms a set of automatically generated LTL properties into source code for various formats, platforms, and competition scenarios via a sequence of property-preserving steps. These steps go through dedicated representations in terms of Büchi Automata, Mealy machines, Decision Diagram Models, Code Models, and finally the source code of the chosen scenario. The required transformations comprise LTL synthesis, model checking, property-oriented expansion, path condition extraction, theorem proving, SAT solving, and code motion. This combination allows us to address different communities via a growing set of programming languages, tailored sets of programming constructs, different notions of observation, and the full variety of LTL properties - ranging from mere reachability over general safety properties to arbitrary liveness properties. The paper illustrates the whole tool chain along accompanying examples, emphasizes the current state of development, and sketches the envisioned potential and impact of our approach.

Key words: Benchmark generation, LTL synthesis, model checking, property-oriented expansion, path condition extraction, theorem proving, SAT solving, code motion

1 Motivation

Twenty years ago, at CAV 1993 in Elounda (Crete), the essence of the business meeting could have been summarized as "We have developed numerous powerful methods and tools, what we are missing are appropriate problems." Since then, numerous impressive case studies have been presented, competitions and challenges have been organized, and industrial cooperations have been conducted, but all these initiatives remained very partial: they focused on very specific scenarios, thus limiting their potential for the generalization of the results and fair comparison of technologies, let alone for establishing a clear application profile for the wealth of academic and industrial tools. What is needed instead for an unbiased evaluation of the tools' application profiles is an infinite source of benchmark problems of tailored complexity, with known properties, accessible to everybody.[4] Only this way can guarantee reproducibility and reveal solutions that are too problem-specific - both very important properties when trying to establish tool profiles as a means for profile-based recommendation for a tool or a technology. In short, what is missing are classified benchmark suites that are expressive enough to reveal the individual tools' strength and weaknesses both at the conceptual and the pragmatic level.

In this paper, we present a systematic approach to the automatic generation of platform-independent benchmarks of tailored complexity for evaluating verification tools for reactive systems. In order to optimally clarify 1) the intuition behind our approach, 2) the three major challenges, and 3) its technical ingredients we explain each dimension in a separate subsection. The overall benchmark generation process has its own full section (cf. Sec. 2), followed by individual sections on its steps (cf. Secs. 3-9.

1.1 The Intuition

The intuition behind our approach is quite similar to that of the now popular outdoor game geocaching[5]: in geocaching, recognizing the points of interest once one is there is trivial. In our case we simply leave a mark that is easily recognizable by any tool. The point of the game is to reach the correct location despite dedicated hurdles. Our corresponding "riddles" are of course program analysis questions to be solved in the overall game. With this mindset, the intuition behind our automatic benchmark generation process is easy to explain:

- We randomly place "treasures" and connect them with an envisioned feasible path, i.e., a path that the player will have to follow. In our case, this results in a Mealy machine that we obtain as a result of a synthesis process from a more declarative specification in terms of LTL properties.
- We randomly insert "riddles" along the path. In our case, these are randomly generated program structures that need to be correctly analyzed to win the game. The point is that the insertion of these obstacles is property preserving. This is realized by applying a sequence of elementary, provably property-preserving insertion/transformation steps.

Like in geocaching, even though recognizing the treasures is simple, it is possible to construct problems of almost arbitrary complexity simply by changing the

[4] Of course, dedicated real(-istic) problems are also extremely important, but because of their "singularity" and a priori unknown properties, they are not suitable for a careful, wide-range profile analysis.

[5] in particular "mystery" and "multi" caches. See http://www.geocaching.com/

riddles. This could mean moving from simple reachability problems to safety, liveness or even arbitrary LTL properties in program structures of increasing complexity that may comprise complex conditions, data structures, loops, or as new vision even polymorphism and virtual methods. The configurable scale and randomization of the generation process guarantees that each of the problems is entirely distinct from any other.

What is truly different from geocaching is the fact that once given the complexity profile, these problems are fully automatically generated without any human interaction. Consequently, the solution can be kept secret even from the organizers of a challenge/competition, enabling them to participate themselves without any advantage – provided that the profile and the code/specification of the benchmark generator is made public.

1.2 The Three Major Challenges

Three quite general questions need to be answered when aiming at quality benchmark problem generation:

- **Where and how to throw the dice:** In order to get balanced benchmarks of challenging size, one needs a fine granular concept of randomization. Our multi-step generation process (cf. Sec. 2) is explicitly designed for aspect-specific randomization, as will be illustrated in the main sections of the paper.
- **How to impose/guarantee the properties:** Obviously, this must be done by construction, as extracting the properties from the final benchmark is meant to be a challenging (open) problem. Central throughout the whole process is therefore the fact that maintaining language inclusion wrt. the ω-language of the Büchi automaton synthesized from the LTL formulas is sufficient to guarantee property preservation. This simple principle is strictly followed during the whole generation process depicted in Fig. 1, and it can be applied straightforwardly to further enhance the program structure with almost arbitrarily complex program elements.
- **How to avoid generator footprints (hints):** An important goal of benchmark generation must be that the generated problems are not biased (beyond their abstract profile), and that it does not make sense to exploit any knowledge about the generation process. Optimally, even the developers of the benchmark generator should have no (significant) advantage.

1.3 Summary of the Involved Technologies

Technical key to this approach is a tool chain that essentially transforms a set of automatically generated (or hand-selected) LTL properties via a series of property-preserving steps into source code for various formats, platforms, and competition scenarios. These steps pass various phases characterized by representations in terms of Büchi automata, Mealy machines, code models, and finally the source code of the chosen scenario. The required transformations are based on

LTL synthesis [1, 2], model checking [3], property-oriented expansion [4], path condition extraction [5], theorem prover/SAT solver-based linking of pre and post conditions [6, 7], and (semantic) code motion [8–12].

In order to make the benchmarks accessible by most tools and interesting for different communities, we aim at supporting

- a growing set of programming languages (e.g. Java, Scala, C/C++, C#, Promela,...),
- a tailored use of programming constructs (loops, linear and non-linear arithmetics, methods calls, virtual methods...) and diverse data structures (integers, arrays, lists, floating-point numbers, ...),
- different notions of observation: reaching program labels, exceptions being thrown, output written to the console, method invocations, ...
- the full variety of LTL properties, ranging from mere reachability over general safety properties to liveness properties and arbitrary mixtures thereof.

Section 2 sketches the process (cf. Fig. 1) connecting all these technologies to fully automatically generate randomized benchmark problems of guaranteed property profile (cf. Therorem 1).

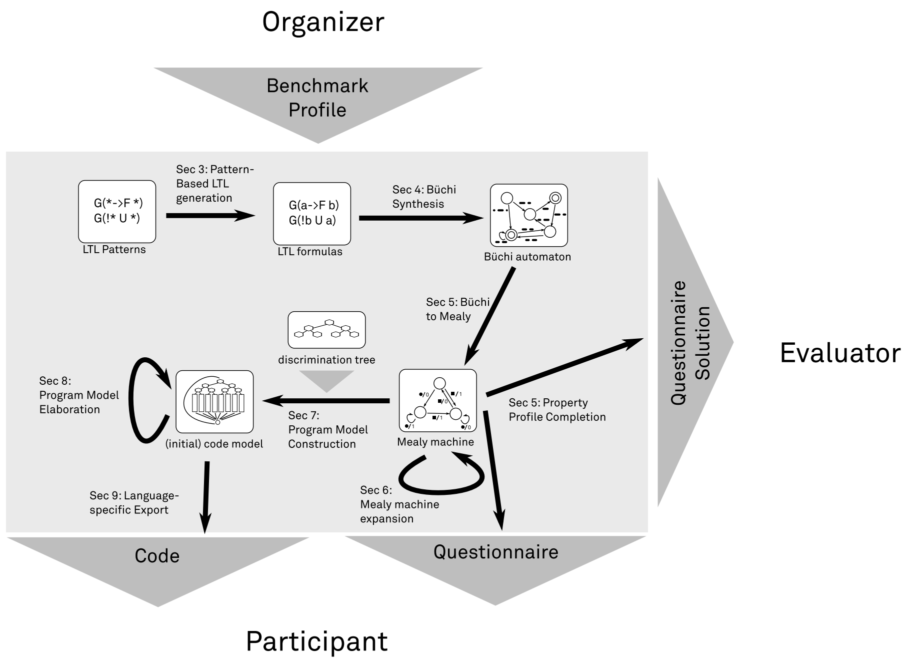


Fig. 1. Overview of property-preserving steps

The paper illustrates the whole tool chain along accompanying examples, emphasizes the current state of development, discusses our experience with our first version gained during the RERS Challenge 2012 [13], and sketches the envisioned potential and impact of our approach. We will use the corresponding tool to generate the problems for the RERS Challenge 2013, which will take place as a satellite event of ASE 2013 in Mountain View (USA) in November.

We will now sketch the conceptual steps of the tool chain (cf. Fig. 1) each in its own dedicated section. Subsequently, Sec. 10 discusses our conclusions and perspectives.

2 The Generation Process from a Bird's Eye View

Our solution is centered around a property-driven benchmark generation process (cf. Fig. 1). It starts with a randomized property selection that can be transformed to Büchi automata using standard techniques, and then it successively enriches/changes the representations via various property-preserving model and code transformations to provide problems of almost arbitrary complexity and size. Its characteristic conceptual steps, which will be discussed in more detail in the remainder of the paper, can be sketched as follows from a bird's eye view:

Pattern-Based LTL Generation: In this step we randomly choose and then instantiate LTL property specification patterns [14] that we partition into a small *defining set* used in the subsequent synthesis step, and a larger set of additional properties whose validity is later checked on the synthesized model via model checking. Typically, we generate around 100 properties, about ten of which can be defining, in order to still allow for automated synthesis.

LTL Synthesis: Here we can apply any of the standard algorithms that translate an LTL formula into a Büchi automaton representing all its models (satisfying paths). Our current implementation uses LTL2Buchi [2], but, e.g., LTL2BA [1] could have been chosen as well.

From Büchi to Mealy: Completing the Property Profile: The point of this step is the generation of a concrete reactive system model (e.g. a Mealy machine) from the Büchi automaton that represents *all* words/paths satisfying the defining properties. The construction of this Mealy machine is randomized and can be customized in various dimensions, e.g., the size of the model, the size of the input and output alphabets, the density of the transition graph etc.. The subsequent completion step for the property profile can straightforwardly be realized by model checking the additional (non-defining) properties on the generated Mealy machine. After this step the property profile for this benchmark is fixed.

Mealy Machine Expansion: These Mealy Machines are then enlarged via randomized property-oriented expansion (POE) [15] and by introducing unreachable states. Both transformations are incremental and can be stopped at any moment, e.g. when a certain threshold of states is reached.

Mealy-to-Program Model Transformation: The transformation is based on viewing Mealy machines operationally as simple loops of guarded commands, whose guards precisely check for the correct state identification. The idea behind the transformation is to replace this simple guard structure with a complex, semantically equivalent decision structure in terms of a *discrimination tree* [6], which essentially resembles a complex nested "if-then-else". The discrimination tree itself is randomly generated both in its branching stucture and its node labeling with predicates, which may well use multiple variables, arithmetics, relations, and data structures. The states of the Mealy machine are modelled by equivalence classes of a randomly constructed partition of the discrimintation tree's leaves. Key to the required property preserving transformation is to establish the correct wiring by means of adequate assignments that guarantee that the postcondition of a transition implies the precondition of the target state, and to extract corresponding complex guarded commands. Path condition generation and SAT solving/theorem proving provide a powerful basis for deriving non-trivial conditional structures (cf. Sec. 7).

Elaboration of the Program Model Structure: We employ data-flow analysis and transformation techniques to randomly elaborate the program model structure along both the logical and the control structure:

– *Overcoming the Simple Loop Structure:* Up to here, the programs are still reminiscent of Event Condition Action Systems [16, 17] or PLC programs [18], a structure the 2012 RERS Challenge focused on [13]. Using randomized property-oriented expansion [4], this structure can be generalized to obtain quite general "while"-program-like structures[19].
– *De-Localization of the Logical Reasoning:* Our approach for establishing postconditions that match the required preconditions characterizing the subsequent state is local, i.e., a Mealy machine can be reconstructed essentially by a pairwise check of the various preconditions and postconditions. We therefore employ code motion techniques [8, 10, 11, 20, 12, 9, 21] for de-localizing the information and therefore require a global analysis for reconstructing the transition relation.

Language Extraction: Currently we have implemented a simple template mechanism which works for Java and C/C++ and maintains the behavioral and structural properties of its argument code models. A wide variety of supported target languages is crucial to fulfill the goal of serving the needs of as many tool developers as possible, so this is going to change in the future.

As a consequence our our strict policy of property-preservation our benchmark generation process guarantees the following theorem:

Theorem 1 (Correctness).
The language-specific code generated by our benchmark generation process is guaranteed to satisfy the property profile established in Sec. 5.

This feature allows for cross-community challenges. The idea of a language specific export is quite general and was indeed used by the winner of the RERS Challenge 2012 Jaco van de Pol to generate Promela input required for his tool landscape. We aim at lowering the entry hurdle for participants by providing them with various formats. It should be noted, however, that providing Promela code is not quite the same as providing code of the other mentioned programming languages as it requires non-trivial design decisions on the adequate abstraction. Thus one may consider to provide a Promela generator instead, which is parameterized in the abstraction.

3 Property Generation

The goal of fully automatically generating a large set of interesting benchmarks inevitably calls for the random generation not only of system models, but also of their underlying properties. While this could be achieved by randomly generating LTL syntax trees, the resulting formulae would most likely be very different from what real-life property specifications look like.

Instead of randomly generating whole formulae, we therefore randomly instantiate specification *patterns*, like for example those described in the seminal work of Matthew Dwyer et al. [14]. Additionally to producing more realistic properties, this approach also yields the benefit that an intuitive textual description can be provided for each LTL formula. Such properties might include, but are not limited to, the following:

– *Absence*, e.g. $\mathbf{G}\,\neg\mathsf{bad}$: "Action bad does never occur"
– *Existence*, $\mathbf{F}\,\mathsf{good}$: "Eventually, action good will be performed"
– *Response*, $\mathbf{G}(\mathsf{a} \Rightarrow \mathbf{F}\,\mathsf{b})$: "Whenever event a occurs, this will lead to action b being observed eventually"

In a challenge scenario, each of these properties (plus dozens of others) would have to be checked separately on the given system.

In the generation phase, however, a small subset of all properties is selected to constrain the randomized on-the-fly construction of the initial Mealy machine model [22].[6] This step is prepared by synthesizing a Büchi automaton from the selected properties, as the next section will detail.

[6] We chose Mealy machines as our intermediate model structure because of their input/output distinction. Of course also labeled transition systems [23] or IO automata [24] could have been chosen as well.

4 Büchi synthesis

Formally, from the set Φ of all generated properties we select two subsets $\Phi^+, \Phi^- \subseteq \Phi$ of properties which should (resp. should not) hold *by construction*. A Büchi automaton A_ψ is created from the conjunct of the selected property sets

$$\psi = \bigwedge_{\phi \in \Phi^+} \phi \wedge \bigwedge_{\phi \in \Phi^-} \neg\phi,$$

Generating Büchi automata from LTL formulae is a very expensive task for larger formulae, thus the choice of the size of the sets Φ^+ and Φ^- crucially depends on how much computing power should be invested in this step. Usually, we obtain fairly good results already with very small sets: the problems of the RERS challenge 2012 were generated using between four and six defining properties.

In the example of the previous section, the conjunct of all non-negated specification formulae is

$$\psi = \mathbf{G}\,\neg\mathsf{bad} \wedge \mathbf{F}\,\mathsf{good} \wedge \mathbf{G}(\mathsf{a} \Rightarrow \mathbf{F}\,\mathsf{b}).$$

Figure 2 shows the corresponding Büchi automaton, which already contains a significant number of transitions given the rather small and simple specification formula. This Büchi automaton was generated using the LTL2BA algorithm [1], but other algorithms such as LTL2Buchi [2] could have been used as well. Note

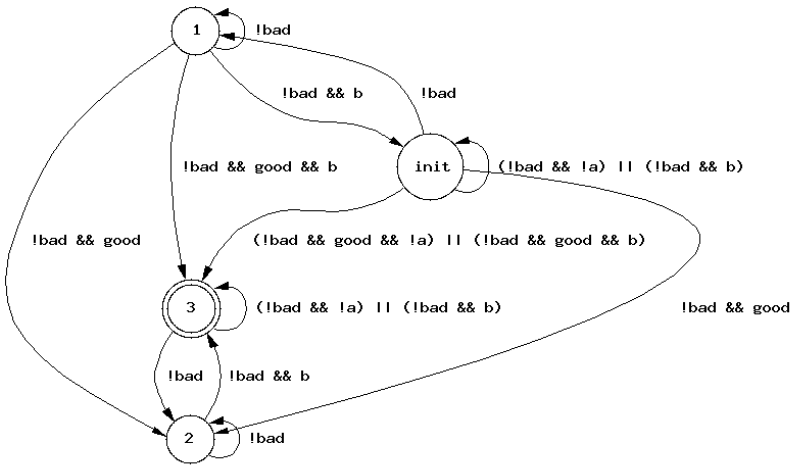

1
!bad
!bad && b
!bad
init
(!bad && !a) || (!bad && b)
!bad && good && b
!bad && good
(!bad && good && !a) || (!bad && good && b)
3
(!bad && !a) || (!bad && b)
!bad && good
!bad
!bad && b
2
!bad

Fig. 2. Resulting Büchi automaton for set ψ

that this automaton, as it is, is not a valid model fulfilling the given properties: first, assuming a granularity of one action/observation per step, some of the transitions such as `!bad && good && b` can never be realized, as *either* action `good` *or* action `b` occurs. Similarly, transitions like `!bad && b` can be shortened to just `b`. The second aspect regards the set of allowed input symbols (events). The transition label `!bad` represents an otherwise unrestricted set of alphabet symbols, although generally we assume that the set of observable events is constrained in one way or the other. Finally, there is no equivalent for accepting states in a Mealy machine: in each concrete instantiation of a system it has to be ensured that the reflexive transition labeled `(!bad && !a) || (!bad && b)` of state `init` cannot be taken infinitely often in a row.

In the next section, we will describe how to construct a Mealy machine whose infinite runs all satisfy the constraints imposed by this Büchi automaton.

5 From Büchi to Mealy: Completing the Property Profile

The construction of a concrete Mealy machine from a constraining Büchi automaton is based on the idea of constructing on-the-fly a product automaton. Starting with the initial state of the Büchi automaton and a freshly created initial state of the Mealy machine, successor states are either newly generated or taken from the set of existing Mealy machine states. This has to be done consistently with the Büchi automaton, i.e., a transition between two states in the Mealy machine needs to match a transition between the associated Büchi states. When creating states in the Mealy machine, several Büchi states might be eligible for being associated with the new state due to non-determinism. In this case, one can be chosen at random. This selection might eliminate accepting runs during the model construction, but does not affect correctness.

Special care has to be taken when transitions to existing states in the Mealy machine are created: this introduces loops. The Büchi acceptance criterion requires every loop in the model includes at least one accepting state. At first, this can be easily achieved by creating back edges only to accepting states. As the model construction proceeds, a set of *safe states* gradually emerges: these are states where all the outgoing infinite paths are accepting. These safe states can be used as targets for cross edges. If no back or cross edge to a safe state can be created even if some given hard limit on the number of states is reached, the transition is completely discarded and we may need to backtrack in order to ensure that there are no states with zero outdegree. Again, this does not affect correctness, as diminishing the set of (infinite) traces preserves ω-language inclusion.

Obviously, each formula $\phi \in \Phi^+$ is satisfied on the resulting Mealy machine *by construction*, whereas each formula in Φ^- is unsatisfied. However, at this point it is unknown whether the remaining properties in $\Phi \setminus (\Phi^+ \cup \Phi^-)$ hold as well, and this cannot be deduced from the construction itself.

The property profile therefore has to be completed by model checking the remaining properties on the model. This is a comparatively quite easy task: whereas generating a Büchi automaton for the conjunct of all formulae in Φ is beyond tractability, an automaton $A_{\neg\phi}$ for each single $\phi \in \Phi$ can be synthesized quite efficiently. Using $A_{\neg\phi}$ to model check the generated Mealy machine is straightforward and can be achieved by standard techniques [3]. We currently use our own implementation, which performs a language emptiness test on the product automaton by analyzing reachability of strongly connected components with accepting states, but of course one could also resort to an external LTL model checker.

6 Mealy Machine Expansion

Once the Mealy machine is constructed from the LTL/Büchi specification, we increase its size (i.e. number of states) artificially while preserving the properties. This is done by iteratively applying the following steps:

– addition of unreachable nodes and model structures,
– splitting nodes with POE according to some randomly set property,
– pruning outgoing transitions.

Adding unreachable stuctures. The first operation simply adds nodes or even arbitrary new Mealy machines over the same alphabet into the original model. To increase the analysis complexity, the states of those newly added model fragments may have transitions into the original Mealy machine. As no transition leads from the original model into the newly added parts, the new nodes remain unreachable and therefore do not alter the original models' behavior.

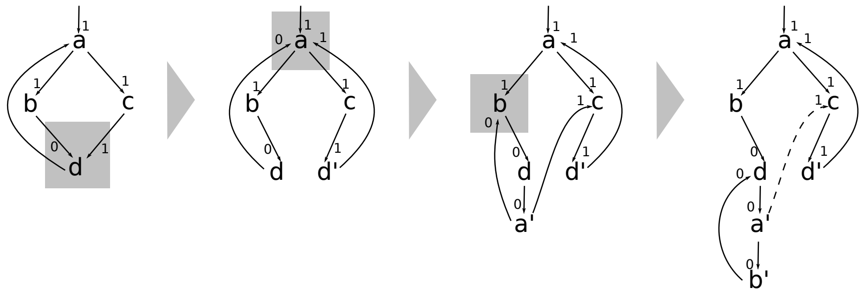


Fig. 3. Property-Oriented Expansion (POE) with random boolean property (0/1)

Splitting under Property-Oriented Expansion. The second operation introduces new reachable states by splitting existing ones. This is done by defining arbitrary new properties and assigning them randomly to every transition. Figure 3 illustrates it using a simple coin toss, i.e., a single boolean property that randomly assumes the values 0 or 1 to each transition. Whenever a node is reached via incoming transitions that have different values for this property, it is duplicated.

Figure 3 shows the effects in detail: the highlighted state d is reached from b with property value 0 and from c with property value 1, causing it to be split into states d and d', connected to the same successor states of the original d (here a) via transitions that inherit the resp. property value. Now a becomes reachable with property values 0 and 1 respectively, so it must be split too. This introduces a new transition from a' to b with a different property value, so finally also b is split. Already from this small example it becomes apparent that randomized property-oriented expansion is a flexible way to significantly increase a model while preserving the set of its traces.

Pruning. These two steps affect only the structure of the model, but not its behavior: the set of traces remains unchanged, and hence a minimization operation would result in the original Mealy machine. This trivially guarantees property preservation, but on the other hand does not truly increase the state space.

A way to overcome this is to prune arbitrary transitions in the intermediate model. Looking at the same example, pruning the dashed transition from a' to c would truly distinguish a' from a, as only in one case there is a transition to c. This transformation is legal (i.e., property preserving) as it only *reduces* the set of all infinite traces in the model, hence it is impossible to introduce unsatisfying paths by pruning outgoing transitions.

7 Mealy-to-Program Model Transformation

This transformation is based on viewing Mealy machines operationally as simple loops of guarded commands, whose guards precisely check for the correct state identification. Its aim is to replace this simple guard structure with a complex, semantically equivalent decision structure in terms of a *discrimination tree* [6], which essentially resembles a complex nested "if-then-else". This is achieved in two steps.

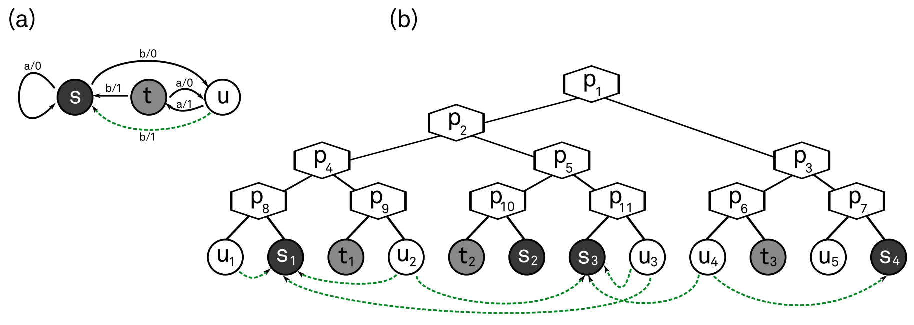
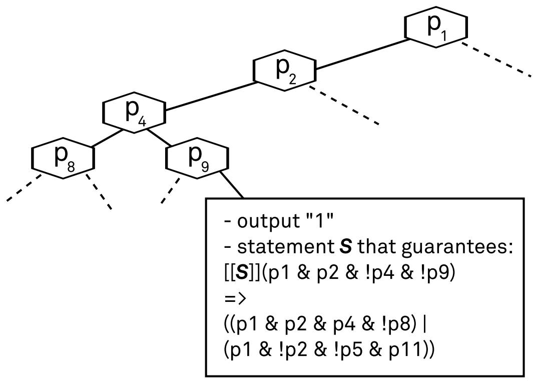


Fig. 4. Discrimination Tree over Mealy machine

In the first step, a discrimination tree is randomly generated both in its branching stucture and its node labeling with predicates. The idea is to represent the states of the Mealy machine by equivalence classes of a randomly constructed partition of the discrimintation tree's leaves. This step is illustrated in Fig. 4 which shows a three state Mealy machine in part (a) and a corresponding discrimination tree in part (b). In this tree, $p_1 \dots p_n$ represent arbitrary predicates, and, e.g., the state s is represented by by the leaves labeled s_1 to s_4.



```
- output "1"
- statement S that guarantees:
[[S]](p1 & p2 & !p4 & !p9)
=>
((p1 & p2 & p4 & !p8) |
(p1 & !p2 & !p5 & p11))
```

Fig. 5. Code model construction

The point of the second step is to maintain property preservation during the transformation step, which essentially requires to establish the correct' wiring' by means of adequate assignments that guarantee that the postcondition of a transition implies the precondition of the target state, and to extract corresponding complex guarded commands. This steps also leaves a lot of room for randomization:

– As there are multiple ways of using the members of the equivalences classes associated to the states, we can first randomly select adequate representations for each transition of the Mealy machine. The dotted lines at the bottom of Fig. 4(b) show the representation of just one such transition: the $b/1$ transition from u to s.
– Fig. 5 sketches an excert of a program model which may have been derived from the discrimintation tree (including the dotted lines) shown in Fig. 4(b). The big box summarizes the condition required at leave u_2 to properly implement the $b/1$ transition from u to s: in response to b, the output 1 needs to be generated and either s_1 or s_3 needs to be reached. In our setting this means that the required reachability constraint must be realized by inserting a statement S that make the correctness assertion/Hoare triple

$$\{p_1 \wedge p_2 \wedge \neg p_4 \wedge \neg p_9\}\mathbf{S}\{(p_1 \wedge p_2 \wedge p_4 \wedge \neg p_8) \vee (p_1 \wedge \neg p_2 \wedge \neg p_5 \wedge p_{11})\}$$

valid [25, 26], or equivalently, that satifies the following implication:

$$[\![S]\!](p_1 \wedge p_2 \wedge \neg p_4 \wedge \neg p_9) \quad \Rightarrow \quad (p_1 \wedge p_2 \wedge p_4 \wedge \neg p_8) \vee (p_1 \wedge \neg p_2 \wedge \neg p_5 \wedge p_{11})$$

The selection of S also leaves plenty of room for random choices.

A deeper discussion of this step, which involves SAT solving/theorem proving [7, 6] is beyond the scope of this paper. Of course, the deeper the analysis the more sophisticated conditional structures can be built. For the RERS Challenge 2012 we first considered a quite simple setting which only uses integers and no arithmetik. As this turned out to be too easy to be truly discriminating, we added arithmetic in the second round. This, from the generation point of view minor change, had a dramatical effect. E.g., it excluded exhaustive symbolic execution, which was still successful in round one. This shows the power of being able to fine-tune the benchmarks' profiles.

8 Elaboration of the Program Model Structure

Program models constructed from Mealy machines and discrimination trees are still quite cycle-oriented and may allow for a reconstruction of the defining Mealy machine, as introduced conditions are locally checkable leaving the transition "wiring" transparent. This can be overcome by applying known global program analysis and transformation techniques:

- Applying randomized POE [4, 27] together with some pruning heuristics allows one to almost arbitrarily "obfuscate" the original cycle orientation, and it provides room for
- a subsequent application of (global) code motion techniques, ranging from merely syntactical analyses reminiscent of partial redundancy elimination [8, 28, 11, 20, 12, 29–31] to more semantic transformations in terms of semantic code motion [10, 9, 32, 33, 21], which shifts the orginal local reasoning problem to the global level.

9 Language-Specific Export

The last step in the benchmark generation process is the process of generating language-specific code. The basis for that is the fully developed abstract code representation that resulted from the previous step. Some of the supported languages are

- Java
- C/C++/C#
- Promela [34]

Ideally the abstract code representation does not contain any real code snippets yet, as this would immediately restrict the generation process to a specific set of programming languages. Even the semantics of simple operations like boolean operators may differ slightly from one language to another. However, as long as the operation semantics are well defined on an abstract level, it is possible to map these operations to concrete language constructs. Adding support for another programming language simply amounts to defining in a template file instantiating patterns for the abstract operations in the target language.

```
if((((((((178 < a12) && (395 >= a12))  &&   ((95 < a23) && (264 >= a23)) )
                && a26) && (a1==2)) && (a19==11)) && a13)){
                throw new IllegalStateException( "error_48" );
}
```

Fig. 6. Java code fragment from problem 13 of RERS 2012

The benefits of template-based code generation were used in the course of the 2012 RERS challenge, where we were able to adapt the problems for the specific needs of some of the contestants' tools in order to attract a larger community. An example is the support of C code along with the specifics of the language. For example, considering the Java code in Figure 6, it makes use of boolean variables and exceptions, both of which are not part of the C language. However, by the C template those constructs are automatically translated to 0/1 integer variables and `assert` statements with corresponding labels, respectively. The resulting code fragment is shown in Figure 7.

```
if((((((((178 < a12) && (395 >= a12))  &&   ((95 < a23) && (264 >= a23)) )
                && (a26==1)) && (a1==2)) && (a19==11)) && (a13==1))){
                error_48: assert(0);
}
```

Fig. 7. C code using integers and `assert`

10 Conclusion and Perspectives

We have presented a systematic approach to automatically generating platform-independent benchmarks of tailored complexity. Key to this approach is a tool chain that essentially transforms a set of automatically generated LTL properties in property-preserving steps into source code for various formats, platforms, and competition scenarios. We have illustrated the whole tool chain, which allows us to address different communites via a growing set of programming languages, tailored sets of programming constructs, different notions of observation, and the full variety of LTL properties, along a accompanying examples.

Central throughout the whole process is the fact that maintaining language inclusion wrt. the ω-language of the Büchi automaton synthesized from the LTL formulae is sufficient to guarantee property preservation. This simple preservation principle has been strictly followed during the whole generation process depicted in Fig. 1, which, in fact, generated very challenging problems for the RERS Challenge 2012, and which will be used with slight extensions for the RERS Challenge 2013. In the future, we plan to release our benchmark generation framework to the public. This will allow research groups across the world to generate benchmarks tailored to their specific needs and profile, and provides a constant source of problems for further development and improvement of their respective tools.

As the preservation principle is not bound to the current "while program"-like structure, it can also be followed when introducing almost arbitrary language and data structure extensions. Those might include, but are not limited to, the procedural abstraction-based construction of methods, pattern-based generation of object structures, or the introduction of further (structured) data types like arrays, lists, `structs`, as well as object-oriented principles like polymorphism and virtual methods. Of course, each of these extensions needs its own reasoning for keeping up to the principle, and therefore introduce their own line of research.

We are currently developing a service-oriented framework for graphically modeling tailored benchmark problems on the basis of a library for property preserving transformations, on top of our service-oriented process modeling framework jABC [35]. Based on this development we envisage to be able to provide a first open version of our benchmark generator early next year. Its flexibility should allow us to address different communities via a growing set of programming languages, tailored sets of programming constructs, different notions of observation, and the full variety of LTL properties, ranging from mere reachability, over general safety properties to arbitrary liveness properties.

References

1. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In Berry, G., Comon, H., Finkel, A., eds.: Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01). Volume 2102 of Lecture Notes in Computer Science., Paris, France, Springer (July 2001) 53–65
2. Giannakopoulou, D., Lerda, F.: From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems. FORTE '02, London, UK, UK, Springer-Verlag (2002) 308–326
3. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2001)
4. Steffen, B.: Unifying models. In Reischuk, R., Morvan, M., eds.: STACS 97. Volume 1200 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 1–20

5. Snelting, G., Robschnik, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Transactions on Software Engineering and Methodology (TOSEM) **15**(4) (2006) 410 – 457
6. Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning Volume I & II. Elsevier (2001)
7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (February 2009)
8. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Comm. ACM **22**(2) (1979) 96–103
9. Steffen, B., Knoop, J.: Finite Constants: Characterizations of a New Decidable Set of Constants. In Kreczmar, A., Mirkowska, G., eds.: Mathematical Foundations of Computer Science (MFCS'89). Volume 379 of LNCS., Springer (1989) 481–491
10. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global Value Numbers and Redundant Computations. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, ACM Press (1988)
11. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proc. of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), ACM (1992) 224–234
12. Briggs, P., Cooper, K.D.: Effective partial redundancy elimination. In: Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl. (PLDI'94). (1994) 159 – 170
13. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems. In Margaria, T., Steffen, B., eds.: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Volume 7609 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 608–614
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. of the 1999 Int. Conf. on Software Engineering, IEEE (1999) 411 – 420
15. Steffen, B.: Property-oriented expansion. In: Static Analysis. (1996) 22–41
16. Hayes-Roth, F.: Rule-Based Systems. Commun. ACM **28**(9) (1985) 921–932
17. McCarthy, D.R., Dayal, U.: The Architecture Of An Active Data Base Management System. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, ACM Press (1989) 215–224
18. Almeida, E.E., Luntz, J.E., Tilbury, D.M.: Event-Condition-Action Systems for Reconfigurable Logic Control. IEEE T. Automation Science and Engineering **4**(2) (2007) 167–181
19. Apt, K.R., Olderog, E.R.: Verification of Sequential and Concurrent Programs. Texts and Monographs in Computer Science. Springer (1991)
20. Knoop, J., Rüthing, O., Steffen, B.: Partial Dead Code Elimination. In: Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), ACM (1994) 147–158
21. Knoop, J., Rüthing, O., Steffen, B.: Expansion-Based Removal of Semantic Partial Redundancies. In: Compiler Construction, 8th International Conference, CC'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings. Volume 1575 of LNCS., Springer (1999) 91–106
22. Mealy, G.H.: A Method for Synthesizing Sequential Circuits. Bell System Technical Journal **34**(5) (1955) 1045–1079
23. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)

24. Kaynar, D.K., Lynch, N.A., Segala, R., Vaandrager, F.W.: Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In: Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), IEEE Computer Society (2003) 166–177
25. Floyd, R.W.: Assigning meaning to programs. In: Proc. of Symposium on Applied Mathematics. Volume 19 of Mathematical aspects of computer science., American Mathematical Society (1967) 19–32
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (1969) 576–580
27. Steffen, B., Rüthing, O.: Quality Engineering: Leveraging Heterogeneous Information - (Invited Talk). In: Proc. of the 12th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011). LNCS (2011) 23–37
28. Dhamdhere, D.M.: A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). Int. J. Comp. Math. **27** (1989) 1–14
29. Knoop, J., Rüthing, O., Steffen, B.: Optimal Code Motion: Theory and Practice. ACM Trans. Program. Lang. Syst. **16**(4) (1994) 1117–1155
30. Knoop, J., Rüthing, O., Steffen, B.: The Power of Assignment Motion. In: Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), ACM (1995)
31. Rüthing, O., Knoop, J., Steffen, B.: Sparse Code Motion. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), ACM (2000) 170–183
32. Steffen, B., Knoop, J., Rüthing, O.: The Value Flow Graph: A Program Representation for Optimal Program Transformations. In Jones, N.D., ed.: Proc. of 3rd European Symposium on Programming (ESOP'90). Volume 432 of LNCS., Springer (1990) 389–405
33. Steffen, B., Knoop, J., Rüthing, O.: Efficient Code Motion and an Adaption to Strength Reduction. In: Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91). Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD). Volume 494 of LNCS., Springer (1991) 394–415
34. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley (2004)
35. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In Bin, E., Ziv, A., Ur, S., eds.: Haifa Verification Conference. Volume 4383 of Lecture Notes in Computer Science., Springer (2006) 92–108