Synthesizing Realistic Verification Tasks

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Marc Jasper

Dortmund

2021

Tag der mündlichen Prüfung:
15.07.2021

Dekan:
Prof. Dr.-Ing. Gernot A. Fink

Gutacher:
Prof. Dr. Bernhard Steffen
Prof. Dr. Stephen F. Siegel

Acknowledgments

First and foremost, I would like to thank my PhD advisor Bernhard Steffen and Stephen F. Siegel very much for reviewing this thesis. I highly appreciate the continuous support and mentoring by Bernhard Steffen, especially during the writing of attached publications. In addition, I would like to thank him and Markus Schordan at Lawrence Livermore National Laboratory for having introduced me to the Rigorous Examination of Reactive Systems (RERS) Challenge and the exciting domain of program verification.

I am very grateful to all my co-authors of scientific publications for the fruitful collaboration. It was a great joy to develop now published ideas together with mentors, collegues, and peers. In the context of this PhD thesis, I would like to especially thank all co-authors of attached publications, particularly Bernhard Steffen, Falk Howar, Stephen F. Siegel, Jaco van de Pol, Alnis Murtovi, Malte Mues, Jeroen Meijer, Maximilian Schlüter, and David Schmidt. Moreover, I appreciate all the submissions by participants of the RERS Challenge who attempted to solve the verification tasks that were synthesized using the framework of this thesis.

My time at TU Dortmund University's Chair of Programming Systems would have not be the same without the friendly and motivating atmosphere among my colleagues. I would like to especially thank my former office mate and good friend Frederik Gossen, Alnis Murtovi, and those colleagues with whom I coordinated the freshman-year math classes at our department of computer science. Furthermore, I am grateful for the support of Maximilian Schlüter and David Schmidt whose work I supervised during the three recent years and who have contributed to the success of this work.

Last but not least, I am very thankful to my parents Annette and Heinrich Jasper, my brother Daniel Jasper, my girlfriend Stefanie Budde, and my friends for their continuous support. Having them in my life means the world to me, and spending time with them further motivated me to always continue pursuing my goals, including the writing of this thesis.

Abstract

This thesis by publications focuses on realistic benchmarks for software verification approaches. Such benchmarks are crucial to an evaluation of verification tools which helps to assess their capabilities and inform potential users. This work provides an overview of the current landscape of verification tool evaluation and compares manual and automatic approaches to benchmark generation. The main contribution of this thesis is a new framework to synthesize realistic verification tasks. This framework allows to generate verification tasks that target sequential or parallel programs.

Starting from a realistic formal specification, a Büchi automaton is synthesized while ensuring realistic hardness characteristics such as the number of computation steps after which errors occur. The resulting automaton is then transformed to a Mealy machine to produce a sequential program in C or Java or to a parallel composition of modal transition systems. A refinement of the latter is encoded in Promela or as a Petri net.

A task that targets such a parallel system requires checking whether or not a given interruptible temporal property is satisfied or whether parallel systems are weakly bisimilar. Temporal properties may include branching-time and linear-time formulas. For the latter, it can be ensured that every parallel component matters during verification.

This thesis contains additional contributions that build on top of attached publications. These are (i) a generalization of interruptibility that covers branching-time properties, (ii) an improved generation of parallel contexts, and (iii) a definition of alphabet extension on a semantic level. Alphabet extensions are a key part for ensuring hardness of generated tasks that target parallel systems.

Benchmarks that were synthesized using the presented framework have been employed in the international Rigorous Examination of Reactive Systems (RERS) Challenge during the last five years. Several international teams attempted to solve the corresponding verification tasks and used ten different tools to verify the newly added parallel programs. Apart from the evaluation of these tools, this endeavor motivated participants of RERS to conceive new formal techniques to verify parallel systems. The result of this thesis thus helps to improve the state of the art of software verification.

Attached Publications

Parts of this dissertation were already published in cooperation with other scientists. These publications with comments on my participation are listed below. Table 1.1 on page 5 further classifies these publications according to their contribution and year of publication.

I Falk Howar, Marc Jasper, Malte Mues, David Schmidt, and Bernhard Steffen.
The RERS challenge: towards controllable and scalable benchmark synthesis.
International Journal on Software Tools for Technology Transfer, 2021.
`https://doi.org/10.1007/s10009-021-00617-z`

Cited as $[\text{HJM}^{+}21]_{\text{AP}}$

The presented ideas were discussed among all authors. I was the main author of Sections 2.5 and 3 and co-authored other sections. The scalability study within Sections 3.3.1 and 3.3.2 was performed by David Schmidt under my supervision.

II Marc Jasper, Maximilian Schlüter, David Schmidt, and Bernhard Steffen.
Every Component Matters: Generating Parallel Verification Benchmarks with Hardness Guarantees.
In Proocedings of the 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA): Tools and Trends. LNCS, vol 12479, pages 241–262. Springer, 2021.
`https://doi.org/10.1007/978-3-030-83723-5_16`

Cited as $[\text{JSSS21}]_{\text{AP}}$

The presented ideas were discussed among all authors. I was the main author of this paper.

III Marc Jasper, Maximilian Schlüter, and Bernhard Steffen.
Characteristic invariants in Hennessy–Milner logic.

Acta Informatica, 57(3):671–687, 2020.
https://doi.org/10.1007/s00236-020-00376-5

Cited as $[JSS20]_{AP}$

The presented ideas were discussed among all authors. I was the main author of this paper. Bernhard Steffen was the main contributor to the introduction and conclusion. Maximilian Schlüter contributed significantly to Section 5 in parallel to his bachelor's thesis which Bernhard Steffen and I supervised.

IV Bernhard Steffen and Marc Jasper.
Generating Hard Benchmark Problems for Weak Bisimulation.
In From Reactive Systems to Cyber-Physical Systems, LNCS, vol 11500, pages 126–145. Springer, 2019.
https://doi.org/10.1007/978-3-030-31514-6_8

Cited as $[SJ19]_{AP}$

The presented ideas were discussed among both authors. I was the main author of Sections 2 to 5.

V Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon Schiffelers, Harco Kuppens, and Frits W. Vaandrager.
RERS 2019: Combining Synthesis with Real-World Models.
In Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol 11429, pages 101–115. Springer, 2019.
https://doi.org/10.1007/978-3-030-17502-3_7

Cited as $[JMM^+19]_{AP}$

I was the main author of Section 3 which contains additional contributions by Alnis Murtovi. Furthermore, I co-authored Section 4.

VI Marc Jasper and Bernhard Steffen.
Synthesizing Subtle Bugs with Known Witnesses.
In ISoLA. LNCS, vol 11245, pages 235–257. Springer, 2018.
https://doi.org/10.1007/978-3-030-03421-4_16

Cited as $[JS18]_{AP}$

The presented ideas were discussed among both authors. I was the main author of this paper.

VII Marc Jasper, Malte Mues, Maximilian Schlüter, Bernhard Steffen, and Falk Howar.
RERS 2018: CTL, LTL, and Reachability.
In ISoLA. LNCS, vol 11245, pages 433–447. Springer, 2018.
https://doi.org/10.1007/978-3-030-03421-4_27

Cited as $[JMS^+18]_{AP}$

The presented ideas were discussed among all authors. I was the main author of Section 3 and co-authored other sections of this paper.

VIII Bernhard Steffen, Marc Jasper, Jeroen Meijer, and Jaco van de Pol.
Property-Preserving Generation of Tailored Benchmark Petri Nets.
In 17th International Conference on Application of Concurrency to System Design (ACSD), pages 1–8, IEEE, 2017.
https://doi.org/10.1109/ACSD.2017.24

Cited as $[\text{SJMvdP17}]_{\text{AP}}$

The presented ideas were discussed among all authors. I was the main author of Sections II to IV.

IX Bernhard Steffen and Marc Jasper.
Property-Preserving Parallel Decomposition.
In Models, Algorithms, Logics and Tools. LNCS, vol 10460, pages 125–145. Springer, 2017.
https://doi.org/10.1007/978-3-319-63121-9_7

Cited as $[\text{SJ17}]_{\text{AP}}$

The presented ideas were discussed among both authors. I was the main author of Section 2 to 7, however all sections are the result of a collaborative effort.

X Marc Jasper, Maximilian Fecke, Bernhard Steffen, Markus Schordan, Jeroen Meijer, Jaco van de Pol, Falk Howar, and Stephen F. Siegel.
The RERS 2017 Challenge and Workshop (Invited Paper).
In Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, SPIN 2017, pages 11–20. ACM, 2017.
https://doi.org/10.1145/3092282.3098206

Cited as $[\text{JFS}^{+}17]_{\text{AP}}$

I was the main author of Sections 2, 3.1, 3.2, and 3.3 of this paper.

XI Maren Geske, Marc Jasper, Bernhard Steffen, Falk Howar, Markus Schordan, and Jaco van de Pol.
RERS 2016: Parallel and Sequential Benchmarks with Focus on LTL Verification.
In ISoLA. LNCS, vol 9953, pages 787–803. Springer, 2016.
https://doi.org/10.1007/978-3-319-47169-3_59

Cited as $[\text{GJS}^{+}16]_{\text{AP}}$

I was the main author of Sections 2.2, 3.3, and 4.2, and contributed to other sections of this paper.

Abbreviations

AE alphabet extension

BA Büchi automaton

CE counterexample

CTL computational tree logic

DFA deterministic finite automaton

GC green contract

HML Hennessy–Milner logic

LTL linear temporal logic

LTS labeled transition system

MC modal contract

MM Mealy machine

MMM modal Mealy machine

MTS modal transition system

NAE nonconvergent alphabet extension

NFA nondeterministic finite automaton

Contents

1 Introduction

Automated verification has seen a number of success stories in the last decades, like the verification of medical device transmission protocols [GCM09], industrial call-processing software [CGP02], or the autonomous behavior of the Curiosity rover [CFL+20]. Nowadays, program verification and other systematic validation techniques are crucial for the development of safety-critical systems [VM05, MWC10].

In order to apply verification in practice, it is vital to know the capabilities, strengths, and weaknesses of existing tools. Moreover, verification tools need to be validated themselves as they may contain implementation errors. Due to a variety of supported programming languages, specification formalisms, and applied techniques, it is quite a challenge to find the best-fitting verification tools for a given applications scenario [Ste17].

Benchmarks have therefore been established as a means to evaluate existing tools in a comparable and sometimes also reproducible fashion. Numerous international verification competitions and challenges attempt to advance the field, classify corresponding tools, and validate the application profile of these tools based on benchmarks [Bey12, KLB+12, HKM15, BFB+17, HIM+14, BdMS05, JLBRS12].

The main goal of this thesis is to create a new framework for the synthesis of realistic verification tasks for such benchmarks that allows to generate both sequential and parallel programs. Key characteristics of the targeted framework are a known solution to synthesized tasks, their scalability, and the fulfillment of specific hardness criteria.

One can distinguish three major approaches to establish a benchmark suite of verification tasks. First, events like the Software Verification Competition (SV-COMP) [Bey12] feature a central, incrementally expanded and consistently maintained benchmark repository that contains for example real-world Linux kernel software, and for which the correct answers to the contained verification tasks is publicly available. Second, property extraction, as for instance used in the Model Checking Contest (MCC) [KLB+12], takes manually selected systems, however (randomly) extracts properties in order to generate a wide range of verification tasks. In some cases, the correct answer to such a task is not known, and majority voting is applied during a corresponding competition. Third, a property-preserving generation of verification tasks based on an initial system that is syn-

thesized from formal specifications—the topic of this thesis—allows for a fully automated yet highly customizable creation of verification tasks. This third approach is employed by the Rigorous Examination of Reactive Systems (RERS) Challenge [HJM+21]$_{\text{AP}}$.

This thesis by publications discusses and presents highlights of the peer-reviewed publications of [HJM+21, JSSS21, JSS20, SJ19, JMM+19, JS18, JMS+18, SJMvdP17, SJ17, JFS+17, GJS+16]$_{\text{AP}}$ that are attached to this work and revolve around the idea to synthesize realistic verification tasks. *Synthesizing* tasks features the advantage that it allows to

- guarantee that the solution to the generated task is known by construction,
- avoid an 'overfitting' of tools and approaches to certain benchmark suites, and
- tailor characteristics of the generated task, including (formal) hardness criteria, to individual needs.

When synthesizing artificial verification tasks, the *realism* of those tasks is especially important: only tasks that feature realistic characteristics can provide insight into whether or not a verification tool is actually capable of solving real-world problems. The synthesis presented in this thesis therefore generates realistic verification tasks that

- are scalable in order to generate tasks that test the limits of state-of-the-art tools,
- feature realistic structural properties that resemble real-world programs, and
- ensure that parameterizable hardness criteria are fulfilled in order to generate not just large, but also hard-to-solve tasks.

Errors in real-world systems can be *subtle*—especially when considering parallel programs—as shown by the August 2003 blackout in the northeastern US that was in part triggered by a race condition and cost several billion US dollars [ZC09].[1] Being able to synthesize verification benchmarks with subtle property violations promotes the development of tools that can detect such subtle yet often severe bugs. This subtlety of incorrect behavior is one example of a criterion that makes a verification task hard to solve. With the goal to generate realistic tasks, this thesis contributes new formal approaches to realize *hard* verification tasks by allowing to adjust the following parameters mostly independently from one another:

- Rarity and depth of violations of linear-time properties, including error reachability.
- Number of individual systems in a parallel composition such that different parallel interleavings induce what is commonly referred to as "state explosion".
- Number of parallel components that matter for analyzing a given property and number of parallel components whose behavior it constrains.

In the following, Section 1.1 states the scientific contribution of this thesis while also embedding attached publications into its larger context. Thereafter, Section 1.2 presents an overview of the chapters in this dissertation.

[1] According to a news article [13], a spokesman of an involved company said: "This fault was so deeply embedded, it took them weeks of poring through millions of lines of code and data to find it."

1.1 Scientific Contributions

The main contribution of this thesis is a new framework for synthesizing realistic verification tasks that advances the state of the art. This contribution consists of the following three major parts.

1. **New approaches to synthesize realistic verification tasks**. These realize
   a) Hard-to-detect counterexamples. Techniques to formally guarantee the hardness of verification tasks by ensuring that counterexamples to linear-time properties are *rare* $[\text{JS18}]_{\text{AP}}$ and *deep* $[\text{HJM}^{+}21]_{\text{AP}}$, the latter in the sense that several computation steps need to be observed in order to be able to identify a property violation.
   b) Extraction of meaningful branching-time properties. A new result on *characteristic invariants* $[\text{JSS20}]_{\text{AP}}$ in Hennessy-Milner logic (HML) [HM80] which states that a labeled transition system can be fully characterized by an invariant in HML. This result is useful for generating verification tasks because it can also be used to characterize abstractions of a system, and thereby allows to *extract* semantically meaningful yet small branching-time properties—even from real-world systems $[\text{JMM}^{+}19]_{\text{AP}}$—in order to create realistic verification tasks.
   c) A unified synthesis centered around modifications of Büchi automata [Bü66] which always creates a modal transition system (MTS) [Lar89]. On the one hand, such an MTS can then be transformed to a Mealy machine [Mea55] that serves as an intermediate specification of a sequential reactive system which is later encoded in C or Java. On the other hand, the constructed MTS can be the basis for generating a hard parallel task by using the following approach.
2. **A new formal method for the generation of parallel verification tasks**, i.e., tasks that target parallel programs. This framework ensures
   a) Property preservation and scalability. Local *parallel decompositions* of MTSs result in an asynchronous parallel system such that *interruptible* temporal properties, i.e., properties whose satisfaction is not influenced by finite interruptions of unobservable actions, are preserved. Key to this preservation is convergent weak modal refinement. Technically, modal contracts, a specific type of assume-guarantee contracts $[\text{BCN}^{+}18]$ that were specifically designed for this decomposition, and alphabet extensions are employed $[\text{SJ17}]_{\text{AP}}$.
   b) Adjustable hardness of the generated verification task. For linear-time properties, it can be guaranteed that *every component matters* when trying to solve the generated verification task $[\text{JSSS21}]_{\text{AP}}$. Key is a propagation of dependencies during local parallel decompositions which is centered around so-called counterexample handles. This new approach also allows to generate tasks where the given property only constrains a local parallel component. Furthermore, it can be adjusted how many parallel components influence the satisfaction or violation of the analyzed property.

c) Applicability to different types of tasks and program models. The generated parallel tasks can be presented in a graph-based DOT format [8], encoded in Promela [Hol11] (see [GJS+16]$_{\text{AP}}$), and transformed to nested-unit Petri nets [Pet81, Gar19] (see [JFS+17]$_{\text{AP}}$). The formal framework ensures preservation of interruptible temporal properties, be they specified in linear temporal logic (LTL) [Pnu77], computational-tree logic [CE81] (see [JMS+18]$_{\text{AP}}$), or the modal μ-calculus [Koz83]. Moreover, the same framework allows to generate hard tasks for weak bisimulation checking [SJ19]$_{\text{AP}}$.

3. **A continuous emission and evaluation** of the above-mentioned contributions through the international Rigorous Examination of Reactive Systems (RERS) Challenge. This endeavor led to

   a) An implementation and automation of both the initial synthesis (see 1.) and framework for generating parallel programs (see 2.). By using the Spot library [DLLF+16] for LTL synthesis and Büchi automaton transformations, systems with hard-to-detect counterexamples can be generated automatically. An extension of the AutomataLib [11] with modal contracts, alphabet extension, and the corresponding decomposition techniques allows to generate parallel verification tasks with adjustable hardness criteria in a push-button approach. I programmed initial versions of the overall framework, designed corresponding algorithms, and supervised their implementation by two student assistants at the Chair of Programming Systems at TU Dortmund University.

   b) A constant evaluation of state-of-the-art verification tools and techniques. The presented framework has been applied to generate verification tasks for the RERS Challenge since 2016. Within the RERS tracks on parallel problems alone, albeit a new addition in 2016 that was steadily expanded, teams from six different institutions from five countries have used (combinations of) 10 different tools in order to solve the available verification tasks. I was responsible for the automatic evaluation of corresponding results by participants and analyzed their submission descriptions when deciding on future steps for the generation of RERS benchmarks.

   c) Fostered collaboration and the invention of new verification techniques. Submissions to the RERS tracks on parallel programs included that of a cross-institutional team of participants whose members are part of national research institutes from different countries. Moreover, generated benchmarks motivated scientific conversations about their characteristics and potential verification techniques between benchmark generators and participants. The new generation framework that this thesis contributes thus impacted other researchers to conceive their own new contributions in the area of program verification, especially new combinations of bisimulations [LMM20, LMM19] and partial-order reduction for action-based systems [SY20].

The three major contributions introduced above have been presented in peer-reviewed

conference and journal publications that have already been published or accepted. Table 1.1 classifies these scientific publications according to both the major contribution of this thesis to which they mostly belong to as well as their year of publication.

Contribution	**2016**	**2017**	**2018**
Synthesizing realistic tasks			[JS18]$_{\text{AP}}$
Key contribution			rare property violations (LTL)
Generating parallel verification tasks		[SJ17, SJMvdP17]$_{\text{AP}}$	
Key contribution		property-preserving parallel decomposition	
Application in the RERS Challenge	[GJS$^+$16]$_{\text{AP}}$	[JFS$^+$17]$_{\text{AP}}$	[JMS$^+$18]$_{\text{AP}}$
Main new technique	abstraction-based property mining	manual use of modal contracts	automated use of modal contracts
Main new features	parallel tasks: LTL, DOT & Promela	hard parallel tasks, Petri nets	CTL properties for parallel tasks

	2019	**2020**	**2021**
Synthesizing realistic tasks		**[JSS20]$_{\text{AP}}$**	**[HJM$^+$21]$_{\text{AP}}$**
Key contribution		generation of HML invariants	deep property violations (LTL)
Generating parallel verification tasks	[SJ19]$_{\text{AP}}$		[JSSS21]$_{\text{AP}}$
Key contribution	hard tasks for weak bisimulation		hard parallel tasks (LTL)
Application in the RERS Challenge	[JMM$^+$19]$_{\text{AP}}$	(see [HJM$^+$21]$_{\text{AP}}$)	
Main new technique	CTL and LTL extraction	Büchi automata-based language reduction	
Main new features	real-world models	deep property violations (LTL)	

Table 1.1: Classification of attached publications based on the major contribution that they belong to and their year of publication. Bold entries mark journal papers.

In addition to what is contained in the attached publications, this thesis contributes the following new, previously unpublished results.

i. A more general definition of *interruptible* properties based on convergent weak modal refinement (Section 4.2). This new definition is shown to be consistent with the previous definition of interruptible linear-time properties [SY20] and character-

izes interruptibility for temporal properties in general, be they specified in linear temporal logic, a branching-time logic, or the modal μ-calculus.

ii. An improved generation of the context component of a modal contract compared to the versions of $[\text{SJ17}]_{\text{AP}}$ (Section 6.1). This improved version is shown to be the coarsest context in case that the given contract is based on a deterministic transition system. Furthermore, this thesis contains proofs of the correctness and coarseness of generated context components.

iii. A revisited, now semantics-based definition of *alphabet extensions*, i.e., specific modal transition systems used to expand the alphabet of a parallel composition (Section 6.2) similar to the ideas of [FBU09]. Fundamental concepts of this new definition are an extension of convergent weak bisimulation to MTSs and parallel composition. This revised definition of alphabet extensions unifies and is consistent with previous descriptions that were either more syntactical/constructive [SJ17, $\text{SJMvdP17}]_{\text{AP}}$ or solely based on language equivalences $[\text{JSSS21}]_{\text{AP}}$.

1.2 Overview

The following Chapter 2 positions the introduced framework for synthesizing realistic verification tasks in the landscape of tool evaluation. That chapter provides an overview of that landscape (Section 2.1), analyzes important characteristics of benchmarks, their realization, and key differences between manual and automatic benchmark creation (Section 2.2), and discusses the contributed synthesis framework and its impact (Section 2.3). Following Chapter 3 that introduces formal preliminaries which are relevant for later chapters, Chapter 4 discusses the types of verification tasks that are generated (Section 4.1), what temporal properties they may feature (Section 4.2), and which hardness characteristics they fulfill (Section 4.3).

Key steps during the synthesis of an initial system for the generated verification task are summarized in Chapter 5. Thereafter, Chapter 6 focuses on new contributions for the generation of parallel programs in addition to attached publications. The two fundamental techniques of property-preserving parallel decomposition (Section 6.1) and alphabet extension (Section 6.2) are explained and extended in a way that both unifies and builds upon previous descriptions in attached publications. Chapter 7 concludes this thesis and presents an outlook to future work.

2

Evaluating Software Verification Tools

Tools for automatic or semi-interactive software verification are nowadays widely used, e.g., in [MWC10, dGRdB+15, VM05, GCM09, CGP02, CFL+20]. A corresponding evaluation of such tools—be it in practice or during research studies—helps to advance the field of software verification. This evaluation to some degree sparked a research area of its own [Bey21, PGG18, SBS18, ADKT11, HIM+14]. In addition, such an evaluation can cover tools that detect errors in software, however not their absence, such as fuzzy testing [SGA07, MFS90] or (active) automata learning [SHM11, IHS14, Ang87]. Constantly and thoroughly evaluating verification tools is important in order to ensure their

- Correctness: Even if a software tool is based on a provably correct program analysis or verification technique, its implementation in some programming language can still be an error-prone process. Especially for tools that should assert the correctness of other software, it is crucial to always produce reliable results.
- Efficiency: For realistic systems, it is usually infeasible to statically analyze every possible behavior explicitly during verification. Several advanced techniques have been conceived to verify the behavior of large systems [KHHH+21], and many are tailored to specific characteristics of the analyzed program [Bie21, CGJ+00, GvLH+96]. It is thus imperative to assess the application profile and scalability of verification tools in order to help potential users to decide which tool is the best choice for their specific goal [Ste17].
- Usability: In order to increase the impact of a successful new verification tool, it is important that this tool can be used effectively not just by the tool developers, but by a larger audience of domain experts who are not necessarily (deeply) familiar with the employed verification techniques [FMB21, HGM20]. The usability of a tool is influenced by many factors, including its installation procedure, the availability of tutorials and exemplary use cases, the organization and understandability of adjustable settings, and the interpretability of the tool's output.

This thesis focuses on the evaluation of the correctness and efficiency of verification tools.

Since at least a decade, verification *benchmarks*—standardized sets of verification tasks—have become increasingly popular as a means to evaluate the state of the art in software verification [Bey12, KLB+12, HIM+14, BFB+17, LLA+17, HKG+12, JLBRS12]. Verification benchmarks are of relevance in multiple domains [OT08, HKP+19, Bey12] and also useful in similar areas like conformance testing [NSVK19]. One can argue that software verification is thereby following the line of other areas in computer science where the use of benchmarks is an established way to measure the performance of hardware or software, for example that of database management systems [NLW+09], (heterogeneous) computing architectures [DMM+10], compiler optimizations [12], and classification techniques in the domain of machine learning [Fis36].

In the following, Section 2.1 gives an overview of the current landscape of tool evaluation in which benchmarks play a central role. Thereafter, Section 2.2 discusses the design and creation of verification benchmarks while comparing manual and automatic approaches. Section 2.3 discusses the new framework to synthesize verification tasks—the main contribution of this thesis—and its impact on the verification community.

2.1 Current Landscape of Tool Evaluation

When evaluating which tool is best suited for a specific category of tasks, it is crucial to ensure that every tool is measured on the same scale: only if results are *comparable*, it is possible to draw meaningful conclusions about which tool is the best choice. In practice, this is typically achieved by confronting different tools with the same data set. For verification tasks, tool performance is measured in the number of correctly solved tasks $[\text{HJM}^+21]_{\text{AP}}$ and sometimes also in terms of required resources such as time or memory consumption [Bey12, KLB+12]. To facilitate this comparability, benchmarks are designed as standardized sets of verification tasks—frequently from a specific category of tasks [HKP+19, LLA+17] or accompanied by a corresponding classification [3].

In addition, tool evaluation is often concerned with the *reproducibility* [10] of results [Bey12, KLB+12]. The kind of reproducibility[1] that current research in the area of software verification mostly focuses on is of a methodological nature [GFI16]: researchers provide an artifact that includes their tool and additional executable scripts if required—sometimes pre-installed in a container [Boe15] or an image of a virtual machine—so that others can rerun the verification on a different machine while comparing the results and monitoring resource consumption [BLW19].

The extend to which the goals of comparability and reproducibility are fulfilled tends to vary, also based on where the evaluation was performed. Tools are commonly evaluated

- within research papers that introduce new tools or approaches,
- in research papers whose goal is to independently compare the state of the art or to introduce a benchmark suite, and
- during international verification competitions and challenges.

[1] See [Ple18] for a clarification of the various meanings of the term “reproducible”.

For research papers on individual techniques, the comparability of results depends on (i) the chosen benchmark and on (ii) whether results of other approaches are just cited or also (re)produced using identical hardware resources. The reproducibility of evaluation results found in publications—no matter if they introduce individual approaches or dedicated comparative studies—depends on the availability of a supplementary evaluation artifact. This reproducibility thereby also indirectly depends on an associated artifact reviewing process which seems to not yet be mature everywhere [HWS20].

Just like research papers that report on a dedicated comparison of tools, competitions and challenges intrinsically ensure the comparability of results by having participants solve the same benchmark, i.e., the same set of verification tasks. Some competitions focus on the reproducibility of results [Bey12, KLB+12] by enforcing resource constraints whereas others omit this requirement in order to allow 'freestyle' solutions.

Whereas there exist plenty of evaluations by tool authors, independent comparative studies exist too [LLA+17, PGG18] but are still rare [Ste17]. International verification competitions and challenges however have become more prominent during the last decade [BBB+19] and typically allow both tool authors and non-authors to participate with a tool (combination) of their choice. These usually annual events provide a platform for

- ranking verification approaches based on their performance while allowing experts to obtain the best possible results with their tools,
- evaluating the improvement not just of an individual tool or approach, but of the entire verification community at regular time steps, and
- exchanging scientific ideas and experiences from attempting to solve the given tasks, both successful and not, with international peers.

Especially the reproduction of tool results requires a certain degree of automation to be scalable. Figure 2.1 presents a qualitative classification of several verification competitions and challenges according to the degree of automation both regarding the competition execution itself and regarding the corresponding benchmark generation. The latter automation will be discussed in Section 2.2. The automation of the competition execution impacts the corresponding submission format and is discussed in the following.

A manual ranking of verification approaches requires the inspection of individual results. This type of ranking is typically used to assess not just the plain result to a verification task, but also the submitted way of solving it. VerifyThis [HKM15] for instance is an annual event where participants solve verification tasks on site. Participants themselves can use semi-interactive tools and, e.g., manually provide loop invariants. Their solution is manually evaluated in whichever way they produce it during the given amount of time.

In contrast, an automatic ranking compares solution vectors and submission vectors. In this scenario, the challenge is to be able to solve the tasks—regardless of how this is achieved. The Rigorous Examination of Reactive Systems (RERS) Challenge $[\text{HJM}^{+}21]_{\text{AP}}$ creates a ranking and awards achievements automatically based on

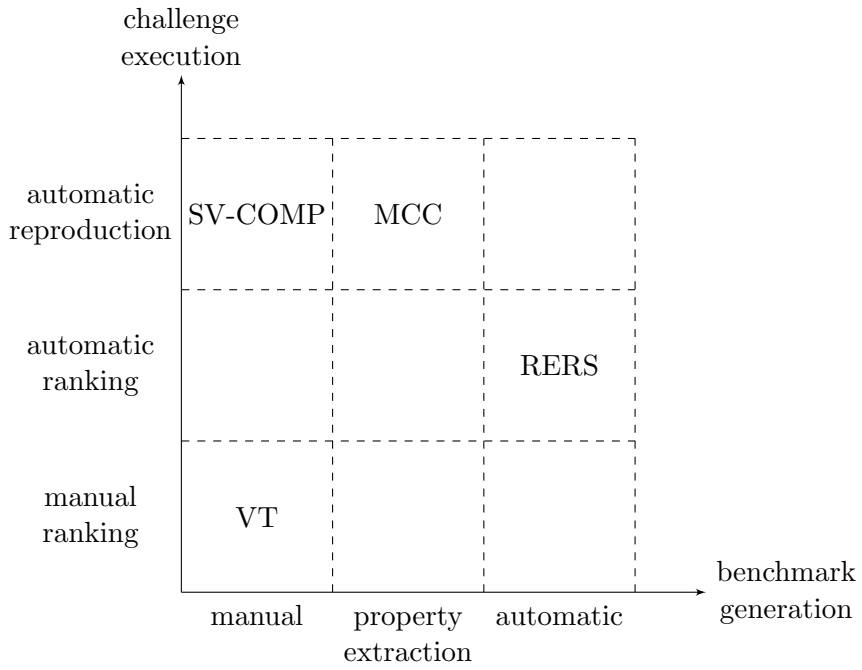

Figure 2.1: Qualitative classification of major competitions and challenges in the area of software verification according to their degree of automation

submissions that are provided by participants. Only the winner of a so-called "Method Combination Award" is selected by a jury.

An automatic reproduction of results is an automated evaluation of executable tools. Instead of submitting result vectors in order to answer given verification tasks, the used tools themselves are submitted. The Competition on Software Verification (SV-COMP) [Bey12] and the Model Checking Contest (MCC) [KLB+12] reproduce the results of participants' tools this way. These submitted tools are run by the competition organizers while also monitoring resource constraints.

The new framework for synthesizing realistic verification tasks that is introduced by this thesis has been applied to create benchmarks for the RERS Challenge during recent years. Within RERS, tasks are usually bundled as 'problems' that consist of a system and a set of properties, i.e., a set of verification tasks that feature the same program. A detailed introduction of RERS can be found in Section 2 of $[\text{HJM}^{+}21]_{\text{AP}}$. Section 2.1 in that publication features a detailed comparison of RERS to other major competitions in the realm of software verification, namely MCC, SV-COMP, and VT.

2.2 Benchmark Design and Creation

Benchmarks are a key component of contemporary evaluations of software verification tools. The following Section 2.2.1 discusses useful characteristics of such benchmarks, be-

fore Section 2.2.2 presents ways to realize these characteristics. Thereafter, Section 2.2.3 compares manual and automatic approaches to create verification benchmarks.

2.2.1 Useful Characteristics of Benchmarks

The main goal of a verification benchmark is to provide a way to evaluate how well verification tools can be applied in a practical scenario. A useful benchmark of verification tasks should therefore

1. feature known solutions,
2. contain realistic programs and properties, and
3. include a wide variety of tasks and/or tasks of different scale and hardness.

Verification tasks can be distinguished based on the availability of its solutions: a solution can be unknown, known (in general), or known to the public. Known solutions—regardless of whether this knowledge is public or kept secret by evaluators—ensure that tools can be judged fair and without bias. This is based on the reason that if a solution is instead unknown, measures like majority voting among tools have to be taken [KHHH+21].

A solution that is known, however not publicly available, can avoid that tools 'overfit' to a benchmark by, e.g., hard-coding specific loop bounds until which all errors can be found within the given benchmark. It further enables the possibility for an unmonitored competition where participants can combine tools and approaches to their liking.

Realism of verification tasks is significant when drawing conclusions from an evaluation based on a corresponding benchmark.

A diversity of contained tasks—be it by covering a wide range of different tasks or by featuring different scales and hardness levels of similar tasks—further helps to identify application profiles and test the limits of tools.

A collection of available academic examples, such as the dining philosophers problem for deadlock detection, rarely suffices to constitute a useful benchmark. Solutions to such examples are known by construction, however such problems typically resemble "toy examples" that are either hardly scalable or lack realism. Finding available programs with documented correctness properties, maybe even including a proof of why they are satisfied or violated, can be difficult, likely also because many programs are proprietary. Especially when it comes to more formal specifications such as linear-time or branching-time properties, available real-world examples can be rare. Carefully designing useful benchmarks is thus an important part of evaluating verification tools.

2.2.2 Realization of Useful Benchmark Characteristics

Several verification benchmarks exist and one can distinguish between different degrees of automation within their creation process. Whereas many benchmarks are (i) designed manually, some take a step towards an automatic generation by (ii) extracting properties from given systems. In addition—and the primary focus of this thesis—benchmarks can be (iii) generated fully automatically by using synthesis based on a given specification.

When discussing a manual creation of verification benchmarks, this thesis means a manual collection and maintenance of contained tasks. Individual verification tasks can be manually designed, e.g., by experts who can formally assert their correct solution. Manually created benchmarks however often contain several existing programs from industry or academia, including a few synthesized ones. An automatic generation of benchmarks is however tightly coupled with an automatic synthesis of contained tasks. For this reason, the terms automatic synthesis of verification tasks and automatic benchmark generation are used interchangeably in the following.

The horizontal axis of Figure 2.1 classifies verification competitions and challenges according to their employed approach to benchmark generation, more specifically regarding the automation of the latter. The classification in Figure 2.1 is always based on the "most automated" part of the used benchmark generation. Some competitions and challenges feature several automation levels: MCC for example uses both pre-defined properties and extracted properties [KHHH+21], and RERS also featured benchmarks based on property extraction.

In the following, details are given on how different approaches to benchmark generation—with different degrees of automation—realize the three useful characteristics of verification benchmarks that are described in Section 2.2.1.

Manually maintained verification benchmarks such as that of SV-COMP provide known solutions mostly based on the sources that tasks were collected from. Such a competition frequently includes (modified versions of) real-world programs along with corresponding properties. Examples include open-source code from Linux driver kernels together with reachable or unreachable violated assertions [Bey12]. The MCC also contains manually collected tasks, and examples of realistic ones include a Petri net that models the mass memory management system in a micro-satellite [2]. Different levels of hardness can be achieved by providing a diverse set of programs. To some extent, collecting tasks for such a repository and maintaining it can further serve as a classification of common errors in programs and ways to detect them, a goal that is intensely studied in the somewhat related area of computer security [GMBG20, BBYW15].

In addition to a collection of existing tasks from academia or industry, manually created verification benchmarks can contain tasks that were manually designed by domain experts. For data race detection, this is realized in [LLA+17] where dozens of artificial, mostly small example programs are designed to cover a wide range of typical patterns that cause data races in real-world applications. In that work, some of the contained data race detection tasks feature an adjustable hardness that is realized via variable array sizes. Verification tasks of VerifyThis are also designed manually, however organizers gather ideas for future tasks from the community [DFH+21].

A step towards automatic benchmark generation is the extraction or mining of properties on chosen, potentially real-world systems. This is done in MCC [KHHH+21] for temporal properties and certain upper bounds. Furthermore, property extraction was used for some benchmarks of the RERS Challenge [GJS+16, JMM+19]$_{\text{AP}}$. A solution to tasks with extracted properties is known either based on choosing specific abstractions that participants are unaware of [Jas18] or because an available and trusted verification

tool can assess it [KGH+18]. Some solutions remain unknown [KHHH+21]. Realism is centered around the models from which properties are extracted, and the extraction of realistic properties still requires additional effort [KGH+18]. Due to the random generation of properties, verification tasks cover a wide variety of specifications, their hardness however is only guaranteed in a very limited fashion by filtering out trivial properties [KGH+18].

Synthesizing verification tasks—and thereby benchmarks—fully automatically, as predominantly applied in the RERS Challenge [HJM+21]$_{\text{AP}}$, is in some way a dual approach to property extraction. Realistic properties, based on common patterns [DAC99] or extracted from real-world models [JMM+19]$_{\text{AP}}$, serve as a specification for the synthesis of a task. In the case of RERS, Büchi automata are synthesized from linear temporal logic specifications before they are enlarged and transformed into programs in a property-preserving fashion [HIM+14]. As a result, solutions to the generated tasks are known, however initially not to the public, and providers of the benchmark can decide when to make them publicly available. Realism of synthesized tasks is ensured within a certain domain of programs. Sequential programs within RERS benchmarks closely resemble programmable logic controllers (PLC) [Eri96], a type of program widely used in industry [AA16] and for which verification approaches have been intensely studied [OAPÜ16, Moo94, RK98, GSF06, JS16]. Different hardness levels of verification tasks are realized both through (i) formal hardness guarantees such as the depth[2] of occurring property violations or the relevance of parallel components, and (ii) a scalable generation based on local transformations.

2.2.3 Manual vs. Automatic Benchmark Creation

The manual creation of verification benchmarks and their automatic generation are quite different approaches. Table 2.1 presents major advantages and disadvantages that can be associated with manual and automatic ways to create these benchmarks. For each of the listed disadvantages, Table 2.1 further mentions feasible ways to mitigate respective drawbacks.

When comparing manual and automatic benchmark creation according to Table 2.1, the availability of solutions turns out to be one major difference between the two approaches. Even though VerifyThis features new manually designed benchmarks that consist of a handful of verification tasks each year, such a complete and regular redesign is impracticable for large benchmarks. As a consequence, solutions to many tasks within manually designed benchmarks—if they are known in general—are also known to those people who intend to solve these tasks again using a specific tool [Bey12, LLA+17, KLB+12]. Verification competitions such as SV-COMP or MCC therefore require a centrally monitored execution of tools. In contrast, the solution to automatically generated verification tasks like those of RERS can be kept secret, allowing participants flexibility regarding their approach to solve these tasks. Verification tasks of RERS are used in SV-COMP and MCC, however such a reuse would thus not be feasible in the opposite direction.

[2]Here, depth means the number of computation steps after which a property violation can be detected.

	manual creation	automatic generation
pros	+ realism of existing programs + categorization of real-world problems + diverse tasks from various sources	+ effortless generation of tasks with publicly unknown solutions + highly parameterizable and scalable + comparable tasks in various programming (sub)languages
cons	− effort to collect and maintain tasks − correct answer typically known to the public − not trivial to adjust hardness	− generating realistic tasks requires care − hard to imitate manually written code − type of program limited by the used formal framework
mitigated by	∘ reducing individual effort through crowd-sourcing ∘ centralized monitoring of tools ∘ including parameterizable or auto-generated tasks	∘ using real-world models or structural properties extracted from them ∘ the relevance of requirement-driven design and real-world use of compilation ∘ extending generation framework to new domains

Table 2.1: Major advantages, disadvantages, and mitigations of the latter associated with the manual creation and automatic generation of verification benchmarks

Furthermore, Table 2.1 states that manual and automatic approaches can show different aspects of realistic software verification. On the level of manually written code, the realism of, e.g., the Linux kernels found in the SV-COMP benchmark is unmatched by that of generated programs. This downside of automatically generated tasks is mitigated by the fact that real-world code such as that found in PLC controllers is often auto-generated too [Sac05, EMIO07], [7]. Still, even within this domain, guaranteeing a realistic hardness of the generated task requires care and additional formal approaches such as the ones introduced in this thesis. Aside from the programs themselves, one can argue that automatically generated benchmarks can be used for a more realistic simulation of verification in practice because solutions to open verification problems are in fact not publicly known.

Another characteristic where manual and automatic benchmark creation differ is the diversity of contained tasks. Benchmarks like the one used in SV-COMP feature code from several sources and thereby reflect different application scenarios of verification tools. In contrast, automatically synthesized benchmarks have a certain common structure imbued into them based on the formal framework that was used during their generation, such as the Mealy machine model used for sequential programs of RERS. When it comes to a group of programs from a specific domain, automatic generation enables diversity by tailoring tasks to specific requirements [HIM+14]. Parameters include the size of code, programming features that are used, different hardness criteria such as the required exploration depth $[\text{HJM}^+21]_{\text{AP}}$, and even the programming language itself. From an abstract point of view, one can argue that manually designed benchmarks tend to be better at evaluating a single tool based on diverse scenarios. In contrast, synthesized benchmarks can have an edge when it comes to (i) comparing as many tools as possible based on the same task by supporting multiple programming languages and (ii)

measure tool performance based on fine-grained variations of the size and hardness of tasks.

Based on Table 2.1 and its above discussion, it becomes clear that manual and automatic benchmark creation have very different characteristics. The creation of verification benchmarks is clearly unlike other areas where automation allowed to replace manual efforts: it is not the goal of the presented automatic benchmark synthesis to replace the collection of real-world tasks and the monitoring of this process by scientists. Instead, it is a different approach to the problem of benchmark design and creation, one that justifies its relevance based on (i) the merits of providing new tasks whose solution can be kept secret, (ii) the use of requirement-driven development in real life $[\text{JSSS21}]_{\text{AP}}$, and (iii) the lack of sufficient—and sufficiently adjustable—verification tasks that are publicly available. Considering these very different profiles, it seems likely that both approaches, the manual creation of verification benchmarks as well as their automatic synthesis based on realistic specifications, will continue to coexist.

2.3 New Synthesis Framework and its Impact

The new framework to synthesize realistic verification tasks that this thesis contributes is discussed in this section, along with its impact on the verification community. First, Section 2.3.1 shows how the benchmark generation framework of RERS was extended by the work of this thesis and presents achieved hardness criteria. Section 2.3.2 summarizes the impact of the synthesized tasks on the verification community as a whole, before Section 2.3.3 details benefits for participants of the RERS Challenge.

2.3.1 New Framework for Benchmark Synthesis

This thesis contributes a new framework for the automatic synthesis of verification benchmarks that is used in the RERS Challenge. Figure 2.2 illustrates this new framework and its modular, extensible structure.

Starting from a pattern for linear temporal logic (LTL) properties, some formulas from that pattern are selected and the presented framework synthesizes a corresponding Büchi automaton (BA). For a verification task with a verifiable property, a regular synthesis suffices. For a task with a refutable property, the new framework allows to modify a BA in a way such that realistic hardness characteristics of the generated task are ensured, namely the rarity or depth of counterexamples to linear-time properties. This is an addition to the generation framework that was previously used for RERS: before the work of this thesis, violated LTL properties were extracted using property mining (see Section 2.2.2), a procedure that lacks control over specific counterexamples. The new framework always transforms the resulting BA to a modal transition system (MTS).

Such an MTS serves as a unifying intermediate representation: both the generation of sequential programs and that of parallel programs are based on this behavioral specification within the new framework $[\text{JS18}]_{\text{AP}}$. Furthermore, this representation can be used to extract computational tree logic (CTL) properties either based on patterns that

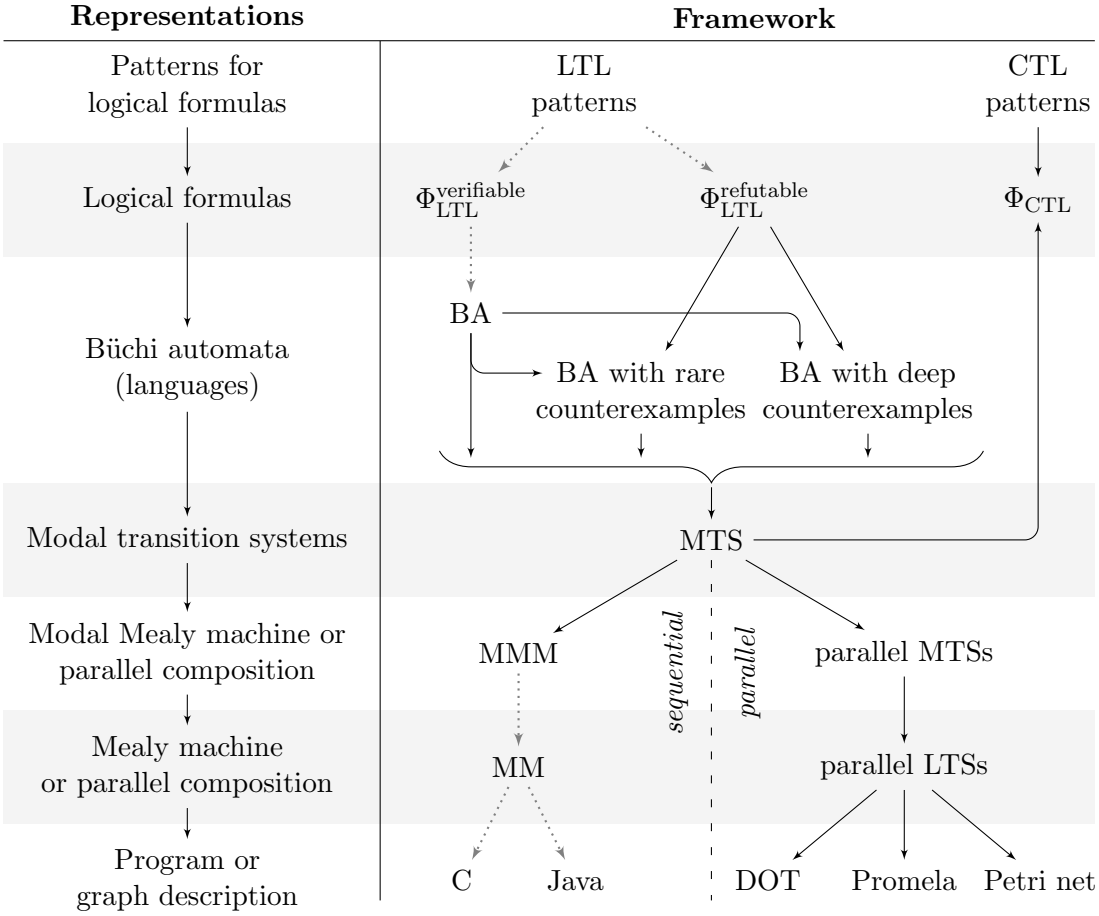

Figure 2.2: Overview of the new framework for synthesizing verification tasks. Solid arrows indicate new contributions and dotted arrows represent previous work.

are syntactically similar to the used LTL formulas [JMS+18]$_{\text{AP}}$ or according to the more semantics-driven ideas centered around weakened characteristic invariants [JSS20]$_{\text{AP}}$.

When synthesizing sequential programs, the MTS is transformed to a modal Mealy machine (MMM) that specifies input-output behavior of a reactive system. The existing modalities allow flexibility w.r.t. a refining (non-modal) Mealy machine (MM), and this chosen MM is encoded as a C or Java program [HIM+14].

In order to generate parallel programs, the development of the new framework involved the conception of specific assume-guarantee contracts [SJ17, SJMvdP17]$_{\text{AP}}$ for parallel decomposition (Section 6.1). In addition to the synthesis of verification tasks, a modification allows to reuse this parallel decomposition to generate hard tasks for weak bisimulation checking [SJ19]$_{\text{AP}}$. As presented in [JFS+17]$_{\text{AP}}$, the communication pattern of a generated parallel system can be obfuscated drastically through an additional post-

processing based on partial evaluations of the given parallel composition. Modalities again allow flexibility w.r.t. a final implementation, and an LTS component refinement chooses such a final model. This final model is either represented by a DOT graph [8], encoded in Promela $[\text{GJS}^+16]_{\text{AP}}$, or transformed to a Petri net $[\text{JFS}^+17, \text{SJMvdP17}]_{\text{AP}}$.

Key characteristics of this new framework are illustrated in Figure 2.3.

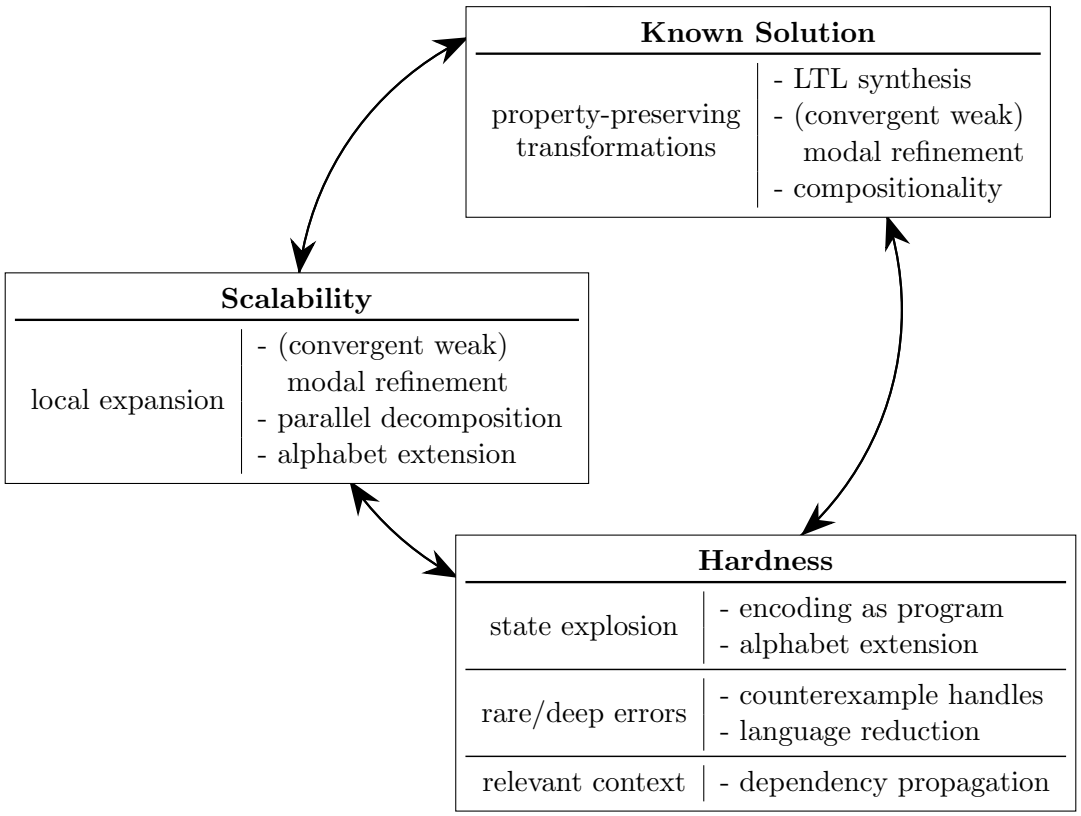

Figure 2.3: Key characteristics of the new framework for synthesizing verification tasks. For each characteristic (bold text), the text below briefly states which concepts (left side) and techniques (right side) are used to achieve it.

A known solution to synthesized tasks is guaranteed on the basis of property-preserving transformations. Whereas the LTL synthesis step is explained in Chapter 5, (convergent weak) modal refinement is presented in Section 3.4. Compositionality ensures that local transformations preserve properties at a global scope, as shown in Section 4.2.

Scalability is accomplished based on local expansions. The mentioned modal refinement relations allow to check if an expanding transformation is still property-preserving. Parallel decomposition (Section 6.1) is used to 'split' individual parallel components and thereby locally expand a parallel system. Alphabet extensions (Section 6.2) allow to expand the size of a parallel component through new actions.

Directly linked to scalability is the supported hardness criteria of state explosion—meaning that the program which needs to be verified has an exponential number of

reachable program states compared to its description. Section 4.3.1 compares encoding-based and interleaving-based ways to achieve state explosion.

Hardness criteria w.r.t. violated linear-time properties are the rarity of counterexamples or their depth, meaning the number of computation steps after which an error can be detected. The former is realized using so-called counterexample handles [JS18]$_{\text{AP}}$ (Sections 4.3.2 and 5.2), whereas the latter utilizes intersections of Büchi automata to reduce a given language [HJM$^+$21]$_{\text{AP}}$ (Section 5.3).

When considering parallel programs, the relevance of the generated parallel context (Section 4.3.3) can be ensured based on dependency propagation such that every parallel component matters during verification [JSSS21]$_{\text{AP}}$.

2.3.2 Impact on the Verification Community

Using the presented synthesis framework during each of the recent iterations of RERS impacted the verification community because it allowed to annually evaluate state-of-the-art verification tools [9]. RERS enables not just a ranking of participants according to their capability of correctly solving provided tasks, but also a further analysis of submissions. Section 2 of [JMM$^+$19]$_{\text{AP}}$ for instance identifies that participating tools in the sequential tracks of RERS 2018 were more successful at identifying refutable linear-time properties than verifiable ones. Other competitions such as MCC demonstrate that such an evaluation can be taken a step further by statistically analyzing their data set of answers to verification tasks with the goal to identify hard-to-verify properties and successful analysis techniques [KHHH$^+$21].

Some verification tasks of RERS have been adopted by SV-COMP since 2014 [4] and the Sequence PredIction ChallengE (SPiCe) [BEL$^+$17] in 2016. Verification tasks of RERS 2017 that feature parallel systems [JFS$^+$17]$_{\text{AP}}$—tasks that were generated based on the introduced framework—are also part of the MCC benchmark since 2018 [5].

Participants of RERS' tracks that feature parallel programs conceived new verification techniques whose discovery was motivated by their participation. Major new techniques are a combination of bisimulations [LMM20, LMM19] and partial-order reduction for state-based systems [SY20]. As presented in Section 2.6 of [HJM$^+$21]$_{\text{AP}}$, RERS has influenced many other scientific contributions and motivated the combination of verification methods.

2.3.3 Benefits for Participants of RERS

For individual participants, the benchmarks of RERS and a submission to the challenge can have several benefits. First, the provided benchmarks serve as convenient test cases for tool developers. Especially when participating in the tracks on linear temporal logic properties that have to hold during every feasible program execution, a correct answer can serve the role of a 'checksum' for the correct implementation of a verification tool. RERS benchmarks are therefore used as part of regression tests: CodeThorn [JS16], a tool based on the ROSE compiler infrastructure [Qui00], has integrated the verification of

RERS benchmarks into its build pipeline [6]. This regularly tests not only the employed model checking library, but also parts of the compiler frontend used for C programs.

Second, obtained scores and earned achievements[3] provide participants with feedback on how well their approach performed. Individual result vectors indicate what worked and which tasks require additional effort in the future. As the results of each iteration of RERS are available online [9], a successful participation can further serve as positive publicity for the corresponding tool, backed by the credibility of a long-running international verification event. Moreover, RERS motivated a special section of peer-reviewed journal papers that allowed participants to report on their experiences [HIM+14]. Nowadays, RERS takes steps in a direction similar to SV-COMP regarding an option for regular submission of tool papers [1]. RERS thus supports tool developers by facilitating the discovery of successful approaches and research prototypes.

Third, RERS features a forum for the exchange of results and ideas: most participants of RERS attend the corresponding annual event. During this event, ranking and achievement results are announced, approaches are presented during scientific talks by both participants and benchmark generators, and future directions of the challenge are discussed among everyone involved in that year's challenge.

[3] See Section 2.5 in [HJM+21]$_{\text{AP}}$ for a detailed description of the reward structure of RERS.

3 Preliminaries

Important definitions and propositions that will be used in later chapters are introduced in the following. In Section 3.1, the formal details of words and languages are stated as used within this thesis. Section 3.2 presents the formal models that are used as intermediate representations to generate verification tasks. Thereafter, Section 3.3 introduces action-based linear temporal logic and corresponding model checking, before Section 3.4 presents important notions and basic results w.r.t. property-preserving transformations.

3.1 Languages

This thesis focuses on linear-time properties that reason about words—mostly as label sequences in transition systems—and about corresponding languages $[\text{HJM}^+21]_{\text{AP}}$.

Definition 1 (Words)**.** *Given a finite alphabet Σ, a* word *over Σ is a (possibly empty or infinite) sequence of symbols from Σ. The alphabet of symbols that occur in w can be accessed as $\Sigma(w)$. Given an integer $n \in \mathbb{N}$ and a finite word $w = a_1 a_2 \dots a_n$, $|w|$ denotes the* length *n of w. An infinite word is a function $w : \mathbb{N} \mapsto \Sigma$ such that $w_i := w(i)$ is the i-th symbol of this word and has the length $|w| = \infty$. Given any word $w = a_1 a_2 \dots$ and any integer $i \in \mathbb{N}$ such that $i \leq |w|$, $w_{\leq i}$ denotes the* prefix *of w of length i. Similarly, $w_{\geq i} = a_i a_{i+1} \dots$ denotes the suffix of w starting at index i.*

Definition 2 (Languages)**.** *Given a finite alphabet Σ, a language (over Σ) is a set of words over Σ. For a given $n \in \mathbb{N}$, the language Σ^n consists of all words $w = a_1 a_2, \dots a_n$ of length $|w| = n$ such that $a_i \in \Sigma$ for all $i \in 1 \,..\, n$. For any $n \in \mathbb{N}$, $\Sigma^{\leq n} := \bigcup_{i=1}^{n} \Sigma^i$, and additionally $\Sigma^* := \bigcup_{i \in \mathbb{N}} \Sigma^i$. A language $\mathcal{L}$ is finite iff $|\mathcal{L}| \in \mathbb{N}$ and infinite otherwise. Σ^ω denotes the language that contains all infinite words over Σ.*

Moreover, $\mathcal{L}$ is a language of finite words iff $\mathcal{L} \subseteq \Sigma^*$, and a language of infinite words iff $\mathcal{L} \subseteq \Sigma^\omega$. The set of prefixes of a language $\mathcal{L}$ with length at most k is denoted by $\mathcal{L}_{\leq k} := \{w_{\leq i} \mid w \in \mathcal{L} \ \wedge \ i \leq k\}$. The concatenation of words extends naturally to languages: given a language $\mathcal{L} \subseteq \Sigma^*$ and any language $\mathcal{L}'$, their concatenation is defined as $\mathcal{L}\mathcal{L}' := \{ww' \mid w \in \mathcal{L} \wedge w' \in \mathcal{L}'\}$. For any word w and any language $\mathcal{L}$, $w\mathcal{L} := \{w\}\mathcal{L}$.

A later chapter further employs traditional concepts based on regular languages, specifically non-deterministic finite automata (NFAs) and deterministic finite automata (DFAs). It is assumed that the reader is familiar with those concepts and corresponding transformations.

3.2 Models of a System

This section covers the program models that verification tasks within this thesis are based on. Section 3.2.1 describes modal transition systems which serve as an intermediate representation for parallel program specifications within this thesis. Their parallel composition is explained in Section 3.2.2. Section 3.2.3 introduces Mealy machines which are used to generate sequential programs.

3.2.1 Modal Transition Systems

Modal transition systems [Lar89] allow to distinguish between behavior that must be feasible and one that may be feasible. The following definitions are taken from $[\text{JSSS21}]_{\text{AP}}$ with only minor adjustments.

Definition 3 (Modal Transition System)**.** *Let S be a set of states and Σ a finite alphabet of action symbols. $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ is called a* (rooted) modal transition system (MTS) *with root $s_0 \in S$ iff the following condition holds:*

$$\longrightarrow \subseteq \dashrightarrow \subseteq (S \times \Sigma \times S)$$

Elements of $\dashrightarrow$ are called may transitions *and those of $\longrightarrow$* must transitions*. The syntax $s \overset{a}{\dashrightarrow} s'$ and $s \overset{a}{\longrightarrow} s'$ is used to denote transitions $(s, a, s') \in \dashrightarrow$ and $(s, a, s') \in \longrightarrow$, respectively. Furthermore, the operator $\Sigma(M) := \Sigma$ accesses the alphabet of M and is overloaded to access labels of transitions, i.e., $\Sigma(s \overset{a}{\dashrightarrow} s') := a$ for any $s \overset{a}{\dashrightarrow} s'$ and $\Sigma(T) := \bigcup_{t \in T} \{\Sigma(t)\}$ for any $T \subseteq \dashrightarrow$.*[1]

Throughout this thesis, two MTSs are presumed to be identical if they are isomorphic.

Definition 4 (Path, Reachability, and Access Sequences)**.** *Let $(S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS. Then a* path *is a sequence $\pi = s_0 \overset{a_1}{\dashrightarrow} s_1 \overset{a_2}{\dashrightarrow} s_2 \cdots$ of transitions that starts in s_0 with i ranging from 0 to either a positive integer or infinity. Set notation is used to refer to the transitions of a path, e.g. $s_0 \overset{a_1}{\dashrightarrow} s_1 \in \pi$. The word w induced by the label sequence $a_1 a_2 \ldots$ is denoted as $w(\pi)$.*

A state $s \in S$ is reachable *iff a path exists that ends in s. For any reachable state $s \in S$, the label sequences of words that lead to s are defined as* $\text{access}(s, M) := \{w(\pi) \mid \text{there exists a path } \pi \text{ in } M \text{ that ends in } s\}$.

[1] Note that not every alphabet symbol has to occur as a transition label, meaning that $\Sigma(M) = \Sigma(\dashrightarrow)$ is not guaranteed in general.

An MTS can be seen as an extension of a traditional (rooted) labeled transition system (LTS), which allows the following definition:

Definition 5 (Labeled Transition Systems)**.** *A* labeled transition system *(LTS) is an MTS* $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ *with* $\dashrightarrow = \longrightarrow$*. Components four and five are thus joined in case of LTSs.*

For the model checking of linear-time properties, the minimal and maximal languages defined by an MTS M are important, which are based on the must and may LTS of M, respectively:

Definition 6 (Must and May LTS)**.** *Given an MTS* $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$*, the LTSs* $L_\bot(M) := (S, s_0, \Sigma, \longrightarrow)$ *and* $L_\top(M) := (S, s_0, \Sigma, \dashrightarrow)$ *are called* must and may LTS *of* M*, respectively.*

Definition 7 (Minimal and Maximal Language)**.** *The* language $\mathcal{L}(L)$ *of words in an LTS* L *equals the label sequences of all paths in* L*. Infinite words in a language* $\mathcal{L}$ *are denoted by* $\mathcal{L}^\omega$*. Given an MTS* M*,*

1. $\mathcal{L}_\bot(M) := \mathcal{L}(L_\bot(M))$ *is called the* minimal language *and*
2. $\mathcal{L}_\top(M) := \mathcal{L}(L_\top(M))$ *the* maximal language

of M*, respectively. This definition propagates to subsets of infinite words.*

Obviously, it holds that $\mathcal{L}_\bot(M) \subseteq \mathcal{L}_\top(M)$ because of $\longrightarrow \subseteq \dashrightarrow$.

3.2.2 Parallel Composition

The parallel composition operator for MTSs used within this thesis, as presented in $[\text{SJ17}]_{\text{AP}}$ and, e.g., also used in $[\text{JSSS21}]_{\text{AP}}$, is reminiscent of CSP [Hoa78] with synchronization of components on their common alphabets:

Definition 8 (Parallel MTS Composition)**.** *Given MTSs* $M_1 = (S_1, s_0^1, \Sigma_1, \dashrightarrow_1, \longrightarrow_1)$ *and* $M_2 = (S_2, s_0^2, \Sigma_2, \dashrightarrow_2, \longrightarrow_2)$*, the* parallel composition

$$M_1 \,||\, M_2 := (S_1 \times S_2, (s_0^1, s_0^2), \Sigma_1 \cup \Sigma_2, \dashrightarrow, \longrightarrow)$$

is defined as a commutative and associative operation satisfying the following operational rules where $\rightharpoonup$ *identifies the type of transition and is once instantiated to represent may transitions and once to represent must transitions:*[2]

$$\frac{p \overset{a}{\rightharpoonup}_1 p' \quad q \overset{a}{\rightharpoonup}_2 q'}{(p,q) \overset{a}{\rightharpoonup} (p',q')} \qquad \frac{p \overset{a}{\rightharpoonup}_1 p' \quad a \notin \Sigma_2}{(p,q) \overset{a}{\rightharpoonup} (p',q)}$$

It follows directly that parallel composition can only reduce the minimal and maximal languages of an MTS if a shared communication alphabet exists $[\text{JSSS21}]_{\text{AP}}$.

[2] Please note that every must transition is also a may transition.

Proposition 1 (Orthogonal Composition)**.** *Let M, M' be MTSs with $\Sigma(M) \cap \Sigma(M') = \emptyset$. Then both $\mathcal{L}_\bot(M) \subseteq \mathcal{L}_\bot(M \mid\mid M')$ and $\mathcal{L}_\top(M) \subseteq \mathcal{L}_\top(M \mid\mid M')$ hold.*

For any alphabet Σ, there exists a simple LTS that always has an outgoing transition for every alphabet symbol $a \in \Sigma$ and is therefore neutral w.r.t. parallel composition.

Definition 9 (Neutral Element of Composition)**.** *Let Σ be an alphabet. The one-state LTS $\mathcal{N}_\Sigma$ with transitions for every $a \in \Sigma$ is called the* composition-neutral Σ-LTS*:*

$$\mathcal{N}_\Sigma := (\{s\}, s, \Sigma, \{s\} \times \Sigma \times \{s\})$$

Proposition 2 (Neutrality)**.** *Let M be an MTS. Then $M \mid\mid \mathcal{N}_\Gamma = M$ holds for any $\Gamma \subseteq \Sigma(M)$.*

3.2.3 Mealy Machines

Mealy machines [Mea55] are the formal specification based upon which sequential programs are generated within this thesis.

Definition 10 (Mealy Machine)**.** *A Mealy machine (MM) is a tuple $M = (S, s_0, \Sigma, \Lambda, \delta, \lambda)$ where*

1. *S is a finite set of states with initial state $s_0 \in S$,*
2. *Σ, Λ are finite input and output alphabets, respectively, and*
3. *$\delta : S \times \Sigma \rightharpoonup S, \lambda : S \times \Sigma \rightharpoonup \Lambda$ are partial transition and output functions, respectively, such that $\delta(s, a)$ is defined iff $\lambda(s, a)$ is defined for all $s \in S, a \in \Sigma$.*

A MM induces a corresponding LTS when input and output are viewed as alternating actions [vdPM19].

Definition 11 (LTS of a MM)**.** *Let $M = (S, s_0, \Sigma, \Lambda, \delta, \lambda)$ be a MM. Furthermore, let $T := \{(s, i) \mid \lambda(s, i)$ is defined $\}$ and $f : T \mapsto S_T$ be a bijection to a new set of states disjoint from S. Then the LTS $L(M) := (S \uplus S_T, \Sigma \uplus \Lambda, \longrightarrow)$ of M[3] is defined with*

$$\longrightarrow := \{s \xrightarrow{i} f(s, i) \mid (s, i) \in T\} \uplus \{f(s, i) \xrightarrow{o} s' \mid \lambda(s, i) = o \wedge \delta(s, i) = s'\}$$

A *modal Mealy machine* (MMM) can be defined analogously to an MTS by featuring separate transition and output functions for may and must behavior.

3.3 Action-based Linear Temporal Logic

The verification tasks considered in this thesis each feature an action-based temporal property: the task is to check whether the given program (or model thereof) satisfies that property. Setion 3.3.1 briefly covers linear-time properties and corresponding model checking of MTSs. Thereafter, Section 3.3.2 presents action-based linear temporal logic (LTL) as a means to describe linear-time properties.

[3]The operator $\uplus$ describes the disjoint union of two sets.

3.3.1 Linear-Time Properties

Linear-time properties specify behavior of infinite sequences. When not expressed in some logic, a linear-time property is frequently represented by a language of infinite words [SY20, BK08].

Definition 12 (Linear-Time Property)**.** *A language $\mathcal{L}$ over an alphabet Σ is a* linear-time property *iff $\mathcal{L} \subseteq \Sigma^\omega$. A word w* satisfies *$\mathcal{L}$ iff $w \in \mathcal{L}$.*

Interesting subsets of linear-time properties include *safety* and *liveness* [BK08].

Definition 13 (Safety and Liveness)**.** *A linear-time property $\mathcal{L}$ over some alphabet Σ is a* safety property *iff for every word $w \in \Sigma^\omega$ with $w \notin \mathcal{L}$, there exists a finite prefix $w_p \in \Sigma^*$ of w such that for all $w' \in \Sigma^\omega$, $w_p w' \notin \mathcal{L}$ holds.*

Moreover, $\mathcal{L}$ is a liveness property *iff every finite word $w \in \Sigma^*$ can be extended to a word $ww' \in \Sigma^\omega$ such that $ww' \in \mathcal{L}$.*

Model checking a linear-time property means to decide which of the following three possibilities hold $[\text{JSSS21}]_{\text{AP}}$:[4]

Definition 14 (Satisfaction/Violation Between MTSs and Linear-Time Properties)**.** *Let M be an MTS and $\mathcal{L}$ a linear-time property. Then M* satisfies *$\mathcal{L}$ (denoted as $M \models \mathcal{L}$) iff $\mathcal{L}^\omega_\top(M) \subseteq \mathcal{L}$. Similarly, M* violates *$\mathcal{L}$ (denoted as $M \not\models \mathcal{L}$) iff $\mathcal{L}^\omega_\bot(M) \not\subseteq \mathcal{L}$. Moreover, M* is indecisive concerning *$\mathcal{L}$ (denoted as $M \overset{?}{\models} \mathcal{L}$) iff M neither satisfies nor violates $\mathcal{L}$.*

A Mealy machine M satisfies, violates, or is indecisive concerning $\mathcal{L}$ iff the same holds for its LTS $L(M)$ (Def. 11). This propagation extends to modal Mealy machines.

3.3.2 Action-based LTL

The following definitions which are taken from $[\text{JSSS21}]_{\text{AP}}$ specify linear temporal logic (LTL) [WVS20, BK08], more precisely, action-based LTL [SY20, GM03]:

Definition 15 (Syntax of Action-based Linear Temporal Logic (LTL))**.** *Let Σ be an alphabet of actions and $a \in \Sigma$. The syntax of (action-based)* LTL *is defined using the following grammar in Backus-Naur form:*

$$\varphi ::= \top \mid a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid (\varphi\ \mathbf{U}\ \varphi)$$

LTL *is the set of formulas φ that can be constructed this way.*

The operator $\mathbf{X}$ (or "next") describes behavior that has to hold at the next time step. A formula $\varphi_1\ \mathbf{U}\ \varphi_2$ describes that φ_2 has to hold eventually and that φ_1 has to be satisfied until φ_2 holds in a word. The formal semantics of LTL is based on a satisfaction relation between infinite words and LTL formulas [BK08]:

[4] Compared to $[\text{JSSS21}]_{\text{AP}}$, this definition was generalized from LTL to arbitrary linear-time properties.

Definition 16 (LTL Semantics)**.** *Let Σ be an alphabet of action symbols. The satisfaction relation $\models \subseteq (\Sigma^\omega \times \mathrm{LTL})$ is defined as the minimal relation that adheres to the following rules for any $w \in \Sigma^\omega$ and $\varphi, \psi \in \mathrm{LTL}$:*

1. $w \models \top$
2. $w \models a$ *iff* $w_1 = a$
3. $w \models (\varphi \wedge \psi)$ *iff* $w \models \varphi$ *and* $w \models \psi$
4. $w \models \neg\varphi$ *iff* $w \not\models \varphi$
5. $w \models \mathbf{X}\,\varphi$ *iff* $w_{\geq 1} \models \varphi$
6. $w \models (\varphi\ \mathbf{U}\ \psi)$ *iff* $\exists k \in \mathbb{N} : w_{\geq k} \models \psi$ *and* $\forall i \in \mathbb{N}_{<k} : w_{\geq i} \models \varphi$

The semantics *of a formula $\varphi \in \mathrm{LTL}$ is given by $[\![\varphi]\!] := \{w \in \Sigma^\omega \mid w \models \varphi\}$.*

An MTS M satisfies, violates, or is indecisive concerning φ iff the same holds for the linear-time property $[\![\varphi]\!]$, respectively.

The terms "LTL formula" and "LTL property" are used interchangeably within this thesis. The former term has a more syntactical meaning whereas the latter focuses on semantics. Common abbreviations in LTL include $\mathbf{F}\,\varphi := \top\ \mathbf{U}\ \varphi$ which expresses that φ will eventually become true and its dual operator $\mathbf{G}\,\varphi := \neg\,\mathbf{F}\,\neg\varphi$ which claims that φ is generally true, meaning that it holds at every symbol of a word.

As LTL is frequently used to specify behavior of state-based systems, LTL semantics is commonly introduced such that 'characters' in a word are sets of atomic propositions $A \in 2^{AP}$ instead of individual actions $a \in \Sigma$. Action-based LTL can be formally seen as a subset of LTL based on sets of atomic propositions. The underlying reason is that it is possible to enforce that only individual atomic propositions occur in a word by adding an LTL invariant called *singleton filter* to a formula $[\mathrm{JS18}]_{\mathrm{AP}}$.

Every LTL property φ can be decomposed into two properties φ_s and φ_l in a way that $[\![\varphi_s]\!]$ is a safety property and $[\![\varphi_l]\!]$ a liveness property such that $[\![\varphi]\!] = [\![\varphi_s]\!] \cap [\![\varphi_l]\!]$ holds [AS87, MDB14].[5] If an LTL property φ is said to feature a safety part within this thesis, then $[\![\varphi]\!]$ is *not* a liveness property.

As known since the early days of LTL model checking, one can synthesize a Büchi automaton B from an LTL formula φ such that $\mathcal{L}(B) = [\![\varphi]\!]$ holds [PR89].[6]

Definition 17 (Büchi Automaton)**.** *Let $B = (S, \Sigma, \Delta, s_0, F)$ be a finite automaton with a set S of states and a finite alphabet Σ. State $s_0 \in S$ represents the initial state and $F \subseteq S$ a set of accepting states. The relation $\Delta \subseteq (S \times \Sigma \times S)$ represents transitions between states in S. The syntax $p \xrightarrow{a} q$ denotes a $(p, a, q) \in \Delta$.*

A path *is a sequence $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots$ of transitions that starts in s_0 with an index that ranges from 0 to either a positive integer or infinity. Path π* spells *the word $w = a_1 a_2 \ldots$.*

[5] Note that for LTL, the definition of safety and liveness relies on the use of action-based LTL: when allowing sets of atomic propositions and therefore using the mentioned singleton filter, there would not exist liveness properties because the singleton filter itself is an invariant and thus a safety property.

[6] Note again that a singleton filter can be used to synthesize Büchi automata for action-based LTL.

Given these definitions, B is called a Büchi automaton *if it adheres to Büchi acceptance, meaning that it accepts infinite words $w \in \Sigma^\omega$ based on the following criteria:*

1. *There exists a path p in B that starts in s_0 and that spells w*
2. *This path p visits a state in F infinitely often*

The set $\mathcal{L}(B) := \{w \in \Sigma^\omega \mid B \text{ accepts } w\}$ defines the language of B.

Büchi automata are strictly more expressive than LTL [Wol83]. Examples of Büchi automaton synthesis from LTL properties can be found in [JS18, JSSS21]$_{\text{AP}}$.

3.4 Property Preservation

Key to the presented synthesis of verification tasks are property-preserving transformations. The fundamental concepts that this preservation is based on are modal refinement and bisimulation. Section 3.4.1 covers modal refinement, before weak refinement, bisimulation, and the notion of convergence are introduced in Section 3.4.2.

3.4.1 Modal Refinement

The following notion of refinement allows to regard certain LTSs as implementations of MTSs [Lar89] (see also [JSSS21]$_{\text{AP}}$[7]).

Definition 18 (MTS Refinement)**.** *Let $M_1 = (S_1, s_0^1, \Sigma_1, \dashrightarrow_1, \longrightarrow_1)$, $M_2 = (S_2, s_0^2, \Sigma_2, \dashrightarrow_2, \longrightarrow_2)$ be MTSs. A relation $\precsim \subseteq (S_1 \times S_2)$ is called a* refinement *iff the following hold for all $(p, q) \in \precsim$:*

$$1.)\ \forall p \overset{a}{\dashrightarrow}_1 p',\ \exists q \overset{a}{\dashrightarrow}_2 q' : p' \precsim q'$$
$$2.)\ \forall q \overset{a}{\longrightarrow}_2 q',\ \exists p \overset{a}{\longrightarrow}_1 p' : p' \precsim q'$$

M_1 refines *M_2, written as $M_1 \precsim M_2$, iff there exists a refinement $\precsim$ with $s_0^1 \precsim s_0^2$.*

Because every must transition is also a may transition, every reachable state of M_1 needs to occur in a refinement:

Proposition 3 (States Contained in a Refinement)**.** *Let $M_1 = (S_1, s_0^1, \Sigma_1, \dashrightarrow_1, \longrightarrow_1)$, $M_2 = (S_2, s_0^2, \Sigma_2, \dashrightarrow_2, \longrightarrow_2)$ be two MTSs. Then for any refinement $\precsim \subseteq (S_1 \times S_2)$ with $s_0^1 \precsim s_0^2$, the following holds: for any state $s_1 \in S_1$ that is reachable in M_1, there exists a state $s_2 \in S_2$ such that $s_1 \precsim s_2$.*

The following observation explains why modal refinement preserves linear-time properties [JSSS21]$_{\text{AP}}$ (see also Section 3.3).

Proposition 4 (Language Monotonicity I)**.** *Let M, M' be two MTSs such that $M' \precsim M$. Then it holds that $\mathcal{L}_\bot(M) \subseteq \mathcal{L}_\bot(M')$ and $\mathcal{L}_\top(M') \subseteq \mathcal{L}_\top(M)$.*

[7] In contrast to the definition in [JSSS21]$_{\text{AP}}$, M_1 and M_2 are allowed to have different alphabets here in order to support alphabet extensions (Section 6.2) in the setting of weak refinement (Section 3.4.2).

Parallel components are relevant during model checking if they cannot be abstracted to their weakest specification [Lar90] (see also $[\text{SJ17}]_{\text{AP}}$):

Definition 19 (Weakest Modal Specification)**.** *Let Σ be an alphabet. The one-state MTS $\mathcal{U}_\Sigma$ with may transitions for every $a \in \Sigma$ is called the* weakest modal Σ-specification*:*

$$\mathcal{U}_\Sigma := (\{s\}, s, \Sigma, \{s\} \times \Sigma \times \{s\}, \emptyset)$$

The following proposition follows from the semantics of parallel composition (Def. 8).

Proposition 5 (Unconstrained Behavior)**.** *Let M be an MTS and $\mathcal{U}_\Sigma$ the weakest modal Σ-specification for some alphabet Σ. Then $M \lesssim M \,||\, \mathcal{U}_\Sigma$.*

3.4.2 Weak Modal Refinement, Bisimulation, and Convergence

With the goal to define weak modal refinement, the common notion of hiding is introduced as in $[\text{SJ17}]_{\text{AP}}$:

Definition 20 (Label Hiding)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS and $\Gamma \subseteq \Sigma$. Then in the* Γ-hiding

$$\text{hide}_\Gamma(M) := (S, s_0, (\Sigma \setminus \Gamma) \cup \{\tau\}, \text{hide}_\Gamma(\dashrightarrow), \text{hide}_\Gamma(\longrightarrow))$$

of M, all transitions t of M with $\Sigma(t) \in \Gamma$ are replaced with the (unobservable) special symbol τ and therefore features the following transition relations for all $\rightharpoonup \in \{\dashrightarrow, \longrightarrow\}$:

$$\text{hide}_\Gamma(\rightharpoonup) = \{p \xrightarrow{\tau} q \mid \exists a \in \Gamma : p \xrightarrow{a} q\} \uplus \{p \xrightarrow{a} q \mid a \in \Sigma \setminus \Gamma\}$$

In order to compare two MTSs based on a certain alphabet Γ, it helps to express that all of their symbols are hidden which are not in Γ $[\text{SJ19}]_{\text{AP}}$:

Definition 21 (Alphabet View of an MTS)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS and $\Gamma \subseteq \Sigma$. Then $[M]_\Gamma := \text{hide}_{\Sigma(M)\setminus\Gamma}(M)$.*

Throughout this thesis, the special symbol τ *only* occurs when hiding or view operators are used, unless explicitly stated otherwise. For every MTS M without hiding or a view applied to it or a note that its alphabet may contain τ , one can assume that $\tau \notin \Sigma(M)$.

A view has no effect if the viewed alphabet contains all symbols of the given MTS.

Proposition 6 (Neutral View)**.** *Let M be an MTS and Σ an alphabet with $\Sigma(M) \subseteq \Sigma$. Then $[M]_\Sigma = M$ holds.*

A view can be applied before or after parallel composition if the viewed alphabet contains all symbols based on which synchronization can occur.

Proposition 7 (Commutativity I)**.** *Let M, M' be two MTSs and Σ an alphabet with $\Sigma(M) \cap \Sigma(M') \subseteq \Sigma$. Then $[M]_\Sigma \,||\, [M']_\Sigma = [M \,||\, M']_\Sigma$.*

Just like $[M]_\Gamma$ views an MTS M w.r.t. a certain alphabet Γ, the same concept is useful for languages [JSSS21]$_{\text{AP}}$.

Definition 22 (Alphabet View of a Language)**.** *For any languages Σ, Γ and any word w over Σ, $[w]_\Gamma$ is defined as the word that results from skipping all symbols in w that do* not *exist in Γ. This definition extends naturally to languages.*

Based on hiding, the (standard) definition of weak MTS refinement is prepared by defining the usual observational relation of a transition relation as in [SJ17]$_{\text{AP}}$:

Definition 23 (Observational Relation)**.** *Given states S, let $\rightharpoonup \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ be a transition relation between states in S and $p, p', q, q' \in S$. Furthermore, let $\epsilon \notin \Sigma$ be a special symbol representing a sequence of invisible actions. The* observational relation $\text{obs}(\rightharpoonup)$ *of $\rightharpoonup$ is then recursively defined as follows:*

$$p \overset{\epsilon}{\Longrightarrow} p \qquad \frac{p \overset{\tau}{\rightharpoonup} p' \quad p' \overset{\epsilon}{\Longrightarrow} q}{p \overset{\epsilon}{\Longrightarrow} q} \qquad \frac{p \overset{\epsilon}{\Longrightarrow} p' \quad p' \overset{a}{\rightharpoonup} q' \quad q' \overset{\epsilon}{\Longrightarrow} q}{p \overset{a}{\Longrightarrow} q}$$

where $a \in \Sigma$ holds and $p \overset{a}{\Longrightarrow} p'$ denotes a transition $(p, a, p') \in \text{obs}(\rightharpoonup)$.

An observational MTS is now simply defined by replacing its original transition relations with their observable counterparts [SJ17]$_{\text{AP}}$:

Definition 24 (Observational MTS)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS. The* observational MTS $\text{obs}(M)$ *of M is based on the observational expansion of its transition relations:*

$$\text{obs}(M) := (S, s_0, (\Sigma \setminus \{\tau\}) \cup \{\epsilon\}, \text{obs}(\dashrightarrow), \text{obs}(\longrightarrow))$$

Some refinements are insensitive to *divergence*, i.e., the possibility that a system engages in an infinite τ sequence, and therefore do not preserve liveness properties [SJ17]$_{\text{AP}}$. The major part of this thesis therefore focuses on convergent systems.

Definition 25 (Convergence)**.** *An MTS M is called* convergent *iff for every path π in M, the word $w(\pi)$ only contains finite τ-sequences.*

This is sufficient to introduce (convergent) weak modal refinement [HL89] (see [SJ17]$_{\text{AP}}$):

Definition 26 ((Convergent) Weak Refinement)**.** *Let M, M' be two MTSs. Then M'* weakly refines *M, denoted as $M' \lessapprox M$, iff $\text{obs}(M') \lesssim \text{obs}(M)$. If M' is furthermore convergent, then M' is called* convergent weak refinement *of M, denoted as $M' \overset{c}{\lessapprox} M$.*

It is straightforward to establish that parallel composition preserves modal refinement for both operands when they have the same alphabets [SJ17]$_{\text{AP}}$. The same holds for weak refinement [BMSH10], and convergence does not influence this preservation.

When allowing different alphabets of involved MTSs however, compositionality does require an additional guarantee w.r.t. these alphabets.[8] This is based on the fact that

[8] See also Propositions 6 and 7 in [JSSS21]$_{\text{AP}}$ for the required guarantees in the linear-time setting.

refinement is in some cases insensitive to the alphabets of MTSs: there exist MTSs M, M' with $M \lesssim M'$ such that $\Sigma(M) \neq \Sigma(M')$ holds. Altering the alphabet of an MTS might however modify synchronization constraints with a parallel context and thereby hinder refinement compositionality.

Proposition 8 (Compositional Refinement)**.** *Let M, M', M'' be three MTSs where M and M' may contain τ in their alphabets and such that*[9] $(\Sigma(M) \triangle \Sigma(M')) \cap \Sigma(M'') = \emptyset$. *Then for any $\preceq \in \{\lesssim, \lessapprox, \overset{c}{\lessapprox}\}$, $\preceq$ is preserved by parallel composition:*

$$M \preceq M' \quad \textit{implies} \quad (M \,||\, M'') \preceq (M' \,||\, M'')$$

Note that due to the commutativity of operator $||$, Proposition 8 holds for both components of a composition.

The frequently used concept of (weak) bisimulation can be derived from the notion of (weak) refinement $[\text{SJ19}]_{\text{AP}}$ such that it coincides with the traditional definition [Par81].

Definition 27 ((Convergent, Weak) Bisimulation)**.** *Let L, L' be two LTSs. Then L is* bisimilar *to L', denoted as $L \sim L'$, iff $L \lesssim L'$. In addition, L is* weakly bisimilar *to L', denoted as $L \approx L'$, iff $L \lessapprox L'$. Furthermore L and L' are* convergent and weakly bisimilar*, denoted as $L \overset{c}{\approx} L'$, iff both L and L' are convergent and $L \approx L'$.*

Bisimulation of Mealy machines is based on the corresponding LTS views (see Section 3.2.3).

[9] The operator $\triangle$ stands for the symmetric difference between sets.

4 Realistic Verification Tasks

This chapter introduces the different types of verification tasks that can be synthesized using the new framework (see Section 2.3.1) on a formal level. Section 4.1 defines these different types of tasks. Thereafter, Section 4.2 clarifies what is meant by interruptible properties—such properties are part of synthesized tasks which target parallel programs—and presents a new result by generalizing the notion of interruptibility as presented in [SY20] to temporal properties from various logics. This chapter closes with a discussion of realized hardness criteria (see also Figure 2.3).

4.1 Verification Tasks

On the one hand, Mealy machines are chosen within this thesis as the intermediate representation of sequential programs, the latter implementing reactive systems. On the other hand, parallel compositions of labeled transition systems represent parallel, asynchronous programs. A verification task combines such a model with a temporal property, i.e., a property that can be expressed in the modal μ-calculus [Koz83].

Definition 28 (Verification Task)**.** *A model M of a system and a temporal property φ specify a* verification task $V(M, \varphi)$ *that is a* positive *verification task iff $M \models \varphi$ and a* negative *verification task iff $M \not\models \varphi$.*

If M is a MM, then $V(M, \varphi)$ is called sequential (positive or negative) task*. If M is a parallel composition $M = (L_1 \,||\, \cdots \,||\, L_n)$ of LTSs, then $V(M, \varphi)$ is called* parallel task*.*

As an intermediate representation during the generation of verification tasks, MTSs (Def. 3) are used within this thesis. Hardness criteria that are introduced in Section 4.3 are therefore sometimes specified based on MTSs. For a parallel task, the parallel composition of LTSs is obtained by means of component-wise modal refinement (Def. 18).

Definition 29 (LTS Component Refinement)**.** *Let $M = (M_1 \,||\, \cdots \,||\, M_n)$ be a parallel composition of n MTSs. A parallel composition $L = (L_1 \,||\, \cdots \,||\, L_n)$ of n LTSs is called* LTS component refinement *of M iff $L_i \lesssim M_i$ holds for each $i \in 1 \,..\, n$.*

In addition to verification tasks—which are inherently model checking queries as defined above—the synthesis framework presented in this thesis can also be used to create tasks for weak bisimulation checking $[\text{SJ19}]_{\text{AP}}$.

Definition 30 (Bisimulation Task)**.** *Let* $L_1 = (L_{11} \,||\, ... \,||\, L_{1n})$ *and* $L_2 = (L_{21} \,||\, ... \,||\, L_{2n})$ *be two parallel compositions of LTSs and* $\Sigma \subseteq \Sigma(L_1) \cap \Sigma(L_2)$*. Then* $B = (L_1, L_2, \Sigma)$ *is called a* weak bisimulation checking task*, or just* bisimulation task*.*

The correct answer *to* B *is 'equivalent' iff* $[L_1]_\Sigma \approx [L_2]_\Sigma$ *holds and 'nonequivalent' otherwise (see Definitions 21 and 27).*

4.2 Interruptible Temporal Properties

Convergent weak modal refinement will be ensured by all transformations applied during the construction of a parallel task (see Chapter 6). Intuitively speaking, the presented generation of (hard) parallel tasks interleaves finite interruptions by artificial new action symbols with the actions of an initial system. As a consequence, the approach presented in this thesis to generate parallel tasks is constrained to *interruptible* temporal properties.

Definition 31 (Interruptible Temporal Property)**.** *Let* φ *be a temporal property. Then* φ *is* interruptible *iff the following holds for all MTSs* M, M_E*:*

$$[M_E]_{\Sigma(M)} \overset{c}{\lessapprox} M \;\; \textit{implies both}^1 \; M \models \varphi \implies M_E \models \varphi \textit{ and } M \not\models \varphi \implies M_E \not\models \varphi$$

Convergent weak modal refinement based on hiding as used in Definition 31 is compositional if the alphabet of M_E is a superset of that of M and does not introduce additional synchronization potential with parallel components.

Theorem 1 (Compositionality of Property Preservation)**.** *Let* M, M'*, and* M'' *be MTSs. Then a satisfaction of the two following conditions*[2] *implies* $[M'||M'']_{\Sigma(M||M'')} \overset{c}{\lessapprox} M||M''$*:*

1. $[M']_{\Sigma(M)} \overset{c}{\lessapprox} M$
2. $(\Sigma(M) \triangle \Sigma(M')) \cap \Sigma(M'') = \emptyset$

Proof. Applying set theory and the second condition in Theorem 1 yields the equality $\Sigma(M) \cap \Sigma(M'') = \Sigma(M') \cap \Sigma(M'')$. Combined with Proposition 7, this results in (i):

$$[M' \,||\, M'']_{\Sigma(M||M'')} = [M']_{\Sigma(M||M'')} \,||\, [M'']_{\Sigma(M||M'')}$$

Because of Proposition 6, it holds that (ii) $[M'']_{\Sigma(M||M'')} = M''$. According to the definition of an alphabet view, $[M']_{\Sigma(M||M'')} = \text{hide}_{\Sigma(M')\setminus\Sigma(M||M'')}(M')$. It holds that $\Sigma(M \,||\, M'') = \Sigma(M) \cup \Sigma(M'')$ and again based on set theory also the following:

$$\Sigma(M') \setminus \Sigma(M \,||\, M'') = (\Sigma(M') \setminus \Sigma(M)) \setminus \Sigma(M'')$$

[1] Note that an MTS may neither satisfy nor violate a given temporal property.
[2] Operator $\triangle$ again stands for the symmetric difference between alphabets.

The outer set difference can be omitted due to the second condition in Theorem 1, thus it holds that (iii) $[M']_{\Sigma(M||M'')} = \text{hide}_{\Sigma(M')\setminus\Sigma(M)}(M') = [M']_{\Sigma(M)}$. Now it follows that

$$[M' \,||\, M'']_{\Sigma(M||M'')} = [M']_{\Sigma(M)} \,||\, M''$$

because of (i), (ii), and (iii). Combining Condition 1 and Proposition 8 finally yields:

$$[M' \,||\, M'']_{\Sigma(M||M'')} = [M']_{\Sigma(M)} \,||\, M'' \overset{\text{c}}{\lessapprox} M \,||\, M''$$

□

Theorem 1 serves as the formal foundation for property preservation when combining parallel decomposition and alphabet extension (see Chapter 6).

For linear-time properties, interruptibility can be expressed directly on the level of languages [SY20] (see also $[\text{JSSS21}]_{\text{AP}}$), as shown by relying on the following lemmas.

Lemma 1 (Consistency of Alphabet Views)**.** *Let L be an LTS and Γ an alphabet such that $[L]_\Gamma$ is convergent. Then $\mathcal{L}^\omega(\text{obs}([L]_\Gamma)) = [\mathcal{L}^\omega(L)]_\Gamma$ holds.*

Proof. Let $w' \in \mathcal{L}^\omega(\text{obs}([L]_\Gamma))$. Then there exists an infinite path π in $\text{obs}([L]_\Gamma)$ with $w(\pi) = w'$. There further exists an infinite path π' in L that (i) traverses the same sequence of states as π does because $[L]_\Gamma$ is convergent, and (ii) such that for all positive integers i, the i-th transition in π is labeled $a \in \Gamma$ iff the i-th transition of π' is labeled identically, because of the two leftmost rules in the definition of observational transition relations (Def. 23). The word that results from skipping symbols in word $w(\pi') \in \mathcal{L}^\omega(L)$ that are not in Γ is thus identical to w'.

Let $w' \in [\mathcal{L}^\omega(L)]_\Gamma$. Then there exists an infinite path π in L with $w' = [w(\pi)]_\Gamma$. In $[\mathcal{L}^\omega(L)]_\Gamma$, all transitions in π labeled with a symbol not in Γ are replaced by τ, and again by ϵ in $\text{obs}([\mathcal{L}^\omega(L)]_\Gamma)$ due to the two above-mentioned rules of observational transition relations. As a result, there exists an infinite path π' in $\text{obs}([L]_\Gamma)$ with $w(\pi') = w'$. Because $[L]_\Gamma$ is convergent, $w(\pi')$ has to be an infinite word, and thus $w' \in \mathcal{L}^\omega(\text{obs}([L]_\Gamma))$. □

Modification to an individual transition relation such as (i) hiding and (ii) a transformation to an observable transition relation are independent from managing different transition relations. As a consequence, the following propositions hold.

Proposition 9 (Commutativity II)**.** *For any MTS M and any alphabet Γ, the following hold:*

$$L_\top([M]_\Gamma) = [L_\top(M)]_\Gamma \quad \textit{and} \quad L_\bot([M]_\Gamma) = [L_\bot(M)]_\Gamma$$

Proposition 10 (Commutativity III)**.** *For any MTS M, the following hold:*

$$L_\top(\text{obs}(M)) = \text{obs}(L_\top(M)) \quad \textit{and} \quad L_\bot(\text{obs}(M)) = \text{obs}(L_\bot(M))$$

Combining Lemma 1 with Propositions 9 and 10 allows to show the following.

Lemma 2 (Language Monotonicity II)**.** *Let M, M' be MTSs with $[M']_{\Sigma(M)} \overset{c}{\lessapprox} M$. Then $\mathcal{L}^\omega_\bot(M) \subseteq [\mathcal{L}^\omega_\bot(M')]_{\Sigma(M)}$ and $[\mathcal{L}^\omega_\top(M')]_{\Sigma(M)} \subseteq \mathcal{L}^\omega_\top(M)$.*

Proof. Let M, M' be two MTSs such that $[M']_{\Sigma(M)} \overset{c}{\lessapprox} M$. Because of $\tau \notin \Sigma(M)$, it follows that $\mathcal{L}^\omega_\bot(\text{obs}(M)) = \mathcal{L}^\omega_\bot(M)$. As $[M']_{\Sigma(M)} \lessapprox M$ holds, it is known that $\text{obs}([M']_{\Sigma(M)}) \lesssim \text{obs}(M)$, which implies $\mathcal{L}^\omega_\bot(M) = \mathcal{L}^\omega_\bot(\text{obs}(M)) \subseteq \mathcal{L}^\omega_\bot(\text{obs}([M']_{\Sigma(M)}))$ according to Proposition 4. Propositions 9 and 10 yield

$$\mathcal{L}^\omega_\bot(M) \subseteq \mathcal{L}^\omega_\bot(\text{obs}([M']_{\Sigma(M)})) = \mathcal{L}^\omega(L_\bot(\text{obs}([M']_{\Sigma(M)}))) = \mathcal{L}^\omega(\text{obs}([L_\bot(M')]_{\Sigma(M)})),$$

and applying Lemma 1 now results in $\mathcal{L}^\omega_\bot(M) \subseteq [\mathcal{L}^\omega(L_\bot(M'))]_{\Sigma(M)} = [\mathcal{L}^\omega_\bot(M')]_{\Sigma(M)}$. The proof of the second statement, $[\mathcal{L}^\omega_\top(M')]_{\Sigma(M)} \subseteq \mathcal{L}^\omega_\top(M)$, is analogous. □

Based on Lemma 2, it can now be proven that Definition 31 is a generalization of the definition of interruptibility that is introduced in [SY20].

Theorem 2 (Sufficient Condition for Linear-Time Interruptibility)**.** *Let $\mathcal{L}$ be a linear-time property over an alphabet Σ. Then $\mathcal{L}$ is* interruptible *if for any alphabet Σ_E and any infinite words $w \in \Sigma^\omega, w_E \in \Sigma_E^\omega$, the following holds:*

$$[w_E]_\Sigma = w \quad \textit{implies} \quad (w_E \in \mathcal{L} \iff w \in \mathcal{L})$$

Proof. Let $\mathcal{L}$ be an arbitrary linear-time property over some alphabet Σ and assume that $\mathcal{L}$ satisfies the premise in Theorem 2. Let M, M_E be any MTSs with $[M_E]_{\Sigma(M)} \overset{c}{\lessapprox} M$. Furthermore, assume that $M \models \mathcal{L}$, meaning that $\mathcal{L}^\omega_\top(M) \subseteq \mathcal{L}$ (Def. 14). Based on Lemma 2, $[\mathcal{L}^\omega_\top(M_E)]_{\Sigma(M)} \subseteq \mathcal{L}^\omega_\top(M)$ holds. Combined with the satisfied condition in Theorem 2, it thus follows that $M_E \models \mathcal{L}$. The other case, assuming that $M \not\models \mathcal{L}$ holds, is analogous. □

4.3 Hardness Guarantees

The different hardness criteria that can be guaranteed using the new framework of this thesis are discussed in the following. The structure of this section can also be found in Figure 2.3. Section 4.3.1 first introduces two variants of state explosion, followed by Section 4.3.2 which clarifies the meaning of rare and deep counterexamples w.r.t. linear-time properties. Thereafter, Section 4.3.3 is dedicated to a relevant parallel context and relies on the notion that all components of a parallel composition are relevant when solving a given task.

4.3.1 Large State Space

When generating realistic verification tasks or bisimulation tasks, one should be able to confront verifiers with what is commonly referred to as state explosion: the number of reachable explicit program states is exponential in the syntactic size of the program. This thesis distinguishes between two types of state explosion. On the one hand, using a

bisimulation-preserving *encoding* of a model—for example a MM—in some programming language allows to generate code that features a number of reachable program states that is exponential in that of the underlying model. This approach is the topic of [HIM+14, Sch21] and not further discussed in this thesis. For parallel tasks on the other hand, another type of state explosion exists based on multiple parallel *interleavings*:

Definition 32 (Interleaving-based State Explosion)**.** *Let $L = (L_1 || \cdots || L_n)$ be a parallel composition of n LTSs. Then L is* interleaving-hard *iff the expanded composition of L contains at least 2^n reachable states.*

Encoding-based state explosion is used to generate hard sequential tasks within the RERS Challenge [HIM+14], whereas interleaving-based state explosion is utilized to produce hard parallel tasks. Note that a parallel composition of LTSs can also be encoded as a program in, e.g., Promela $[\text{GJS}^+16]_{\text{AP}}$. Therefore, an encoding-based expansion could further be employed to enlarge individual components of a parallel composition.

4.3.2 Subtle Errors

A hardness criterion that solely applies to negative verification tasks is *subtlety*: In practice, errors ("bugs") in programs are frequently (i) rare and (ii) deep in the sense that an error can only be detected after a certain number of computation steps $[\text{JS18}]_{\text{AP}}$. Thus, a generator for realistic verification tasks should be able to control the subtlety of errors (property violations) in order to produce relevant benchmarks. This thesis discussed subtle errors in the context of LTL properties, which subsume error reachability such as violated assertions.

Rarity

For LTL properties which constrain infinite paths (see Def. 16), rarity of counterexamples cannot easily be expressed, e.g., in terms of a quotient of the number of paths that violate or satisfy a given property, respectively, because there usually exist infinitely many infinite paths in a system. Instead, this thesis uses the notion of a *counterexample handle* which all counterexamples to an LTL property in a given system have to traverse.

Definition 33 (Counterexample Handle)**.** *Let M be a transition system and φ an LTL property such that $M \not\models \varphi$. A transition t in M is called a* counterexample handle (CE-handle) *for φ in M iff the removal of t results in $M \models \varphi$.*

Note that if M is an MTS, then a CE-handle has to be a must transition (see Def. 14). A CE-handle imposes a certain rarity of violation witnesses if most states of M can be reached without traversing it. In addition, such a handle is very useful for *controlling* the satisfaction of φ in M, an aspect that the generation of hard parallel tasks as presented in $[\text{JSSS21}]_{\text{AP}}$ heavily relies on.

Depth

For the purpose of evaluating verification tools, it is helpful to be able to adjust the *depth* at which counterexamples can be detected. Here, the goal is to set an interval $(m, n]$ of positive integers m, n with $m < n$ such that no path of m states or less gives away a property violation, however such that there exists a path of at most n states which does. The following pattern is used to generate corresponding verification tasks: starting from a small model, paths that indicate a property violation and are deemed too short are rendered infeasible, before a subsequent check ensures that one such violating path of length at most n still exists. A depth interval $(m, n]$ for the violation of an LTL property φ in a model M can be stated in at least two ways such that it is relevant for model checking:

1. Guarantee that every counterexample path to φ in M contains at least m distinct states, however that one exists which contains at most n distinct states.
2. Ensure that every word of length less or equal to m can in general be extended to a word that satisfies φ, however also that there exists a word of length at most n for which all possible continuations violate φ.

These two perspectives on depth intervals have different impact on the hardness of a negative verification task, and the second one is apparently only applicable to properties that feature a safety part (see Def. 13). The first option ensures for example that an explicit-state analyzer which proceeds in a breadth-first search cannot determine a property violation before having reached depth m. The underlying reason is that a state has to be observed twice on the same path to obtain an infinite word. For properties that feature a safety part however, this does not exclude the possibility that a path of less than m states already indicates the violation of φ if it can be continued to some infinite path. If such an "early indicator" prefix of a path exists, a smart search heuristic for counterexamples might prioritize the exploration of its successor states. This type of *monitorability* is excluded by the second of the above-listed options. Please note that neither of the above guarantees implies the other, and that both can be combined.

The first guarantee of at least m distinct states is comparably easy to accomplish and has been used to generate benchmarks for the RERS Challenge since at least 2016 $[\text{GJS}^+16]_{\text{AP}}$: by iteratively *unrolling* a simple cycle that the shortest counterexample to φ in M traverses (in terms of distinct states), one can retrieve a bisimilar model M' where this counterexample has been prolonged. A check for a violating path of at most n states then ensures the upper bound of the hardness interval and frequently succeeds when using incremental loop unrolling. This first type of depth interval is thus not further discussed in this thesis.

In contrast, realizing the second type of depth interval—especially prohibiting that an LTL monitor is able to detect that a short word can no longer satisfy φ—is more involved and has only recently been used for RERS benchmarks $[\text{HJM}^+21]_{\text{AP}}$. The following definitions are taken from $[\text{HJM}^+21]_{\text{AP}}$ and formally introduce this hardness-related depth guarantee on the level of languages of infinite words.

Definition 34 (Violating Prefix)**.** *Let $w \in \Sigma^*$. Then w violates φ iff the following holds:*

$$\forall w' \in \Sigma^\omega.\ ww' \not\models \varphi$$

An infinite word $w \in \Sigma^\omega$ k-violates φ iff its prefix $w_{\leq k}$ violates φ. A language $\mathcal{L}' \subseteq \Sigma^\omega$ k-violates φ iff there exists a word $w \in \mathcal{L}$ which k-violates φ.

Intuitively speaking, a finite word violates φ if it cannot be extended to a word that satisfies φ (see also Def. 13). The following proposition follows straightforwardly:

Proposition 11 (Monotonicity)**.** *If a word $w \in \Sigma^\omega$ k-violates φ, then for all $k' \in \mathbb{N}$ with $k' \geq k$, w also k'-violates φ.*

This monotonicity property allows to specify $(m, n]$-depth simply based on the boundaries of this integer interval.

Definition 35 (Counterexample Depth)**.** *A language $\mathcal{L} \subseteq \Sigma^\omega$ is called $(m, n]$-deep w.r.t. φ iff the following hold:*

1. *$\mathcal{L}$ does* not *m-violate φ*
2. *$\mathcal{L}$ n-violates φ*

Note that if $\mathcal{L}$ is $(m, n]$-deep w.r.t. φ, then it is also $(m', n']$-deep for any positive integers m', n' with $m' \leq m$ and $n \leq n'$. As a consequence, there always exists a minimal depth interval if one exists in general.

4.3.3 Relevant Parallel Context

Parallel verification tasks give rise to additional hardness criteria due to the existence of multiple parallel components. Even if a parallel task $V(L, \varphi)$ is interleaving-hard (Def. 32), one might be able to abstract from a large subset—in the worst case all but one—parallel components of L while still being able to correctly analyze whether or not L satisfies φ. The following definitions are mostly taken from [SJ17, JSSS21]$_{\text{AP}}$ and are used for a hardness criterion that excludes the possibility to easily solve even an interleaving-hard task by abstracting from entire components.

Definition 36 (Component Abstraction)**.** *Let $M = (M_1 \,||\, \cdots \,||\, M_n)$ be a parallel composition of MTSs, $\Sigma_i = \Sigma(M_i)$ the alphabet of the i-th component of M, and $\mathcal{U}_{\Sigma_i}$ the weakest modal Σ_i-specification (see Def. 19). Then the parallel MTS composition*

$$\alpha(M, i) := (M_1 \,||\, \cdots \,||\, M_{i-1} \,||\, \mathcal{U}_{\Sigma_i} \,||\, M_{i+1} \,||\, \cdots \,||\, M_n)$$

*is called the i-*th component abstraction *of M.*

Definition 37 (φ-Lossy Generalization)**.** *Let M be an MTS and φ a temporal property such that M either satisfies or violates φ. Then any MTS M' that is indecisive concerning φ is called a φ-*lossy generalization *of M.*

System sensitivity guarantees that all components of a parallel composition are relevant for the verification/refutation of a considered property φ.

Definition 38 (System-Sensitive Properties)**.** *Let $M = (M_1 \,||\, \cdots \,||\, M_n)$ be a parallel composition of MTSs and φ a temporal property. We call φ M-*sensitive *iff the following holds:*

$$\forall i \in 1 \,..\, n : \alpha(M, i) \text{ is a } \varphi\text{-lossy generalization of } M$$

As a consequence of the above definition, a parallel task $V(L, \varphi)$ such that φ is L-sensitive is a task where every component matters for the verification or refutation of φ (see $[\text{JSSS21}]_{\text{AP}}$). Together with interleaving-based state explosion, this constitutes what is called a *hard parallel task* within this thesis:

Definition 39 (Hard Parallel Task)**.** *Let $V(L, \varphi)$ be a parallel task such that L contains n parallel components. Then $V(L, \varphi)$ is called n-*hard *iff the following hold:*

1. *L is interleaving-hard*
2. *Property φ is L-sensitive*

In $[\text{JSSS21}]_{\text{AP}}$, an approach to generate hard parallel tasks for any interruptible LTL property is introduced. That approach can further be easily modified to produce parallel tasks where many but not all components matter when analyzing if φ is satisfied, meaning that the relevance of a given parallel context can be adjusted. Moreover, it allows to ensure *property locality*.

Definition 40 (Property Locality)**.** *In a parallel task $V(L, \varphi)$, property φ is k-local in L iff it only contains symbols of k different parallel components of L.*

Locality is not necessarily related to hardness as some approaches to solve a task might benefit from knowing that a property only constrains one (or a few) parallel components, whereas other, likely compositional approaches might benefit from non-local properties.

However, it should be stated that certain hard parallel tasks can likely be solved easily by compositional model checking [CLM89] if their property is not local. As an example, one could generate a positive verification task which features an LTL property by

1. synthesizing n LTSs such that each L_i satisfies an LTL property φ_i,
2. defining $\varphi := \bigwedge_{i \in 1..n} \varphi_i$, and
3. choosing $L := (L_1 \,||\, \cdots \,||\, L_n)$.

Because parallel composition can only reduce the language of an LTS w.r.t. its own alphabet (see Proposition 5 in $[\text{JSSS21}]_{\text{AP}}$), L satisfies φ. Generating the different L_i such that L is interleaving-hard is trivial as the alphabets of the L_i have not been constrained by the sketched construction: they might even be pairwise-disjoint. In that case, every non-trivial φ_i would contain a symbol unique to L_i, rendering φ L-sensitive.

A compositional model checker could verify that $L_i \models \varphi_i$ holds for every $i \in 1 \,..\, n$ and then deduce that $L \models \varphi$ holds without ever computing the expanded LTS of L. In addition, property φ would likely be quite lengthy and therefore not necessarily realistic.

Based on the above example, it becomes apparent that it is desirable to—at least optionally—feature a certain locality of properties in a parallel task. This does not only allow to generate tasks that test the limits of state-of-the-art tools for compositional model checking, but also allows to better simulate the scenario of a *relevant parallel context*: the correctness of a single component is of interest, however its behavior is (directly or indirectly) influenced by communication with other parallel components.

5 Synthesizing Realistic Tasks

The starting point for synthesizing a verification task according to the new framework (Section 2.3.1) is presented in this section. Regardless of whether or not the generated task is sequential or parallel, the presented approach always starts with LTL synthesis (Section 5.1) to construct a Büchi automaton [Bü66] B that features realistic behavior. In order to fulfill realistic hardness criteria for negative verification tasks as discussed in Section 4.3.2, the subsequent Section 5.2 presents how such a constructed automaton B can be altered to realize rarely occurring property violations based on a CE-handle (Def. 33), before Section 5.3 shows how B can be modified to ensure deep counterexamples (Def. 35). Afterwards, Section 5.4 sketches how automaton B can be transformed to an MTS, and thereby paves the way for the parallel decomposition that will be the topic of Section 6.

As illustrated in Figure 2.2, a generated MTS can further be transformed to a (modal) Mealy machine $[\text{JS18}]_{\text{AP}}$ and then encoded as a sequential C or Java program along the lines of [HIM+14]. Details of the transformation to a Mealy machine are omitted here.

5.1 Temporal-Logic Synthesis

As a starting point for synthesizing realistic verification tasks, patterns for interesting LTL properties are used within this thesis. Such a pattern can reflect realistic scenarios [DAC99] or ensure, e.g., that properties are interruptible (see Theorem 2). From this pattern, both the LTL property φ that has to be analyzed in the generated task $V(M, \varphi)$ as well as structural LTL properties Φ are (randomly) selected. Property φ will either be satisfied or violated by the generated system, depending on the desired solution. Structural properties are always satisfied in M as their intend is to imbue additional meaningful temporal behavior into an otherwise randomly generated system. These structural properties can for example be the result of property mining on real-world systems (see also $[\text{JMM}^+19]_{\text{AP}}$). During this step, it is furthermore possible to enforce an alternation between input and output symbols in order to ease a later

property-preserving transformation to a Mealy machine.

In order to transform logical formulas into transition-based systems, LTL synthesis is employed to generate a corresponding Büchi automaton (see Section 3.3).

5.2 Rarely Occurring Errors

As an approach to generate positive verification tasks $V(M, \varphi)$, one can combine φ and chosen structural properties Φ by means of conjunction in order to synthesize a corresponding Büchi automaton B with $\mathcal{L}(B) = [\![\varphi \wedge \Phi]\!]$. A naive analogous procedure for negative verification tasks would be to simply omit φ during synthesis: as long as the formula $\neg\varphi \wedge \Phi$ is satisfiable, there will exist a counterexample to φ in the generated Büchi automaton B' with $\mathcal{L}(B') = [\![\Phi]\!]$. This way, $\mathcal{L}(B')$ can contain words that satisfy φ and some that violate φ, however in an unknown ratio. In order to generate tasks with *rare* counterexamples or even CE-handles (Def. 33), a different approach is thus required. The following high-level construction sketch is such an approach and inserts a counterexample for φ with CE-handle into a Büchi automaton that satisfies φ $[\text{JSSS21}]_{\text{AP}}$:

1. Synthesize a Büchi automaton B with language $\mathcal{L}(B) = [\![\varphi]\!]$.
2. Choose a counterexample lasso h with $w(h) \in [\![\neg\varphi]\!]$.
3. Merge h into B while heuristically aiming for a long shared prefix between h and B.
4. The first transition after this shared prefix is then a CE-handle for φ.

Note that the lasso needed for the second step can be retrieved by choosing an accepting word from a synthesized Büchi automaton B' with $\mathcal{L}(B') = [\![\neg\varphi]\!]$. Details of a heuristic to obtain a long shared prefix for LTL properties that feature a safety part (see Def. 13)—including an example—are given in $[\text{JS18}]_{\text{AP}}$.

5.3 Deeply Hidden Errors

This section presents a method that is introduced in $[\text{HJM}^+21]_{\text{AP}}$ to generate $(m, n]$-deep tasks as defined in Def. 35 for LTL properties that feature a safety part. A notion of an abstract verification task whose model is a language of infinite words or a Büchi automaton is used in this section. Note that such models again entail two-valued model checking as opposed to the three-valued version that has to be used when inspecting the behavior of MTSs. The following presentation is based on $[\text{HJM}^+21]_{\text{AP}}$.

Given some alphabet Σ and a language $\mathcal{L} \subseteq \Sigma^\omega$, one can deduce a constructive approach to generate the maximal sub-language of $\mathcal{L}$ that is $(m, n]$-deep w.r.t. φ.

5.3.1 Language Manipulation

First, the maximal sub-language $\mathcal{L}_\varphi^m$ of $\mathcal{L}$ that does *not* m-violate φ (Def. 34) is constructed before it is then checked whether or not $\mathcal{L}_\varphi^m$ n-violates φ. If it does, $V(\mathcal{L}_\varphi^m, \varphi)$

is an $(m, n]$-deep verification task. Otherwise, it follows that no $(m, n]$-deep verification task exists for $\mathcal{L}$ and φ, in which case one can continue by heuristically modifying the parameters.

The remainder of this section is therefore dedicated to the construction of $\mathcal{L}_\varphi^m$ and the subsequent check whether it n-violates φ.

Definition 41 (Violating Prefixes)**.** *Let $\mathcal{L} \subseteq \Sigma^\omega$ and $k \in \mathbb{N}$. Given a $\varphi \in$ LTL, $\mathrm{VP}(\mathcal{L}, \varphi, k) := \mathcal{L}_{\leq k} \setminus [\![\varphi]\!]_{\leq k}$ are the* violating prefixes *of φ in $\mathcal{L}$ with length at most k.*

It is straightforward to prove that the above characterization is correct (see Def. 34):

Proposition 12 (Correct Definition of Violating Prefixes)**.** *Let $k \in \mathbb{N}$. Then $\mathrm{VP}(\mathcal{L}, \varphi, k)$ consists of all words $w \in \mathcal{L}_{\leq k}$ that violate φ.*

The following theorems follow straightforwardly from Propositions 11 and 12:

Theorem 3. $\mathcal{L}_\varphi^m = \mathcal{L} \setminus (\mathrm{VP}(\mathcal{L}, \varphi, m)\Sigma^\omega)$

Theorem 4. *$\mathcal{L} \subseteq \Sigma^\omega$ n-violates φ iff* $\mathrm{VP}(\mathcal{L}, \varphi, n) \neq \emptyset$

Complementation of Büchi automata is a very expensive operation. The following theorem guarantees that this operation can be avoided and instead be replaced by one that executes in quadratic time:

Theorem 5. $\mathcal{L}_\varphi^m = \mathcal{L} \cap ([\![\varphi]\!]_{\leq m}\Sigma^\omega)$

A proof of Theorem 5 can be found in [HJM+21]$_{\mathrm{AP}}$. The next section presents the Büchi automaton-based realization of $\mathcal{L}_\varphi^m$ in the way that it is used in this thesis to generate verification tasks.

5.3.2 Realization using Büchi Automata

As discussed in Section 5.1, the initial languages $\mathcal{L}$ used to generate verification tasks are of the form $\mathcal{L} = [\![\Phi]\!]$ where Φ is a set of structural LTL properties. Thus, the goal is to construct $\mathcal{L}_\varphi^m = [\![\Phi]\!]_\varphi^m$. According to Theorem 5 this means that one has to compute

$$\mathcal{L}' = [\![\Phi]\!] \cap ([\![\varphi]\!]_{\leq m}\Sigma^\omega).$$

This can be done by means of well-known technology for Büchi automata as follows:

1. Compute $\mathcal{L} = [\![\Phi]\!]$ and $[\![\varphi]\!]$. The Spot library [DLLF+16] is used for this purpose.
2. Concatenate the prefix tree of depth m for $[\![\varphi]\!]$ with Σ^ω to obtain a Büchi automaton for $[\![\varphi]\!]_{\leq m}\Sigma^\omega$. Essentially, this means to add an accepting Σ^ω-loop at the end of each leaf of this prefix tree.
3. Compute the intersection of the two Büchi automata constructed in steps 1 and 2. This is again accomplished using the Spot library.

4. Heuristically minimize the Büchi automaton that results from step 3, again based on the Spot library. This is important for the scalability of later transformation steps and helps to obfuscate the tree expansion in step 2.

In order to be sure that $(\mathcal{L}', \varphi)$ is indeed an $(m, n]$-deep verification task, it remains to be shown that $\mathcal{L}'$ n-violates φ (cf. Def. 35). This can be done simply by means of an emptiness check for

$$\mathcal{L}' \setminus (\llbracket\varphi\rrbracket_{\leq n}\Sigma^{\omega})$$

If it fails, a violating prefix that is longer than m but shorter than or equal to n is guaranteed to exist. Otherwise, it follows that no $(m, n]$-deep verification task exists for $\llbracket\Phi\rrbracket$ and φ, and one continues by heuristically modifying the parameters. An example of using this approach and a corresponding scalability study can be found in [HJM+21]$_{\text{AP}}$.

5.4 Transformation to an MTS

As the model expansion that will be detailed in Section 6 works on MTSs, a property-preserving transformation from a Büchi automaton B to an MTS M is required. The following paragraphs briefly summarize such a transformation. Details—including an example—can be found in [JS18]$_{\text{AP}}$.

An important step towards an MTS is to discard the acceptance condition of B: an MTS does not possess such a condition because all infinite paths within an MTS are relevant for the validity of an LTL property (see Def. 14). Therefore, to preserve LTL properties, all infinite paths π in B with $w(\pi) \notin \mathcal{L}(B)$ have to be rendered infeasible during the transformation to an MTS. This is achieved by removing a transition on every non-accepting simple cycle[1] in B. Note that for the result B' of this transition pruning, it holds that $\mathcal{L}(B') \subseteq \mathcal{L}(B)$.

Afterwards, all transitions in B' are perceived as may transitions, before every counterexample path that should be preserved during later transformations is added as must transitions. For a negative verification task $V(M, \varphi)$, only one counterexample path that violates φ needs to be preserved. It is thus always possible to 'protect' transitions on such a counterexample path when removing transitions during the previous step. In [JS18]$_{\text{AP}}$, a method is presented that furthermore accomplishes to preserve multiple, so called *orthogonal* counterexamples.

[1] Simple cycles are also called elementary circuits [Joh75].

6 Generating Parallel Verification Tasks

The synthesis of verification tasks that target parallel programs is a key aspect of this thesis, especially that of hard parallel tasks [JSSS21]$_{\text{AP}}$. This chapter presents new contributions regarding the corresponding transformation from an MTS to a parallel composition of MTSs (see Figure 2.2). This transformation revolves around iterated, modal contract-based, and property-preserving parallel decompositions as illustrated in Figure 6.1.

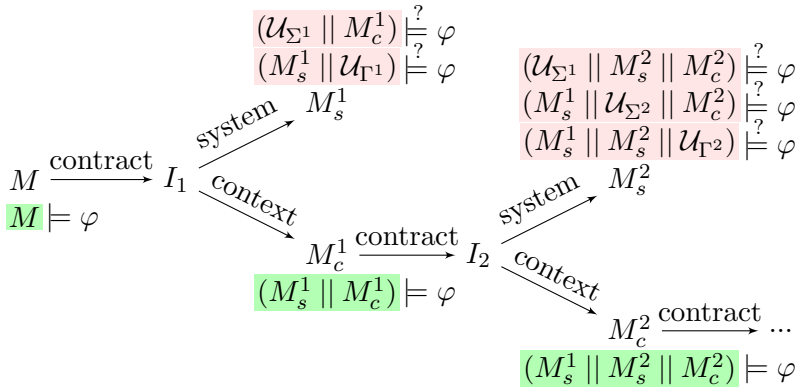

Figure 6.1: Sketch of iterated parallel decompositions during the generation of a hard parallel task. A satisfied property cannot be verified if one abstracts from an entire component [SJ17, JSSS21]$_{\text{AP}}$.

An important part of this contract-based decomposition is the generation of admissible context components. Section 6.1 presents an improved generation of these contexts compared to the approaches of [SJ17]$_{\text{AP}}$ and [Jas18]. That section furthermore discusses the coarseness of generated contexts in terms of modal refinement and provides corresponding proofs for both correctness and coarseness.

In addition to generating additional parallel components, the concept of alphabet

extension is employed in order to achieve interleaving-based state explosion. Besides (convergent weak) modal refinement itself, this is the main method to enlarge the state space of a generated parallel system. Section 6.2 contributes a new result by lifting the notion of alphabet extension to a semantic level reminiscent of contracts. This is accomplished by extending the concept of bisimulation to MTSs while viewing may and must transitions as different entities.

6.1 Property-Preserving Parallel Decomposition

Modal contracts $[\text{SJ17}]_{\text{AP}}$ have been introduced in order to decompose an MTS into two parallel ones, called system and context, respectively, such that temporal properties are preserved by their parallel composition. These contracts are a specific type of assume-guarantee contracts [GL94, RBB$^+$11, BDH$^+$12, BC17, BCN$^+$18]. Intuitively speaking, transitions that are feasible in the resulting parallel composition based on synchronization are colored green, whereas transitions that the system supports, however the context prohibits, are colored red.

Formally, modal contracts are reminiscent of MTS quotient problems [LX90, BDF$^+$13], however with a constrained synchronization alphabet. Given two MTS M and M_s, the quotient M/M_s describes the coarsest MTS M_c w.r.t. modal refinement such that $M_s \parallel M_c \lesssim M$ holds. For a property-preserving parallel decomposition, it suffices to select *some* MTS $M_c' \lesssim M_c$. Because parallel tasks constructed in this thesis consist of compositions of LTSs and hence do not feature modalities, the sole purpose of a coarse MTS during a corresponding generation is to be flexible w.r.t. a final implementation.

6.1.1 Green Contracts

The following definition introduces green contracts as a subset of modal contracts without red transitions $[\text{JSSS21}]_{\text{AP}}$.

Definition 42 (Green Contract)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS and $\Gamma \subseteq \Sigma$. The* green contract *(GC) $I = (M, \Gamma)$ specifies a set of context MTSs $\mathcal{M}_c(I)$ such that for every $M_c \in \mathcal{M}_c(I)$, it holds that $\Sigma(M_c) = \Gamma$ and $M \parallel M_c \lesssim M$. Furthermore, $G(I) := \{s \xrightarrow{a} s' \mid a \in \Gamma\}$ and transitions of $G(I)$ are colored green. The GC I is deterministic iff M is deterministic.*

In the context of parallel decomposition, M is referred to as the *system* of I. Intuitively speaking, a green contract specifies a set of must transitions for which a corresponding context component always has to guarantee synchronization.

Definition 43 (Coarsest Context)**.** *For any GC I, the maximal element of $\mathcal{M}_c(I)$ w.r.t. modal refinement (if existing) is called* coarsest context *of I.*

Note that in general, MTSs are not closed under the quotient operation [BDF$^+$13], and the question of whether a coarsest GC context always exists is left open for future research here. In the following, a construction of a context is shown which is the coarsest

context for deterministic GCs. To ease the description of that construction, the following concept of may-completion is used.

Definition 44 (May-Completion)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS. Then M is* may-complete *iff*

$$T := \{(s, a) \mid \nexists s' \in S : s \overset{a}{\dashrightarrow} s'\} = \emptyset.$$

The may-completion *M' of M is defined as $M = (S \uplus \{s_\Sigma\}, s_0, \Sigma, \dashrightarrow', \longrightarrow)$ where s_Σ is a new 'sink' state not in S and*

$$\dashrightarrow' := \dashrightarrow \uplus \{s \overset{a}{\dashrightarrow} s_\Sigma \mid (s, a) \in T\} \uplus \{s_\Sigma \overset{a}{\dashrightarrow} s_\Sigma \mid a \in \Sigma\}.$$

The concepts of may-completion and alphabet views allow to define a specific context $M_c^g(I)$ of a GC I by utilizing traditional operations based on finite automata.

Definition 45 (Green Context Construction)**.** *Let $I = (M, \Gamma)$ be a GC that contains the MTS $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$. Then $M_c^g(I)$ is defined as the result of the following construction based on the may-completion M' of M:*

1. *Regard $[M']_\Gamma$ as a prefix-closed τ-NFA N by disregarding must transitions and viewing all states as accepting.*
2. *Determinize N using the traditional powerset construction, resulting in a DFA D.*
3. *Transform D to the MTS $M_c^g(I)$ by*
   a) *disregarding the acceptance condition,*
   b) *viewing all transitions as may transitions, and*
   c) *afterwards adding the following set of must transitions:*

$$\{P \xrightarrow{a} P' \mid P, P' \text{ are states in } D \ \wedge \ \exists p \in P, p' \in P' : \ p \xrightarrow{a} p'\}$$

The just-presented construction always yields a context:

Lemma 3 (Green Context Admissibility)**.** *For any GC I, $M_c^g(I) \in \mathcal{M}_c(I)$ holds.*

Proof. Let all identifiers be defined as in Def. 45 and $M_c^g(I) = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \longrightarrow_c)$. Given the parallel composition $M_{pc} := (M \,||\, M_c^g(I)) = (S_{pc}, s_0^{pc}, \Sigma, \dashrightarrow_{pc}, \longrightarrow_{pc})$, it needs to be shown that $M_{pc} \lesssim M$ holds. It is therefore proven that the following relation based on the concept that a state of M is contained in each corresponding state of $M \,||\, M_c^g(I)$,

$$\lesssim_r := \{(p, q) \mid \exists p_c \in S_c : \ p = (q, p_c) \ \wedge \ p \text{ is reachable in } M_{pc}\} \subseteq S_{pc} \times S,$$

is a refinement with $s_0^{pc} \lesssim_r s_0$. The latter obviously holds because $s_0^{pc} = (s_0, s_0^c)$ according to the definition of parallel composition. Let (p, q) with $p = (q, p_c)$ be an arbitrary but fixed element of $\lesssim_r$. Both conditions in the definition of modal refinement are satisfied:

Case 1: Let $p \overset{a}{\dashrightarrow}_{pc} p'$ with $p' = (q', p_c')$ be an arbitrary outgoing may transition of p in M_{pc}. Based on the definition of parallel composition, there exists a transition $q \overset{a}{\dashrightarrow} q'$ because $\Gamma \subseteq \Sigma$ holds. Obviously, $p' = (q', p_c') \lesssim_r q'$ follows.

Case 2: Let $q \xrightarrow{a} q'$ be an arbitrary outgoing must transition of q. If $a \notin \Gamma$, then component M of M_{pc} will proceed alone according to the definition of parallel composition, i.e., $(q, p_c) \xrightarrow{a}_{pc} (q', p_c)$, and nothing remains to be shown because $p' \lesssim_r q'$. If $a \in \Gamma$, then all that remains to be shown is that there exists a transition $p_c \xrightarrow{a}_c p'_c$ for some $p'_c \in S_c$.

Note that because of the powerset construction to create $M_c^g(I)$ (step 2 in Def. 45) and because a reachable state in a parallel composition requires a common access sequence to its elements, $q \in p_c \subseteq S$ holds. Again due to the powerset construction, $q \in p_c$ and the existence of $q \xrightarrow{a} q'$ imply that there has to exist a state $p'_c \in S_c$ with $p_c \overset{a}{\dashrightarrow}_c p'_c$ and $q' \in p'_c$. Because of $p \in q$, $p' \in q'$, there also exists a transition $p_c \xrightarrow{a}_c p'_c$ according to step 3.c during the construction of $M_c^g(I)$. Thus, $p' \lesssim_r q'$.

Because $\lesssim_r \subseteq S_{pc} \times S$ is a refinement with $s_0^{pc} \lesssim_r s_0$, $M_{pc} = M \,||\, M_c^g(I) \lesssim M$ holds. □

The MTS $M_c^g(I)$ is the coarsest context if I is deterministic. In order to show this, it helps to first understand that the following holds:

Lemma 4 (Unmatched Transition if not Refining)**.** *Let $M_1 = (S_1, s_0^1, \Sigma_1, \dashrightarrow_1, \longrightarrow_1)$, $M_2 = (S_2, s_0^2, \Sigma_2, \dashrightarrow_2, \longrightarrow_2)$, be two MTSs with $M_1 \not\lesssim M_2$. Then there exist states $p \in S_1$, $q \in S_2$ such that at least one of the following holds:*

1. *There exists a transition $p \overset{a}{\dashrightarrow}_1 p'$, however no transition $q \overset{a}{\dashrightarrow}_2 q'$*
2. *There exists a transition $q \xrightarrow{a}_2 q'$, however no transition $p \xrightarrow{a}_1 p'$*

Proof. $M_1 \not\lesssim M_2$ means that no refinement relation $\lesssim' \subseteq S_1 \times S_2$ with $s_0^1 \lesssim' s_0^2$ exists. This implies that the following relation based on existing common access sequences[1]

$$\lesssim_a := \{(p, q) \mid \text{access}(p, M_1) \cap \text{access}(q, M_2) \neq \emptyset\} \subseteq S_1 \times S_2$$

is *not* a refinement. Therefore, there exist states $p \in S_1, q \in S_2$ which are reachable via the same label sequence in their respective MTS, however such that they violate the first or second condition in the definition of modal refinement (Def. 18). Due to the definition of $\lesssim_a$, this violation is directly based on the lack of a matching transition because continuing with the same label from both p and q would again result in a common access sequence to their successor states. □

Lemma 4 allows to prove that the desired coarseness for deterministic GCs is ensured:

Lemma 5 (Coarsest Context of a Deterministic GC)**.** *For any deterministic GC I, $M_c^g(I)$ is the coarsest context of I.*

Proof. Let all identifiers be defined as in Definition 45. Assume that the constructed MTS $M_c := M_c^g(I) = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \longrightarrow_c)$ is not the coarsest context of I. Then there exists an MTS $M_f = (S_f, s_0^f, \Gamma, \dashrightarrow_f, \longrightarrow_f)$ with $M_f \in \mathcal{M}_c(I)$ and $M_f \not\lesssim M_c$. Let $p \in S_f$, $q \in S_c$ be the states that exist according to Lemma 4 because of $M_f \not\lesssim M_c$.

[1] These access sequences have been defined in Def. 4.

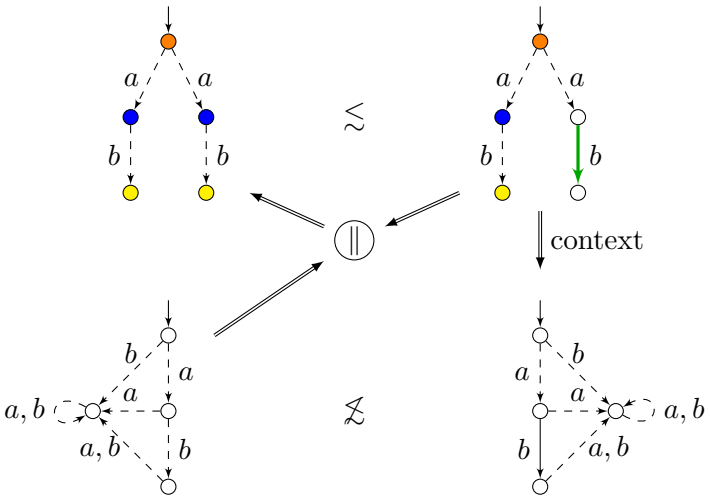

Figure 6.2: Example of a GC $I = (M, \{a, b\})$ (top right) for which $M_c^g(I)$ (bottom right) is not the coarsest context. A truly coarser context M_c exists (bottom left). A refinement between the corresponding composition $M \,||\, M_c$ (top left) and M is illustrated through colored states.

Case 1: There exists a transition $p \overset{a}{\dashrightarrow}_f p'$, however no transition $q \overset{a}{\dashrightarrow}_c q'$. This is a contradiction to the fact that M_c is may-complete.

Case 2: There exists a transition $q \overset{a}{\longrightarrow}_c q'$, however no transition $p \overset{a}{\longrightarrow}_f p'$. Because of $q \overset{a}{\longrightarrow}_c q'$, there has to exist a must transition $s \overset{a}{\longrightarrow} s'$ in M with $s \in q \subseteq S$ because of steps 2 (powerset construction) and 3.c during the construction of M_c.

Let $w \in \text{access}(p, M_f) \cap \text{access}(q, M_c)$. For every state $s'' \in q \subseteq S$, the subset relation $\text{access}(q, M_c) \subseteq \text{access}(s'', [M]_\Gamma)$ holds due to the powerset construction used to create M_c. Thus, $w \in \text{access}(s, [M]_\Gamma)$ follows.

In combination with the fact that $\Gamma \subseteq \Sigma$ holds, $w \in \text{access}(s, [M]_\Gamma)$ implies that the parallel composition $M \,||\, M_f$ contains a reachable state (s, p) that does *not* feature an outgoing must transition labeled a, whereas M itself does at state s.

Because M is deterministic and due to Proposition 3, $(s, p) \lesssim' s$ has to hold for *any* refinement $\lesssim'$ with $s_0^{pc} \lesssim' s_0$. Thus, $M \,||\, M_f \not\lesssim M$ follows, contradicting the assumption that $M_f \in \mathcal{M}_c(I)$.

As each case results in a contradiction, $M_c^g(I)$ has to be the coarsest context of I. □

In case of a non-deterministic GC I, $M_c^g(I)$ is *not* always the coarsest context, as illustrated in Figure 6.2. Intuitively, non-deterministic branching enables choice when trying to find a refinement relation, and a successful choice cannot be identified locally.

The determinization of an NFA can in general produce a DFA that is exponentially larger, hence this step in the construction of $M_c^g(I)$ is a limiting factor for its efficiency.

Corollary 1 (Efficiency of Green Context Construction)**.** *The efficiency of constructing the coarsest context of a GC $I = (M, \Gamma)$ depends on the efficiency of the traditional powerset determinization of $[M]_\Gamma$.*

6.1.2 Red Contracts

In order to fully harness the potential of the presented decomposition approach, modal contracts that include both green and red transitions are utilized. The following presentation focuses on the red aspect of such a contract, before both green and red transitions are employed thereafter.

Definition 46 (Modal Contract)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS, $\Gamma \subseteq \Sigma$, $R \subseteq S \times \Gamma \times S$ with $R \cap \dashrightarrow = \emptyset$, and $M_s := (S, s_0, \Sigma, \dashrightarrow \cup R, \longrightarrow \cup R)$. Then the tuple $I = (M, \Gamma, R)$ is a* modal contract (MC) *of M with communication alphabet Γ iff*

$$\mathcal{M}_C(I) := \{\ M_c\ \mid\ M_s \,||\, M_c \lesssim M\ \wedge\ \Sigma(M_c) = \Gamma\ \} \neq \emptyset.$$

Moreover, $G(I) := \{s \xrightarrow{a} s' \mid a \in \Gamma\}$. Transitions of $G(I)$ are colored green and transitions of R red. The MTS $M_s(I) := M_s$ is called the system of I and any $M_c \in \mathcal{M}_C(I)$ a context of I. MC I is deterministic iff M is deterministic, and I is also called red contract *iff $G(I) = \emptyset$.*

The coarsest context of an MC I is defined analogously to that of a GC (Def. 43). A path in I is allowed to traverse red transitions. For any alphabet Σ', the view $[I]_{\Sigma'}$ propagates to its MTS and is equivalent to $[M]_{\Sigma'}$ with the additional transitions[2] in R.

Intuitively speaking, (only) green transitions enforce the presence of certain must transitions in a context $M_c \in \mathcal{M}_c(I)$—just like they do in a GC—whereas (only) red transitions prohibit the existence of some may transitions in M_c. Note that for the attribute of being deterministic, red transitions in an MC are irrelevant.

If there exist no green transitions, then an empty MTS without any transitions is always an admissible context. Therefore, the following sufficient condition can be stated.

Proposition 13 (Red Contract Always Exists)**.** *Tuple I in Def. 46 is an MC if $G(I) = \emptyset$.*

Because an MTS without transitions is generally not the coarsest context, the following construction is used [SJ17]$_{\text{AP}}$.

Definition 47 (Red Context Construction)**.** *Let $I = (M, \Gamma, R)$ be an MC, $\mathcal{L}_R$ the language of words $w \in \Gamma^*$ for which a path in $[I]_\Gamma$ exists that contains a red transition $t \in R$, and D the minimal DFA that describes the prefix-closed language $\Gamma^* \setminus \mathcal{L}_R$. Then $M_c^r(I)$ is defined as the result of the following construction based on D:*

[2] Labels of transitions in R are also affected by the applied view. Within this chapter, such an effect does however not occur because the communication alphabet Γ is viewed and $\Sigma(R) \subseteq \Gamma$ holds.

1. *Remove all incoming and outgoing transitions of the unique non-accepting sink state together with this sink state itself.*
2. *Consider all remaining transitions as may transitions.*
3. *Disregard the acceptance condition.*

The construction of $M_c^r(I)$ always produces a context for red contracts:

Lemma 6 (Red Context Admissibility)**.** *For any MC I with $G(I) = \emptyset$, $M_c^r(I) \in \mathcal{M}_c(I)$ holds.*

Proof. Let all identifiers be defined as in Def. 47 with $M_s(I) = (S_s, s_0^s, \Sigma, \dashrightarrow_s, \longrightarrow_s)$ and $M_c := M_c^r(I) = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \longrightarrow_c)$. It needs to be shown that $M_{pc} \lesssim M$ holds where $M_{pc} := (M_s(I) \,||\, M_c^r(I)) = (S_{pc}, s_0^{pc}, \Sigma, \dashrightarrow_{pc}, \longrightarrow_{pc})$. It is therefore proven that

$$\lesssim_r := \{(p, q) \mid \exists p_c \in S_c : \; p = (q, p_c) \;\wedge\; p \text{ is reachable in } M_{pc}\} \subseteq S_{pc} \times S$$

is a refinement with $s_0^{pc} \lesssim_r s_0$. The latter obviously holds. Let (p, q) with $p = (q, p_c)$ be an arbitrary but fixed element of $\lesssim_r$. Both conditions of modal refinement are satisfied:

Case 1: Let $p \overset{a}{\dashrightarrow}_{pc} p'$ with $p' = (q', p_c')$ be an arbitrary outgoing may transition of p. If $q \overset{a}{\dashrightarrow}_s q'$ also exists in $\dashrightarrow$, then there is nothing to show because $\Gamma \subseteq \Sigma$ and thus $p' \lesssim_r q'$. Otherwise, $q \overset{a}{\dashrightarrow}_s q'$ would have to be a red transition in R. This is however a contradiction to the construction of M_c: because q' and p_c' are reachable via the same access sequence when only viewing symbols in Γ, $\mathcal{L}_\top(M_c)$ would contain a word for which a path exists in $[I]_\Gamma$ that traverses a red transition.

Case 2: Let $q \overset{a}{\longrightarrow} q'$ be an arbitrary outgoing must transition of q. It follows that $a \notin \Gamma$ because otherwise, there would exist a green transition in I. Therefore, component $M_s(I)$ of M_{pc} will proceed alone, i.e., $(q, p_c) \overset{a}{\longrightarrow}_{pc} (q', p_c)$, and nothing remains to be shown because $M_s(I)$ contains all must transitions of M and thus $p' \lesssim q'$ holds.

As $\lesssim_r \subseteq S_{pc} \times S$ is a refinement with $s_0^{pc} \lesssim_r s_0$, $M_{pc} = M_s(I) \,||\, M_c^r(I) \lesssim M$ holds. □

In contrast to the presented context construction for green contracts, $M_c^r(I)$ is the coarsest context also for non-deterministic red contracts.

Lemma 7 (Coarsest Red Context)**.** *For any MC I with $G(I) = \emptyset$, $M_c^r(I)$ is the coarsest context of I.*

Proof. Let all identifiers be defined as in Def. 47. Assume that the constructed MTS $M_c := M_c^r(I) = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \longrightarrow_c)$ is not the coarsest context of I. Then there exists an MTS $M_f = (S_f, s_0^f, \Gamma, \dashrightarrow_f, \longrightarrow_f)$ with $M_f \in \mathcal{M}_c(I)$ and $M_f \not\lesssim M_c$. Let $p \in S_f$, $q \in S_c$ be the states that exist according to Lemma 4 because of $M_f \not\lesssim M_c$.

Case 1: There exists a transition $p \overset{a}{\dashrightarrow}_f p'$, however no transition $q \overset{a}{\dashrightarrow}_c q'$. Choose any $w \in \text{access}(p, M_f) \cap \text{access}(q, M_c)$. Due to the construction of M_c, there exists a path π in $[I]_\Gamma$ with word $w(\pi) = wa$ that traverses a red transition in R. Let $s \in S$ be the state in which this red transition starts.

Because $M_s(I)$ contains all transitions of M and also red transitions in R, it follows that $wa \in [\mathcal{L}_\top(M_s(I) \parallel M_f)]_\Gamma$. As $w \in [\text{access}(s, M)]_\Gamma$ holds and because M and $M_s(I)$ are identical except for red transitions, there exist reachable states (s, p) in $M_s(I) \parallel M_f$ and s in M such that (s, p) has an outgoing transition labeled a whereas s does not. Thus, $M_s(I) \parallel M_f \not\preceq M$ holds, contradicting the assumption that M_f is a context of I.

Case 2: There exists a transition $q \overset{a}{\longrightarrow}_c q'$, however no transition $p \overset{a}{\longrightarrow}_f p'$. This is impossible because M_c does not contain must transitions.

As each case results in a contradiction, $M_c^r(I)$ has to be the coarsest context of I. □

The following efficiency statement is analogous to that of Corollary 1.

Corollary 2 (Efficiency of Red Context Construction)**.** *Given an MC $I = (M, \Gamma, R)$ with $G(I) = \emptyset$, the efficiency of constructing the coarsest context of I depends on an efficient determinization of $[I]_\Gamma$.*

6.1.3 Modal Contracts

Given context constructions for both green and red contracts, MTS *conjunction* is used to combine these approaches into a context construction for any modal contract $[\text{SJ17}]_{\text{AP}}$.

Definition 48 (MTS Conjunction)**.** *Given two MTSs $M_1 = (S_1, s_0^1, \Sigma_1, \dashrightarrow_1, \longrightarrow_1)$ and $M_2 = (S_2, s_0^2, \Sigma_2, \dashrightarrow_2, \longrightarrow_2)$, the* conjunction

$$M_1 \wedge M_2 =_{def} (S_1 \times S_2, (s_0^1, s_0^2), \Sigma, \dashrightarrow, \longrightarrow)$$

of M_1 and M_2 is then defined as a commutative and associative operation satisfying the following operational rules:[3]

$$\frac{p \overset{a}{\longrightarrow}_1 p' \quad q \overset{a}{\dashrightarrow}_2 q'}{(p,q) \overset{a}{\longrightarrow} (p',q')} \qquad \frac{p \overset{a}{\dashrightarrow}_1 p' \quad q \overset{a}{\dashrightarrow}_2 q'}{(p,q) \overset{a}{\dashrightarrow} (p',q')} \qquad \frac{p \overset{a}{\longrightarrow}_1 p' \quad q \not\overset{a}{\dashrightarrow}_2 q'}{error}$$

Whenever an error occurs, the conjunction of M_p and M_q is undefined.

Intuitively speaking, MTS conjunction gives both must transitions and non-existing transitions precedence over may transitions. Conjunction guarantees that a refining MTS refines both components $[\text{SJ17}]_{\text{AP}}$:

[3] This definition depends on the fact that each must transition is also a may transition.

Proposition 14 (Conjunction of Refinement Constraints)**.** *Let M, M', M'' be three MTSs. If $M' \wedge M''$ is defined, then the following holds:*

$$M \lesssim M' \wedge M'' \quad \textit{iff} \quad M \lesssim M' \text{ and } M \lesssim M''$$

Definition 49 (Context Construction)**.** *Let $I = (M, \Gamma, R)$ be an MC and $I_g = (M, \Gamma)$ the corresponding GC. Then $M_c(I) := M_c^g(I_g) \wedge M_c^r(I)$.*

Note that despite the monotonicity of composition w.r.t. refinement (Proposition 8), Proposition 14 does not immediately imply that $M_c(I)$ is a context because M and $M_s(I_r)$ can be different. Nonetheless, the above definition of $M_c(I)$ yields an admissible context.

Theorem 6 (Context Admissibility)**.** *For any MC I, $M_c(I)$ is defined and it holds that $M_c(I) \in \mathcal{M}_c(I)$.*

Theorem 6 can be shown straightforwardly by combining the proofs of Lemmas 3 and 6 because must transitions and non-existing transitions from conjunct MTSs are propagated to their conjunction. A corresponding proof is therefore omitted here.

The construction of $M_c(I)$ based on conjunction produces a coarsest context in case that I is deterministic. In order to show this, it is helpful to first inspect the *compositionality* of green and red transitions in a coarsest context, as done in the following. Because red transitions in an MC $I = (M, \Gamma, R)$ are disjoint from may transitions in M, they never affect the reachability of green transitions in $G(I)$. In combination with the fact that the coarsest context of a GC is may-complete, it is apparent that, given the coarsest context of I, one can extend it to that of the GC (M, Γ) by filling in missing may transitions that were prohibited by R. Similarly, one can retrieve the coarsest context for only the red "aspect" of I by simply omitting all must transitions. The following proposition formally summarizes these statements.

Proposition 15 (Coarsest Context Decomposition)**.** *Let $M_c = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \longrightarrow_c)$ be the coarsest context of an MC $I = (M, \Gamma, R)$. Then*

1. *The may-completion of M_c is the coarsest context of the GC $I = (M, \Gamma)$, and*
2. *the MTS $M_c^r = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \emptyset)$ is the coarsest context of the MC $I_r = (M_r, \Gamma, R)$ with $M_r = (S, s_0, \Sigma, \dashrightarrow, \emptyset)$.*

By relying on Proposition 15, one can show that MTS conjunction preserves the attribute of being the coarsest context:

Theorem 7 (Coarsest Context of a deterministic MC)**.** *For any deterministic MC I, $M_c(I)$ is the coarsest context of I.*

Proof. Let all identifiers be defined as in Def. 49. Assume that the constructed MTS $M_c := M_c(I) = (S_c, s_0^c, \Gamma, \dashrightarrow_c, \longrightarrow_c)$ is not the coarsest context of I. Then there exists an MTS $M_f = (S_f, s_0^f, \Gamma, \dashrightarrow_f, \longrightarrow_f)$ with $M_f \in \mathcal{M}_c(I)$ and $M_f \not\lesssim M_c$. Let $p \in S_f$, $q \in S_c$ be the states that exist according to Lemma 4 because of $M_f \not\lesssim M_c$. This time, state q itself is a pair $q = (q_g, q_r)$ of states in $M_c^g(I)$ and $M_c^r(I)$, respectively.

Case 1: There exists a transition $p \overset{a}{\dashrightarrow}_f p'$, however no transition $q \overset{a}{\dashrightarrow}_c q'$. Because every state in $M_c^g(I_g)$ features an outgoing may transition for every label $a \in \Gamma$, it follows that q_r does not have an outgoing transition labeled a due to the definition of MTS conjunction.

The MTS $M_f^r = (S_f, s_0^f, \Gamma, \dashrightarrow_f, \emptyset)$ however contains the transition $p \overset{a}{\dashrightarrow}_f p'$, which implies that $M_f^r \not\preceq M_c^r(I)$. Because M_f^r is the coarsest context of $I_r = (M_f^r, \Gamma, R)$ according to Proposition 15.2, this contradicts the fact that $M_c^r(I)$ is the coarsest context of I_r according to Lemma 7.

Case 2: There exists a transition $q \overset{a}{\longrightarrow}_c q'$, however no transition $p \overset{a}{\longrightarrow}_f p'$. Because $M_c^r(I)$ does not contain must transitions, state q_g has to feature an outgoing must transition labeled a in $M_c^g(I_g)$, again based on the definition of MTS conjunction.

By definition, the may-completion M_f^g of M_f contains the same set of must transitions as M_f itself. Therefore, M_f^g also lacks an outgoing must transition labeled a at state p, which implies that $M_f^g \not\preceq M_c^g(I_g)$. Because M_f^g is the coarsest context of I_g according to Proposition 15.1, this contradicts the fact that $M_c^g(I_g)$ is the coarsest context of I_g.

As each case results in a contradiction to the definitions of $M_c^r(I)$ and $M_c^g(I_g)$, respectively, $M_c(I)$ has to be the coarsest context of I. □

Note that the above proof of Theorem 7 does not rely on determinism and instead only requires the fact that $M_c^g(I_g)$ and $M_c^r(I)$ are coarsest contexts. As a result, MTS conjunction can also be used to produce a coarsest context for non-deterministic contracts in case that a coarsest green context is given.

Because MTS conjunction can only result in a quadratic number of states w.r.t. its arguments, the efficiency of the presented context construction again depends on that of NFA determinization.

Corollary 3 (Efficiency of Context Construction)**.** *The efficiency of constructing the coarsest context of an MC $I = (M, \Gamma, R)$ depends on the efficiency of determinizing $[I]_\Gamma$.*

An example of a context construction can be found in $[\text{SJMvdP17}]_{\text{AP}}$. Theorem 7 has implications for the identification of an MC: it guarantees that a successful construction of context $M_c(I)$ during the MTS conjunction not only serves as a sufficient, but also as a necessary requirement that a deterministic I is an MC.

Corollary 4 (MC Identification)**.** *Let $I = (M, \Gamma, R)$ such that $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ is an MTS, $\Gamma \subseteq \Sigma$, and $R \subseteq S \times \Gamma \times S$ with $R \cap \dashrightarrow = \emptyset$. If $M_c(I)$ exists, then I is an MC. If M is deterministic and $M_c(I)$ does not exist, then I is not an MC.*

6.2 Alphabet Extension

The parallel decomposition based on modal contracts as introduced in the previous section can be used to generate parallel tasks with many components, however does not

scale the reachable state space of the corresponding parallel composition [SJ17]$_{\text{AP}}$. In order to incorporate interleaving-based state explosion into the generation of verification tasks, the concept of *alphabet extension* is used: finite interruptions by new symbols are added through a parallel composition with another MTS such that properties are preserved [SJ17, SJ19, JSSS21]$_{\text{AP}}$.

6.2.1 Extending Bisimulation to Modal Transition Systems

With the goal to capture the property preservation during alphabet extensions precisely, the notion of bisimulation is extended to MTSs in the following. For this extension, an MTS is viewed as an LTS that contains may and must transitions as separate entities.

Definition 50 (LTS View)**.** *Let $M = (S, s_0, \Sigma, \dashrightarrow, \longrightarrow)$ be an MTS and $\mu \notin \Sigma$ a new symbol. Then $L(M) := (S, s_0, \Sigma \uplus \mu\Sigma, \dashrightarrow \uplus \{s \xrightarrow{\mu a} s' \mid s \xrightarrow{a} s'\})$.*

Whenever the special symbol ϵ (see Def. 23) is transformed to $\mu\epsilon$, the latter is perceived as an individual character and not reduced to μ, however both ϵ and $\mu\epsilon$ are still perceived as the empty word. Based on the concept of LTS views, bisimulation can now be extended to MTSs.

Definition 51 ((Convergent, Weakly) Bisimilar MTSs)**.** *Let M, M' be MTSs. M is bisimilar to M', short $M \sim M'$, iff $L(M) \sim L(M')$. Moreover, M is weakly bisimilar to M', denoted as $M \approx M'$, iff $L(\text{obs}(M)) \sim L(\text{obs}(M'))$. Furthermore, M and M' are* convergent and weakly bisimilar*, denoted as $M \overset{c}{\approx} M'$, iff both M and M' are convergent and $M \approx M'$.*

Because both the behavioral structure and the different transition relations are fully preserved when two MTSs are bisimlar, the following proposition follows immediately.

Proposition 16 (Refinement and Individual Bisimulation Implied)**.** *Let M, M' be MTSs with $M \sim M'$. Then $M \lesssim M'$, $M' \lesssim M$, $L_\top(M) \sim L_\top(M')$, and $L_\bot(M) \sim L_\bot(M')$.*

Due to the definition of $\approx$ for MTSs, the above proposition directly implies:

Corollary 5 (Refinement and Individual Bisimulation Implied, Part II)**.** *Let M, M' be two MTSs such that $M \approx M'$. Then $M \lessapprox M'$, $M' \lessapprox M$, $L_\top(M) \approx L_\top(M')$, and $L_\bot(M) \approx L_\bot(M')$. The same hold for the convergent variants.*

Note that as defined above, bisimulation between MTSs is a truly finer equivalence relation than the one induced by MTSs that modally refined each other. Figure 6.3 illustrates a corresponding example.

The following is analogous to Proposition 8, and again holds for both components of a composition.

Proposition 17 (Bisimulation Preservation)**.** *Let M, M', and M'' be MTSs. Then for any $\backsim \in \{\sim, \approx, \overset{c}{\approx}\}$, $\backsim$ is preserved by parallel composition:*

$$M \backsim M' \quad \textit{implies} \quad (M \parallel M'') \backsim (M' \parallel M'')$$

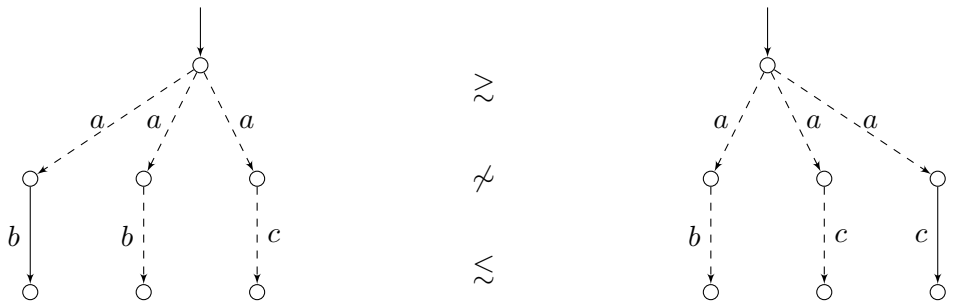

Figure 6.3: Example of two MTSs that modally refine each other, however which are not bisimilar when may and must transitions are viewed as different entities.

6.2.2 (Nonconvergent) Alphabet Extension

The extension of (convergent) weak bisimulation now allows to define alphabet extensions in a requirement-driven fashion akin to the definition of modal contracts in Section 6.1.

Definition 52 (Alphabet Extension)**.** *Let M, M_E be MTSs such that the condition $\Sigma_E := \Sigma(M_E) \setminus \Sigma(M) \neq \emptyset$ is satisfied. Then M_E is called* Σ_E-alphabet extension *(Σ_E-AE) of M iff $[M \mid\mid M_E]_{\Sigma(M)} \overset{c}{\approx} M$. Similarly, M_E is called* nonconvergent Σ_E-alphabet extension *(Σ_E-NAE) of M iff $[M \mid\mid M_E]_{\Sigma(M)} \approx M$.*

A version of alphabet extension similar to the above-defined NAE has been studied in [FBU09]. Because neither the size of M_E nor the size of Σ_E is bound by any means, alphabet extensions allow to incorporate interleaving-based state explosion into a parallel composition, as illustrated by the example in $[\text{SJMvdP17}]_{\text{AP}}$.

If M_E is a Σ_E-alphabet extension of M according to Definition 52, then it is also such an extension according to Definition 24 in $[\text{JSSS21}]_{\text{AP}}$. This is a direct consequence of the following theorem. The above definition is thus an extension of the corresponding one in $[\text{JSSS21}]_{\text{AP}}$ from language equivalences to a relation that also preserves other temporal properties.

Theorem 8 (Preserved Languages)**.** *Let M, M' be MTSs such that $[M']_{\Sigma(M)} \overset{c}{\approx} M$. Then $[\mathcal{L}^{\omega}_{\bot}(M')]_{\Sigma(M)} = \mathcal{L}^{\omega}_{\bot}(M)$ and $[\mathcal{L}^{\omega}_{\top}(M')]_{\Sigma(M)} = \mathcal{L}^{\omega}_{\top}(M)$ hold.*

The proof of Theorem 8 is based on Corollary 5 which ensures that $[M']_{\Sigma(M)} \overset{c}{\approx} M$ implies both $L_{\top}(M) \approx L_{\top}([M']_{\Sigma(M)})$ and $L_{\bot}(M) \approx L_{\bot}([M']_{\Sigma(M)})$. From there on, the proof is analogous to that of Lemma 2 and is therefore omitted here.

The following sufficient condition for identifying (nonconvergent) alphabet extensions can be used for their efficient construction.

Lemma 8 (Sufficient Condition for Alphabet Extensions)**.** *Let M, M_E be MTSs such that $\Sigma_E := \Sigma(M_E) \setminus \Sigma(M) \neq \emptyset$, $\Gamma := \Sigma(M) \cap \Sigma(M_E)$, and $[M_E]_{\Gamma} \approx \mathcal{N}_{\Gamma}$. Then M_E is Σ_E-NAE of M. If $[M_E]_{\Gamma} \overset{c}{\approx} \mathcal{N}_{\Gamma}$ holds, then M_E is a Σ_E-AE of M.*

Proof. Let all identifiers be defined as in Lemma 8 and $\backsim \in \{\approx, \overset{c}{\approx}\}$. Because of Proposition 2, $M = M \,||\, [M_E]_\Gamma$ holds and therefore apparently also $M \backsim M \,||\, [M_E]_\Gamma$. Due to the fact that symbols in $\Sigma(M) \backslash \Gamma$ do not exist in M_E, this yields $M \backsim M || [M_E]_{\Sigma(M)}$. Because of $M = [M]_{\Sigma(M)}$, Proposition 7 thus yields $M \backsim M \,||\, [M_E]_{\Sigma(M)} \backsim [M \,||\, M_E]_{\Sigma(M)}$. □

An automatic and potentially randomized generation of (nonconvergent) alphabet extensions can be accomplished based on the following constructions introduced in $[\text{SJ19}]_{\text{AP}}$ and $[\text{SJ17}]_{\text{AP}}$, respectively.

Theorem 9 (NAE Construction)**.** *Let M be an MTS, Σ_E a new alphabet, meaning $\Sigma_E \cap \Sigma(M) = \emptyset$, and $\Gamma \subseteq \Sigma(M)$. Any MTS $M_E = (S, s_0, \Gamma \uplus \Sigma_E, \dashrightarrow, \longrightarrow)$ is a Σ_E-NAE of M if it adheres to the following two constraints:*

1. *The directed graph $(S, \{(s, s') \mid s \xrightarrow{a} s' \wedge a \in \Sigma_E\})$ is strongly connected*
2. *$\forall a \in \Gamma.\ \exists s, s' \in S :\ s \xrightarrow{a} s'$*

A detailed proof of Theorem 9 is tedious and therefore omitted here. The intuition behind such a proof is that $[M_E]_\Gamma \approx \mathcal{N}_\Gamma$ holds because M_E can never 'block' transitions in M if composed with it due to the following reasons: (i) the alphabet Σ_E only occurs in M_E and (ii) within M_E, one can always reach every state by traversing must transitions with labels from Σ_E. This construction of an NAE is rather unconstrained and allows for a variety of possible choices, as illustrated in $[\text{SJ19}]_{\text{AP}}$.

In order to also preserve convergence and thereby guarantee liveness properties, Floyd-like cut points that enforce an eventual synchronization are used to construct an AE, as presented in $[\text{SJ17, SJMvdP17}]_{\text{AP}}$.

Theorem 10 (AE Construction)**.** *Let M be an MTS, Σ_E a new alphabet, meaning $\Sigma_E \cap \Sigma(M) = \emptyset$, and $\Gamma \subseteq \Sigma(M)$. Any MTS M_E with $\Sigma(M_E) = \Sigma_E$ such that*

i. M_E restricted to its must transitions is free of deadlocks and
ii. each state of M_E is reachable via must transitions

is a Σ_E-AE of M after it has been modified according to the following two steps:

1. *Select a set T of transitions with the property that every infinite path in M_E visits a state in S infinitely often.*
2. *Replace each $s \overset{a}{\dashrightarrow} s' \in T$ (and the corresponding $s \xrightarrow{a} s'$, if existing) by the set of must transitions $\{s \xrightarrow{b} s' \mid b \in \Gamma\}$.*

The correctness of Theorem 10 is based on the fact that $[M_E]_\Gamma \overset{c}{\approx} \mathcal{N}_\Gamma$ holds and can be shown by relying on the two following observations. On the one hand, the Floyd-like interruption of every cycle ensures that $[M_E]_\Gamma$ is convergent, and on the other hand, an outgoing must transition exists for every $a \in \Gamma$ at every state of the corresponding observable MTS $\text{obs}([M_E]_\Gamma)$. Again, a detailed proof is omitted in this thesis.

7 Conclusion and Future Work

This thesis presents a new framework for the synthesis of realistic verification tasks. The associated automatic generation of (software) verification benchmarks features characteristics that are very different from those of a manual benchmark creation. For example, this generation provides an efficient way to ensure that solutions are known, however not necessarily to the public. This characteristic can be beneficial for a meaningful evaluation of verification tools: when attempting to solve real-world problems, the solution is not yet known either. Due to their different profiles, it is likely that manually maintained and automatically synthesized benchmarks will continue to coexist.

This work has led to new ways to use existing synthesis libraries like Spot [DLLF+16] for a generation of automata with (orthogonal) counterexample handles $[\text{JS18}]_{\text{AP}}$ and for a manipulation of these automata to remove short counterexamples $[\text{HJM}^{+}21]_{\text{AP}}$. Moreover, it motivated the implementation of modal contracts—along with the associated parallel decomposition—and alphabet extensions in the AutomataLib [11][1].

These implementations have been constantly extended and applied to automatically generate verification benchmarks for several iterations of the international Rigorous Examination of Reactive Systems (RERS) Challenge. Using these available benchmarks, several participants of RERS have applied many (combinations of) tools to solve the contained verification tasks, and were thereby inspired to conceive new formal verification techniques [SY20, LMM20, LMM19].

This thesis has therefore accomplished its main goals. On the one hand, it provides a framework for the synthesis of realistic verification tasks that are scalable and fulfill realistic hardness criteria. On the other hand, this framework contains a new method for synthesizing tasks that target parallel programs and which realizes parallelism-specific hardness guarantees. The entire framework can synthesize verification tasks fully automatically for several program representations and property specifications. As demonstrated by its use in the RERS Challenge, it provides developers of verification tools with tailored benchmarks and thereby valuable feedback on their tool's performance. Thus, the result of this work helps to advance the state of the art of program verification.

[1] Support for modal contracts is planned to be made publicly available with the next release (0.11).

7.1 Future Work

The introduced generation framework involves clearly defined intermediate representations and preservation guarantees that are associated with individual transformations. This framework is therefore modular and it would be interesting to see further applications and extensions thereof. First of all, the extraction of abstraction-based CTL properties using the state-distinguishing formulas of $[\text{JSS20}]_{\text{AP}}$ has not yet been applied in RERS due to time constraints. Similarly, a RERS track on weak bisimulation checking is planned but was not yet released. A full implementation of generating hard parallel tasks for LTL properties $[\text{JSSS21}]_{\text{AP}}$ based on the AutomataLib is on schedule to be used during RERS 2021. In addition, it is planned to employ a new encoding of Mealy machines for the sequential programs of RERS[2] based on algebraic decision diagrams [BFG+97]—an approach similar to the compilation method presented in [GJMS19].

The list of hardness criteria that the introduced framework ensures can be extended. An apparent extension is the combination of counterexample handles with deep counterexamples, and further hardness guarantees are conceivable. A generation of parallel verification tasks $V(M, \varphi)$ such that φ is a CTL property which is M-sensitive—meaning that all parallel components are required to correctly verify or refute φ on M—is left as an apparent yet uncompleted addition to this work. Furthermore, another extension of the formal framework would be to support sets Φ of properties such that every $\varphi \in \Phi$ is M-sensitive. A relaxation of the clearly defined transition labeling within alphabet extensions could provide a useful tool to proceed in this direction.

Synthesizing additional types of programs can expand the impact of the introduced framework. Supporting state-based systems in addition to action-based systems was identified as a welcome addition by participants of the Model Checking Contest (MCC) because many verification tasks in the MCC benchmark contain state-based properties. Moreover, an extension from handshake synchronization to buffered communication as found in distributed computing would add further application scenarios to the introduced synthesis. Because the presented parallel decomposition is inherently relying on the used CSP-like composition, such an extension to distributed systems will likely require deeper modifications to the framework or an explicit modeling of buffers as processes.

Key features of the presented overall approach are that (i) new verification benchmarks can be generated with the push of a button and that (ii) verification tasks can be tailored to individual hardness profiles by adjusting parameters like program size, the number of computation steps after which errors occur, and the number of (relevant) parallel components. Future work should therefore involve a publicly available generation tool which makes the presented framework directly available to every developer of a verification tool. To ease its use and allow for design flexibility, such a benchmark generator could include graphical editing [NLKS18] of, e.g., modal contracts. With adjustable verification tasks ready for generation, developers of verification tools could directly inspect strength and weaknesses of their tools during development.

[2] This work is based on a recent bachelor's thesis by David Schmidt [Sch21] that Bernhard Steffen and I supervised.

References

[AA16] Ephrem Ryan Alphonsus and Mohammad Omar Abdullah. A review on the applications of programmable logic controllers (PLCs). *Renewable and Sustainable Energy Reviews*, 60:1185–1205, 2016.

[ADKT11] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. Making software verification tools really work. In *Automated Technology for Verification and Analysis*, pages 28–42. Springer, 2011.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[AS87] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[BBB+19] Ezio Bartocci, Dirk Beyer, Paul E. Black, Grigory Fedyukovich, Hubert Garavel, Arnd Hartmanns, Marieke Huisman, Fabrice Kordon, Julian Nagele, Mihaela Sighireanu, Bernhard Steffen, Martin Suda, Geoff Sutcliffe, Tjark Weber, and Akihisa Yamada. TOOLympics 2019: An overview of competitions in formal methods. In *TACAS*, pages 3–24. Springer, 2019.

[BBYW15] Paul E Black, Irena Bojanova, Yaacov Yesha, and Yan Wu. Towards a periodic table of bugs. In *15th High Confidence Software and Systems Conference (HCSS)*, 2015.

[BC17] Albert Benveniste and Benoît Caillaud. Synchronous interfaces and assume/guarantee contracts. In *Models, Algorithms, Logics and Tools*, volume 10460 of *LNCS*, pages 233–248. Springer, 2017.

[BCN+18] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen.

Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400, 2018.

[BDF+13] Nikola Beneš, Benoît Delahaye, Uli Fahrenberg, Jan Křetínský, and Axel Legay. Hennessy-Milner logic with greatest fixed points as a complete behavioural specification theory. In *CONCUR 2013 – Concurrency Theory*, pages 76–90. Springer, 2013.

[BDH+12] Sebastian S Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In *International Conference on Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 43–58. Springer, 2012.

[BdMS05] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability modulo theories competition. In *Computer Aided Verification*, pages 20–23. Springer, 2005.

[BEL+17] Borja Balle, Rémi Eyraud, Franco M. Luque, Ariadna Quattoni, and Sicco Verwer. Results of the sequence prediction challenge (SPiCe): a competition on learning the next symbol in a sequence. In *Proceedings of The 13th International Conference on Grammatical Inference*, volume 57 of *Proceedings of Machine Learning Research*, pages 132–136. PMLR, 05–07 Oct 2017.

[Bey12] Dirk Beyer. Competition on Software Verification. In *TACAS*, volume 7214 of *LNCS*, pages 504–524. Springer, 2012.

[Bey21] Dirk Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In *TACAS*, pages 401–422. Springer, 2021.

[BFB+17] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, et al. First International Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer*, pages 1–40, April 2017.

[BFG+97] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal methods in system design*, 10(2):171–206, 1997.

[Bie21] Armin Biere. Bounded model checking. In *Handbook of Satisfiability*, pages 739–764. IOS Press, 2021.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, February 2019.

[BMSH10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On weak modal compatibility, refinement, and the MIO Workbench. In *TACAS*, pages 175–189. Springer, 2010.

[Boe15] Carl Boettiger. An introduction to Docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[Bü66] J. Richard Büchi. Symposium on decision problems: On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1 – 11. Elsevier, 1966.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CFL+20] Rafael C. Cardoso, Marie Farrell, Matt Luckcuck, Angelo Ferrando, and Michael Fisher. Heterogeneous verification of an autonomous Curiosity rover. In *NASA Formal Methods*, pages 353–360. Springer, 2020.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169. Springer, 2000.

[CGP02] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 431–441, 2002.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, 1989.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 411–420, May 1999.

[DFH+21] Claire Dross, Carlo A. Furia, Marieke Huisman, Rosemary Monahan, and Peter Müller. VerifyThis 2019: a program verification competition. *International Journal on Software Tools for Technology Transfer*, 2021.

[dGRdB+15] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *Computer Aided Verification*, volume 9206 of *LNCS*, pages 273–289. Springer, 2015.

[DLLF+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *LNCS*, pages 122–129. Springer, October 2016.

[DMM+10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. GPGPU-3, page 63–74, New York, NY, USA, 2010. ACM.

[EMIO07] E. Estevez, M. Marcos, N. Iriondo, and D. Orive. Graphical modeling of PLC-based industrial control applications. In *2007 American Control Conference*, pages 220–225, 2007.

[Eri96] K. T. Erickson. Programmable logic controllers. *IEEE Potentials*, 15(1):14–17, Feb 1996.

[FBU09] Dario Fischbein, Victor Braberman, and Sebastian Uchitel. A sound observational semantics for modal transition systems. In *Theoretical Aspects of Computing - ICTAC 2009*, pages 215–230. Springer, 2009.

[Fis36] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[FMB21] Alessio Ferrari, Franco Mazzanti, and Davide Basile. Systematic evaluation and usability analysis of formal tools for railway system design. *arXiv preprint arXiv:2101.11303*, 2021.

[Gar19] Hubert Garavel. Nested-unit Petri nets. *Journal of Logical and Algebraic Methods in Programming*, 104:60 – 85, 2019.

[GCM09] N. Goga, S. Costache, and F. Moldoveanu. A formal analysis of ISO/IEEE P11073-20601 standard of medical device communication. In *3rd Annual IEEE Systems Conference*, pages 163–166, 2009.

[GFI16] Steven N Goodman, Daniele Fanelli, and John PA Ioannidis. What does research reproducibility mean? *Science translational medicine*, 8(341):341ps12–341ps12, 2016.

[GJMS19] Frederik Gossen, Marc Jasper, Alnis Murtovi, and Bernhard Steffen. Aggressive aggregation: a new paradigm for program optimization. *arXiv preprint arXiv:1912.11281*, 2019.

[GJS+16] Maren Geske, Marc Jasper, Bernhard Steffen, Falk Howar, Markus Schordan, and Jaco van de Pol. RERS 2016: Parallel and sequential benchmarks with focus on LTL verification. In *ISoLA*, volume 9953 of *LNCS*, pages 787–803. Springer, 2016.

[GL94] Orna Grumberg and David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. *ACM SIGSOFT Software Engineering Notes*, 28(5):257–266, 2003.

[GMBG20] Carlos Cardoso Galhardo, Peter Mell, Irena Bojanova, and Assane Gueye. Measurements of the most significant software security weaknesses. In *Annual Computer Security Applications Conference*, pages 154–164, 2020.

[GSF06] V. Gourcuff, O. De Smet, and J. M. Faure. Efficient representation for formal verification of plc programs. In *2006 8th International Workshop on Discrete Event Systems*, pages 182–187, July 2006.

[GvLH+96] Patrice Godefroid, Jan van Leeuwen, Juris Hartmanis, Gerhard Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996.

[HGM20] Marieke Huisman, Dilian Gurov, and Alexander Malkis. Formal methods: From academia to industrial practice. a travel guide. *arXiv preprint arXiv:2002.07279v1*, 2020.

[HIM+14] F. Howar, M. Isberner, M. Merten, B. Steffen, D. Beyer, and C. Păsăreanu. Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. *International Journal on Software Tools for Technology Transfer*, 16(5):457–464, 2014.

[HJM+21] Falk Howar, Marc Jasper, Malte Mues, David Schmidt, and Bernhard Steffen. The RERS challenge: towards controllable and scalable benchmark synthesis. *International Journal on Software Tools for Technology Transfer*, 2021.

[HKG+12] Hossein Hojjat, Filip Konečný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems. In *FM 2012: Formal Methods*, pages 247–251. Springer, 2012.

[HKM15] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis 2012. *International Journal on Software Tools for Technology Transfer*, 17(6):647–657, Nov 2015.

[HKP+19] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The quantitative verification benchmark set. In *TACAS*, volume 11427 of *LNCS*, pages 344–350. Springer, 2019.

[HL89] Hans Hüttel and Kim G Larsen. The use of static constructs in a model process logic. In *International Symposium on Logical Foundations of Computer Science*, pages 163–180. Springer, 1989.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, pages 299–309, 1980.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[Hol11] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[HWS20] Ben Hermann, Stefan Winter, and Janet Siegmund. Community expectations for research artifacts and evaluation processes. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 469–480. ACM, 2020.

[IHS14] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification*, pages 307–322. Springer, 2014.

[Jas18] Marc Jasper. Benchmarks for the verification of parallel programs: Guaranteed properties, hardness, and scalability. Master's thesis, TU Dortmund University, 2018.

[JFS+17] Marc Jasper, Maximilian Fecke, Bernhard Steffen, Markus Schordan, Jeroen Meijer, Jaco van de Pol, Falk Howar, and Stephen F. Siegel. The RERS 2017 challenge and workshop (invited paper). In *24th SIGSOFT Intl. SPIN Symp. on Model Checking of Software*, pages 11–20. ACM, 2017.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, Mar. 2012.

[JMM+19] Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon Schiffelers, Harco Kuppens, and Frits W. Vaandrager. RERS 2019: Combining synthesis with real-world models. In *TACAS 2019*, volume 11429 of *LNCS*, pages 101–115. Springer, 2019.

[JMS+18] Marc Jasper, Malte Mues, Maximilian Schlüter, Bernhard Steffen, and Falk Howar. RERS 2018: CTL, LTL, and reachability. In *ISoLA. LNCS, vol 11245*, pages 433–447. Springer, 2018.

[Joh75] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[JS16] Marc Jasper and Markus Schordan. Multi-core model checking of large-scale reactive systems using different state representations. In *ISoLA*, volume 9952 of *LNCS*, pages 212–226. Springer, 2016.

[JS18] Marc Jasper and Bernhard Steffen. Synthesizing subtle bugs with known witnesses. In *ISoLA*, volume 11245 of *LNCS*, pages 235–257. Springer, 2018.

[JSS20] Marc Jasper, Maximilian Schlüter, and Bernhard Steffen. Characteristic invariants in Hennessy–Milner logic. *Acta Informatica*, 57(3):671–687, 2020.

[JSSS21] Marc Jasper, Maximilian Schlüter, David Schmidt, and Bernhard Steffen. Every component matters: Generating parallel verification benchmarks with hardness guarantees. In *ISoLA*, volume 12479 of *LNCS*, pages 241–262. Springer, 2021.

[KGH+18] Fabrice Kordon, Hubert Garavel, Lom Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, Francis Hulin-Hubard, Elvio Amparore, Marco Beccuti, Bernard Berthomieu, Hugues Evrard, Peter G. Jensen, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Jiří Srba, Yann Thierry-Mieg, Jaco van de Pol, and Karsten Wolf. MCC'2017 – the seventh model checking contest. In *Transactions on Petri Nets and Other Models of Concurrency XIII*, volume 11090 of *LNCS*, pages 181–209. Springer, 2018.

[KHHH+21] Fabrice Kordon, Lom Messan Hillah, Francis Hulin-Hubard, Loïg Jezequel, and Emmanuel Paviot-Adet. Study of the efficiency of model checking techniques using results of the MCC from 2015 to 2019. *International Journal on Software Tools for Technology Transfer*, 2021.

[KLB+12] Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Kai Lampka, Niels Lohmann, Emmanuel Paviot-Adet, Yann Thierry-Mieg, and Harro Wimmel. Report on the Model Checking Contest at Petri Nets 2011. In *Transactions on Petri Nets and Other Models of Concurrency VI*, volume 7400 of *LNCS*, pages 169–196. Springer, 2012.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.

[Lar89] Kim Guldstrand Larsen. Modal specifications. In *International Conference on Computer Aided Verification*, pages 232–246. Springer, 1989.

[Lar90] Kim Guldstrand Larsen. Ideal specification formalism= expressivity+ compositionality+ decidability+ testability+... In *International Conference on Concurrency Theory*, pages 33–56. Springer, 1990.

[LLA+17] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. DataRaceBench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17. ACM, 2017.

[LMM19] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. Compositional verification of concurrent systems by combining bisimulations. In *Formal Methods – The Next 30 Years*, pages 196–213. Springer, 2019.

[LMM20] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. Sharp congruences adequate with temporal logics combining weak and strong modalities. In *TACAS*, volume 12079 of *LNCS*, pages 57–76. Springer, 2020.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 108–117. IEEE, 1990.

[MDB14] Grgur Petric Maretić, Mohammad Torabi Dashti, and David Basin. LTL is closed under topological closure. *Information Processing Letters*, 114(8):408–413, 2014.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[Moo94] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems*, 14(2):53–59, April 1994.

[MWC10] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, February 2010.

[NLKS18] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *International Journal on Software Tools for Technology Transfer*, 20(3):327–354, 2018.

[NLW+09] Raghunath Othayoth Nambiar, Matthew Lanken, Nicholas Wakou, Forrest Carman, and Michael Majdalany. Transaction processing performance council (TPC): Twenty years later – a look back, a look ahead. In *Performance Evaluation and Benchmarking*, volume 5895 of *LNCS*, pages 1–10. Springer, 2009.

[NSVK19] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. Benchmarks for automata learning and conformance testing. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *LNCS*, pages 390–416. Springer, 2019.

[OAPÜ16] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An overview of model checking practices on verification of PLC software. *Software & Systems Modeling*, 15(4):937–960, 2016.

[OT08] William L. Oberkampf and Timothy G. Trucano. Verification and validation benchmarks. *Nuclear Engineering and Design*, 238(3):716–743, 2008.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical computer science*, pages 167–183. Springer, 1981.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.

[PGG18] Christian R. Prause, Rainer Gerlich, and Ralf Gerlich. Evaluating automated software verification tools. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 343–353, 2018.

[Ple18] Hans E. Plesser. Reproducibility vs. replicability: A brief history of a confused terminology. *Frontiers in Neuroinformatics*, 11:76, 2018.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190. ACM Press, 1989.

[Qui00] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel processing letters*, 10(02n03):215–226, 2000.

[RBB+11] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. A modal interface theory for component-based design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.

[RK98] M. Rausch and B. H. Krogh. Formal verification of PLC programs. In *Proceedings of the 1998 American Control Conference. ACC*, volume 1, pages 234–238, 1998.

[Sac05] Krzysztof Sacha. Automatic code generation for PLC controllers. In *Computer Safety, Reliability, and Security*, pages 303–316. Springer, 2005.

[SBS18] Markus Schordan, Dirk Beyer, and Stephen F. Siegel. Evaluating tools for software verification (track introduction). In *ISoLA*, volume 11245 of *LNCS*, pages 139–143. Springer, 2018.

[Sch21] David Schmidt. Mealy-machine based program synthesis using algebraic decision diagrams. Bachelor's thesis. TU Dortmund University, 2021.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011. Advanced Lectures*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.

[SJ17] Bernhard Steffen and Marc Jasper. Property-preserving parallel decomposition. In *Models, Algorithms, Logics and Tools*, volume 10460 of *LNCS*, pages 125–145. Springer, 2017.

[SJ19] Bernhard Steffen and Marc Jasper. Generating hard benchmark problems for weak bisimulation. In *From Reactive Systems to Cyber-Physical Systems. LNCS, vol 11500*, pages 126–145. Springer, 2019.

[SJMvdP17] Bernhard Steffen, Marc Jasper, Jeroen Meijer, and Jaco van de Pol. Property-preserving generation of tailored benchmark Petri nets. In *17th Intl. Conf. on Appl. of Concurrency to Sys. Design*, pages 1–8. IEEE, 2017.

[Ste17] Bernhard Steffen. The physics of software tools: SWOT analysis and vision. *International Journal on Software Tools for Technology Transfer*, 19(1):1–7, 2017.

[SY20] Stephen F. Siegel and Yihao Yan. Action-based model checking: Logic, automata, and reduction. In *Computer Aided Verification*, volume 12225 of *LNCS*, pages 77–100. Springer, 2020.

[vdPM19] Jaco van de Pol and Jeroen Meijer. Synchronous or alternating? In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *LNCS*, pages 417–430. Springer, 2019.

[VM05] Willem Visser and Peter Mehlitz. Model checking programs with Java PathFinder. In *International SPIN Symposium on Model Checking of Software*, volume 3639 of *LNCS*, pages 27–27. Springer, 2005.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1):72 – 99, 1983.

[WVS20] J. Stanley Warford, David Vega, and Scott M. Staley. A calculational deductive system for linear temporal logic. *ACM Computing Surveys*, 53(3), 2020.

[ZC09] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security Privacy*, 7(2):87–90, March 2009.

Online References

[1] Various authors. Competition contributions written by participants of SV-COMP 2019. https://rd.springer.com/book/10.1007/978-3-030-17502-3?page=1#toc, 2019. Accessed: May 31st, 2021.

[2] Frédéric Cristini and Silvano Dal Zilio. Model "SatelliteMemory" from the Model Checking Contest benchmark. https://mcc.lip6.fr/pdf/SatelliteMemory-form.pdf, 2020. Accessed: May 29th, 2021.

[3] Dirk Beyer et al. 10th Competition on Software Verification (SV-COMP 2021): Benchmark verification tasks. https://sv-comp.sosy-lab.org/2021/benchmarks.php. Accessed: May 23rd, 2021.

[4] Dirk Beyer et al. Benchmark verification tasks of SV-COMP 2014. https://sv-comp.sosy-lab.org/2014/benchmarks.php, 2014. Accessed: May 29th, 2021.

[5] Fabrice Kordon et al. Benchmark of the 2018 edition of the Model Checking Contest. https://mcc.lip6.fr/2018/models.php, 2018. Accessed: May 31st, 2021.

[6] Markus Schordan et al. Codethorn, a tool for program analysis and verification. https://github.com/rose-compiler/rose/tree/master/projects/CodeThorn, 2012. Accessed: May 31st, 2021.

[7] Philip Zweihoff et al. EasyDelta: Industrial programming by example. https://cinco.scce.info/applications/easy-delta-industrial-programming-by-example/, 2015. Accessed: May 31st, 2021.

[8] Stephen North et al. Graphviz - graph visualization software. https://graphviz.org/doc/info/lang.html. Accessed: May 21st, 2021.

[9] Bernhard Steffen et al. Falk Howar. Results of previous iterations of the RERS challenge. http://www.rers-challenge.org/<insert-year-here>/index.php?page=results. Accessed: May 31st, 2021.

[10] Association for Computing Machinery. Artifact review and badging version 1.1. https://www.acm.org/publications/policies/artifact-review-and-badging-current. Accessed: May 23rd, 2021.

[11] Members of the Chair of Programming Systems at TU Dortmund University. Automatalib. https://learnlib.de/projects/automatalib/. Accessed: May 21st, 2021.

[12] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/, 2012. Accessed: May 29th, 2021.

[13] Kevin Poulsen. Software bug contributed to blackout. https://www.securityfocus.com/news/8016. Accessed: May 21st, 2021.