UNIVERSITÄT
DES
SAARLANDES

Parameterized Verification and Repair of Concurrent Systems

by
Mouhammad Sakr

A dissertation submitted towards the degree Doctor of Engineering (Dr.-Ing.) of the Faculty of Mathematics and Computer Science of Saarland University

Saarbrücken, 2021

Dean of the faculty	Prof. Dr. Thomas Schuster
Advisor	PD Dr.-Ing. Swen Jacobs
Date of the colloquium	May 31, 2021
Chair of the committee	Prof. Dr. Sebastian Hack
Reviewers	Prof. Paul Attie, Ph.D. Prof. Bernd Finkbeiner, Ph.D. PD Dr.-Ing. Swen Jacobs
Academic Assistant	Dr. Charlie Jacomme

Abstract

In this thesis, we present novel approaches for model checking, repair and synthesis of systems that may be parameterized in their number of components. The *parameterized model checking problem* (PMCP) is in general undecidable, and therefore the focus is on restricted classes of parameterized concurrent systems where the problem is decidable. Under certain conditions, the problem is decidable for guarded protocols, and for systems that communicate via a token, a pairwise, or a broadcast synchronization. In this thesis we improve existing results for guarded protocols and we show that the PMCP of guarded protocols and token passing systems is decidable for specifications that add a quantitative aspect to LTL, called Prompt-LTL.

Furthermore, we present, to our knowledge, the first parameterized repair algorithm. The *parameterized repair problem* is to find a refinement of a process implementation p such that the concurrent system with an arbitrary number of instances of p is correct. We show how this algorithm can be used on classes of systems that can be represented as well structured transition systems (WSTS).

Additionally we present two *safety synthesis* algorithms that utilize a lazy approach. Given a faulty system, the algorithms first symbolically model check the system, then the obtained error traces are analyzed to synthesize a candidate that has no such traces. Experimental results show that our algorithm solves a number of benchmarks that are intractable for existing tools. Furthermore, we introduce our tool AIGEN for generating random Boolean functions and transition systems in a symbolic representation.

Zusammenfassung

In dieser Arbeit stellen wir neuartige Ansätze für das Model-Checking, die Reparatur und die Synthese von Systemen vor, die in ihrer Anzahl von Komponenten parametrisiert sein können. Das Problem des parametrisierten Model-Checking (PMCP) ist im Allgemeinen unentscheidbar, und daher liegt der Fokus auf eingeschränkten Klassen parametrisierter synchroner Systeme, bei denen das Problem entscheidbar ist. Unter bestimmten Bedingungen ist das Problem für Guarded Protocols und für Systeme, die über ein Token, eine Pairwise oder eine Broadcast-Synchronisation kommunizieren, entscheidbar. In dieser Arbeit verbessern wir bestehende Ergebnisse für Guarded Protocols und zeigen die Entscheidbarkeit des PMCP für Guarded Protocols und Token-Passing Systeme mit Spezifikationen in der temporalen Logik Prompt-LTL, die LTL einen quantitativen Aspekt hinzufügt.

Darüber hinaus präsentieren wir unseres Wissens den ersten parametrisierten Reparaturalgorithmus. Das parametrisierte Reparaturproblem besteht darin, eine Verfeinerung einer Prozessimplementierung p zu finden, so dass das synchrone Systeme mit einer beliebigen Anzahl von Instanzen von p korrekt ist. Wir zeigen, wie dieser Algorithmus auf Klassen von Systemen angewendet werden kann, die als Well Structured Transition Systems (WSTS) dargestellt werden können.

Außerdem präsentieren wir zwei Safety-Synthesis Algorithmen, die einen "lazy" Ansatz verwenden. Bei einem fehlerhaften System überprüfen die Algorithmen das System symbolisch, dann werden die erhaltenen "Gegenbeispiel" analysiert, um einen Kandidaten zu synthetisieren der keine solchen Fehlerpfade hat. Versuchsergebnisse zeigen, dass unser Algorithmus eine Reihe von Benchmarks löst, die für bestehende Tools nicht lösbar sind. Darüber hinaus stellen wir unser Tool AIGEN zur Erzeugung zufälliger Boolescher Funktionen und Transitionssysteme in einer symbolischen Darstellung vor.

Acknowledgments

I am extremely grateful to my supervisor Dr. Swen Jacobs for accepting me as his student and for all of the opportunities he gave me to further my research. Thank you Swen for your invaluable advice, endless patience, and continuous support. I am very grateful that you were always there to listen to my ideas and problems and for your valuable recommendations and guidance.

I would also like to express my gratitude to Prof. Bernd Finkbeiner for giving me the opportunity to work at his group, for the fruitful discussions, for reviewing my thesis, and for all the wonderful social gatherings.

Also, I would like to thank Prof. Paul Attie for introducing me to formal methods, for encouraging me and supporting me to pursue my doctorate, and for reviewing my thesis.

I want to thank my colleagues Joachim Bard, Norine Coenen, Peter Faymonville, Michael Gerke, Christopher Hahn, Jesko Hecking-Harbusch, Jana Hofmann, Felix Klein, Shyam Lal Karra, Noemi Passing, Christa Schäfer, Malte Schledjewski, Maximilian Schwenger, Leander Tentrup, Hazem Torfah, Alexander Weinert, and Martin Zimmermann for their support, interesting discussions, coffee breaks, and delicious cakes. Special thanks to Shyam Lal Karra, Alexander Weinert, Jesko Hecking-Harbusch, and my dear friend Mohamad Fawaz for their feedback on a draft of this thesis's introduction.

No words are enough to express my gratitude to my father, Issam, and my mother, Hawla, for their endless love, devotion, sacrifice, and support. I want also to thank my wife Sandra, my sisters, Hiba, Siwar, Dana, and my brother, Ali, for being always beside me during the good and bad times.

My sons, Issam and Noah, you are my motivation, incentive, and reason to keep going. I dedicate this thesis to you.

Contents

Chapter 1

Introduction

1.1 Formal Verification

The outbreak of Coronavirus that started in December 2019 is a big concern to everyone in the world. Due to the absence of a vaccine or medication, many countries are now relying on mobile applications to track new cases and potential infections, as this is currently one of the most effective weapons to restrict and bound the quick spread of the virus. This situation underscores the importance of technology in our lives and the extent to which the current world is dependent on technical systems. Computers and embedded systems are nowadays fundamental in our world, as they play a crucial role in our society and are embedded in every aspect of our life. A large part of these systems are critical in the sense that they exist in crucial areas of our lives ranging from medicine, traffic control, aircraft, chemical and nuclear plants where humans' lives are directly concerned to banking, currency exchange, and stock markets where the economy is directly affected. That said, it is obvious that errors in such systems do not just jeopardize humans lives but may also have catastrophic financial consequences.

For instance the crash of the unmanned rocket Ariane-5 on June 4, 1996 only 36 seconds of after launch was due to an error in one of its systems (The error was in the conversion from a 64-bit floating point into a 16-bit integer). The project costed European Space Agency around 7 billion USD. Another dramatic example is the bug in Intel's Pentium II floating-point division which caused a loss of about 475 million USD. Thus, the correctness of such safety-critical systems is very crucial. This has led to the introduction of *formal verification* which is the mechanism to formally (mathematically) prove that a hardware or software system is correct with respect to certain formal specifications. The two main approaches for formal verification are the deductive approach and the model checking approach.

In *Deductive Reasoning*, a system is verified using formal deduction based on a set of inference rules. Given the formal system description and a set of axioms and inference rules, deductive reasoning consists of logically deriving the specification. This approach is not considered in this thesis.

Model checking is a fully automatic technique to prove correctness of a system with respect to a set of properties. It consists of a systematic exploration

Figure 1.1: Ariane-5, on June 4, 1996, exploded after 36 seconds of its launch. source:http://www-users.math.umn.edu/ arnold/disasters/ariane.html

of all states of a model of the system, which may be an abstraction that omits unimportant behavior. A model checking algorithm usually takes as input a mathematical model and a formal specification and outputs yes if the model is correct and a counter example otherwise. The counter example is an execution of the model that violates the given specifications. Investigations have revealed that if formal verification techniques had been used for the Ariane-5 rocket and Intel's Pentium II processor, the above-mentioned floating points errors would have been detected. Furthermore scientists at IBM found that 40% of the design errors detected by model checking could never have been found by testing.

Unfortunately, Alan Turing proved that program termination is undecidable in general (there is no algorithm that takes an arbitrary program as input, and can decide whether it terminates or not for every input). Furthermore Henry Rice has shown that every non-trivial property of programs is undecidable and Marvin Minsky has demonstrated that every non-trivial property of while-programs with two counter variables is undecidable.

Despite these negative results scientists in the formal verification field didn't give up as these are general results. A thorough analysis of these findings shows that undecidability requires some source of infinity, e.g., variables with an infinite domain. However there exists many systems with a finite number of states especially hardware systems (e.g. sequential circuits) and concurrent systems. Moreover, in recent decades, a plethora of different abstraction techniques were introduced in which an infinite-state system (or a very large system) can be simulated by a finite-state model.

Clarke and Emerson introduced branching temporal logic model checking [33] in the early 1980's. Temporal logic is a logic for specifying properties over time. In linear temporal logic (or short LTL) the model of time is linear and in branching temporal logic the time is modeled in a tree-like structure. Two important properties that can be specified using these logic are *safety* and *liveness*. Informally, a safety property denotes that nothing bad will happen, and a liveness property states that something good will happen. In [33], Clarke and Emerson presented an algorithm that takes as parameters a finite graph (i.e. a system model) and a branching temporal logic formula and decides in finite time whether the graph models the formula. The complexity of the algorithm is linear in the size of the formula and the graph.

Despite the fact that Emerson's and Clarke's seminal paper was a breakthrough in the formal verification area, another important complication emerged especially in model checking: The *state explosion problem.* This complication can be encountered when the size of the system space grows exponentially and a model checking algorithm might need years to check a temporal logic property. Various approaches can be used to address this problem, including:

- Abstraction. The aim of any abstraction approach is to reduce the state space of the system by discarding details that are unnecessary to the given specifications.

- Symbolic model checking (BDD based). A set-based model checking technique where set of states are represented symbolically using Binary Decision Diagrams (or BDD).

- Bounded model checking (BMC). In BMC, given a number k, a symbolic representation of a system M, and a specification formula φ, a SAT formula is constructed (propositional statement in conjunctive normal form) such that it is satisfiable if and only if there exists a counterexample for φ of length k.

Using such techniques current model checkers can handle programs with up to 10^{476} states [11].

Another important technique that we see as a complementary procedure to model checking is *automatic model repair.* Typically, when a model checker returns a counter example, a system designer has to manually analyze this undesired execution with the aim of modifying the system and eliminating this erroneous behavior. This procedure has to be repeated until the system is error free, or in other words until the model checker outputs the yes answer. An intuitive solution would be to automate this tedious verify-diagnose-repair cycle used in software and hardware design. This led to the problem of model repair introduced by Buccafurri et al. [24], as a natural extension of model checking.

> Given a mathematical model M and a specification formula φ such that M does not satisfy φ, is there a way to modify M such that the resulting new model M' satisfies φ?

1.2 Concurrent Systems

A huge amount of hardware and software systems are not sequential but parallel. A concurrent system comprises of a set of finite sequential systems (called processes) running concurrently. That is, at any moment in time either one process executes an action or two or more processes execute simultaneously a synchronous action. The study of concurrency started with Edsger W. Dijkstra in 1965 when he introduced the mutual exclusion problem [39]. Although processes in a concurrent system are often of a finite size they are hard to get correct. The main source of complexity is the remarkably large number of possible interactions between processes. Additionally, the number of states grows exponentially with the number of processes which results in the infamous state explosion problem. Furthermore concurrency brings in new tedious problems like *deadlocks*. A deadlock is a undesired state of a system where one or more processes are prevented from taking any action (informally they freeze). These difficulties make concurrent systems a promising application area for formal methods like model checking, repair, and synthesis.

Network Topology. The arrangement of processes in a concurrent system is called *topology*. A system's topology can be seen as a directed graph where nodes are processes and an edge from node p_1 to node p_2 indicates that process p_1 can send a message to process p_2. For instance in a *clique* topology, a process can communicate with any other process. If for a certain system no topology is mentioned, then one can assume it is a clique. In a *ring* topology a process can only communicate with its left or right neighbor. As we will see in later sections, concurrent system's topologies are very important and plays a decisive role in the decidability/undecidability of the model checking problem for the so-called *parameterized concurrent systems* (or shortly parameterized systems).

1.3 Parameterized Systems

This thesis focuses on *Parameterized Systems* which is a subclass of concurrent systems that consists of an arbitrary number of instances (replications) of the same component. We call each of these replications a process, and each component a process template. Many systems are normally modeled as parameterized systems. Concrete examples of such systems are: mutual exclusion protocols, leader election, cache coherence protocols, sensor networks, cryptographic protocols with unbounded principals and sessions, multi-threaded programs, robot swarms, and vehicular networks. For a fixed number of processes the system is finite, however the state space of the whole family is considered infinite. Namely, a parameterized system is an infinite family of finite state systems. Although existing general-purpose formal verification methods can give correctness guarantees for fixed size concurrent systems, the state explosion problem forbids applying these techniques on systems with a very large number of processes, and additional arguments are needed to extend a proof of correctness to systems of arbitrary size. Both problems can be tackled by techniques for parameterized model checking and synthesis, which grants correctness guarantees for systems with any desired number of components without examining every possible system instance explicitly.

> The parameterized model checking problem (PMCP) [43] is to decide whether a temporal logic property is true for every size instance of a given system.

Note that the size of a concurrent system is the number of its processes.

Example 1 (mutual exclusion)**.** Suppose that we want to protect the integrity of a piece of memory MeM. A process template B that can write to MeM can be modeled as in Figure 1.2, where *idle* means that the process does not want to access the memory, *trying* denotes that the process wants to write, and *writing* represents the state in which the process is writing to the memory. Let B^k denotes the system that comprises k instances of B running in parallel. Then to preserve memory integrity the desired property would be : *mutual exclusion* i.e.

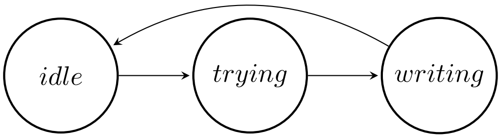


Figure 1.2: Mutex

only one process in the writing section at a time. In this case the parameterized model checking is to check if for all $n \in \mathbb{N} : B^n$ satisfies mutual exclusion.

Unfortunately the PMC problem is in general undecidable, even for simple *reachability* problems. Note that the reachability problem is to decide whether a certain state of a system is reachable from an initial state. Likewise, the *coverability* problem is a special form of the reachability where instead of checking if a state q_i is reachable from an initial state q_0, we check if q_0 can reach a state q_j that covers (subsumes) q_i. For instance, consider the two concurrent systems B^3 and B^2 where B is the process template in Figure 1.2. Then, A global state s_1 of B^3 in which two process instances are in the *writing* state and one instance in the *trying* state *covers* a global state s_2 of B^2 in which two process instances are in the *writing* state. In other words the counters of s_1 are pointwise greater or equal than those of s_2. Suzuki has shown in [92] that PMCP is undecidable even if the system consists of a unidirectional ring of identical finite-state processes. In light of this unfortunate result, to obtain correctness guarantees for parameterized systems, researchers have been using seven main approaches:

- Partial order reduction
- Cutoff
- Well structured transition systems
- Abstraction (and semi-decision procedures)
- Invisible Invariants
- Restricted Systems
- Regular model checking

Many of these approaches have been collected in surveys of the literature recently [19,49] [34, Chapter 21].

1.3.1 Partial Order Reduction

Partial order reduction is used in an attempt to reduce the time and space needed to automatically verify concurrent systems relying on equivalence of different interleavings of executed actions. As mentioned earlier, the main source of complexity in concurrent systems is the huge number of possible interactions between processes. Partial order reduction makes use of the fact that the satisfiability of a temporal logic formula is often insensitive to the order in which processes' independent actions are interleaved. Therefore instead of exploring all possible executions of a concurrent system, designers first abstract the system into a model that contains only representatives of classes of executions that are equivalent for a given correctness property [72].

1.3.2 Cutoffs

The cutoff technique [9,31,43,65,67] reduces the parameterized model checking problem to the ordinary model checking problem. Formally the cutoff method reduces model checking of systems of an arbitrary size to the model checking of systems of fixed size. This number is denoted by *cutoff* for the given parameterized system. The cutoff is also used in parameterized synthesis [61,73]. The existence of a cutoff c for a specific subclass of parameterized systems can be proved by showing that there is an execution in the system of size c that violates a given property φ if and only if there exists a violation of φ in every system of size larger than c. More details about this technique can be found in Chapters 3 and 4.

1.3.3 Well Structured Transition Systems

A *transition system* (or short TS) is a (finite or infinite) directed graph where nodes are called states and edges are called transitions. We also denote the set of transitions by the transition relation. A well structured transition system (or short WSTS) is a TS equipped with a *well quasi order* (wqo) $\lesssim$ on its states where $\lesssim$ is compatible with the set of transitions, i.e. if (q, q') and (q_1, q'_1) are two transitions of the system with $q \lesssim q_1$ then $q' \lesssim q'_1$ (we also say that $\lesssim$ is monotonic with respect to the transition relation). Finkel et al have shown in [56] that the reachability problem for well structured transition systems[1] is decidable. Thus, to prove that the reachability of a given transition system TS is decidable, it is enough to prove the existence of a well quasi order over the states of TS that is compatible with the transition relation. The main advantage of the existence of a well quasi order is that it allows us to work with infinite sets that are upward closed by making use of the fact that these sets can be uniquely represented by a finite minimal set. This interesting property makes WSTS appealing to use in reachability analysis. More details about WSTS and well quasi order can be found in Chapter 5.

1.3.4 Abstraction

For parameterized model checking, using abstraction becomes fundamental and indispensable due to the fact that parameterized systems can be viewed as infinite systems. In the following we summarize the six main approaches along which researchers have been conducting their work. Note that some of these methods are *semi procedures*, i.e., they are not guaranteed to terminate in general.

[1] Well structured transition systems with effective predecessor basis. Check Chapter 5 for more details.

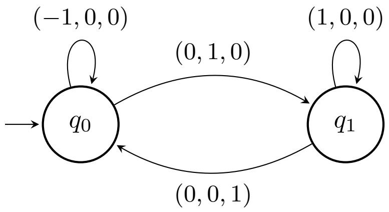


Figure 1.3: A VASS of dimension three with two states

Vector Addition Systems with States (Counter Abstraction)

In Counter abstraction, a state of a parameterized system of unbounded size is abstracted into a set of counters which count, for each local state q of a process, the number of processes which currently are in q. The counters may never hold negative values. The resulting abstraction is called the *counter representation system (or CRS)*. A *vector addition system with states (or VASS)* [59] is a special case of a CRS in which transitions are labeled with integer vectors, and each abstract state is equipped with a control state. Therefore a VASS is a finite directed graph with edges labeled by integer vectors.

A VASS can model systems that consist of a single *controller process* and an arbitrary number of *user processes* with identical process template. Figure 1.3 depicts a simple VASS with two states. The system can be seen as a three counters, where a transition from q_0 to q_1 denotes the start of incrementing counter one, and a transition from q_1 to q_0 denotes a reset to counter one. Mayr has shown in [80] that the reachability problem is decidable for VASSs. Hence, for a given system $A \| B^n$ (one controller process A, and n instances of user process B running in parallel) the following two statements are equivalent:

- The reachability problem for $A \| B^n$ is decidable.
- $A \| B^n$ behavior can be simulated by a VASS.

A *vector addition system* (VAS) is a VASS with a single state.
A *Petri net* is another widely used classical model to simulate concurrent systems. A Petri net consists of a finite set of places, transitions, and arcs. An arc connects places to transitions or vice versa. Places may hold token, hence, a transition t can be fired if there are enough tokens in all its *input places* (places that have an arc to t). When a transition is fired, the tokens in input places are moved to the *output places* (places that have an arc from t). A configuration of a Petri net is a vector that specifies the number of tokens in each place. Obviously, the configuration space of a Petri net is a VAS. According to German and Sistla [59], Petri nets, VAS, and VASS are equally powerful.

Monotonic Abstraction

As aforementioned, the existence of a well quasi order on the states of a given system is advantageous and valuable. Unfortunately many systems do not have such an order, meaning that it is not trivial to find a wqo that is monotonic with respect to the transition relation of the system. Given a transition system TS, the main idea of *monotonic abstraction* [4] is to compute an over-approximation $\mathit{Abs}(TS)$ of TS, called the abstract system, such that the behavior of $\mathit{Abs}(TS)$ over-approximates the behavior of TS. The abstract system should be simpler than the original system such that there exists a wqo over the states of $\mathit{Abs}(TS)$ that is compatible with the transition relation. In abstraction in general, we denote the original system by the *concrete system*. This abstraction technique is a sound method, that is, if the abstract system satisfies a certain property, then we can conclude that the concrete system also satisfies the same property. On the other side, over-approximations bring in the problem of false-positives which is a violation of a given property in the abstract system that does not exist in the concrete system.

Environment Abstraction

Given a parameterized system, the *environment abstraction* approach [32] constructs a finite state transition system such that if the abstract model satisfies a safety or a liveness property φ then the concrete parameterized system satisfies φ. Therefore this approach is sound but not complete. Briefly, this technique combines counter abstraction with *predicate* abstraction. In predicate abstraction, a state is a tuple of Boolean values each of which denote whether a certain property (predicate) holds true or not. Unlike counter abstraction, and instead of the number of processes in a given states, counters carry the number of processes that satisfy a given predicate. Furthermore, the counters used are cutoff at the value 1. In contrast to all techniques mentioned so far, environment abstraction is usually used for parameterized systems that comprise processes with local variables over unbounded domains (e.g. integers). The crucial point of the technique is the separation between the finite control variables and the unbounded data variables. A state of the abstract system is a tuple $(pc, e_1, \ldots, e_T)$ where pc is the control location of the reference process (the process that appears in the specification), and $e_1, \ldots, e_T$ denote the number of processes that satisfy *environment predicates* $\varepsilon_1, \ldots, \varepsilon_T$. An environment predicate ε_i has the form $R_i \wedge pc = j$ which is satisfied by any process that is in control location j and has a *relationship* R_i with the reference process. A *relationship* R_i is a Boolean formula over the set of predicates. The set of predicates is the finite set of all cases which denote how the data variables in the reference process can relate to the data variables in another process. Therefore, to model check a parameterized system, we first construct a finite abstract model and we feed it to an ordinary model checker. Using this type of abstraction, Lamport's bakery algorithm [77] was verified [32] although it uses process id's. It is known that the use of processes' identities in a parameterized system is one of the main reason behind the undecidability of PMCP [49] due to the fact that these types of systems lack symmetry.

View Abstraction

To solve the reachability problem for parameterized systems, the *view abstraction* technique [2] dynamically detects a halting point beyond which a forward search algorithm need not to continue. Given a parameterized system B^n, the forward analysis is an infinite loop that starts from the set of initial states for a system with fixed small size k (usually from $k = 1$), and then it computes an overapproximation $\mathcal{ARE}_k$ for the reachability set $\mathcal{RE}$ of B^n. If the intersection of $\mathcal{ARE}_k$ with the set of unsafe states is not empty, k is increased by one. The analysis terminates once $\mathcal{ARE}_k$ does not include any unsafe states, and then k would be the dynamic halting point. A configuration of B^n is a word over states of B. The basic idea of the overapproximation is that by combining configurations of size up to k we can construct configurations that although not reachable by the system B^k, they are subwords or configurations for systems bigger than B^k. The method is in general sound and becomes complete for a large class of systems with well quasi ordering. The name *View Abstraction* comes from the fact that the complete parameterized system is considered from the *view* of only k processes.

1.3.5 Invisible Invariants

An *invariant* is a property that holds in all reachable states of a system. An *inductive invariant* is a property that holds in every initial state and is preserved with respect to the transition relation. To prove that an assertion is an invariant, we usually synthesize an auxiliary assertion that is inductive and that strengthens (implies) the original assertion. The goal of the *Invisible Invariant* technique [87] is to generate an inductive auxiliary assertion for parameterized systems by considering only a fixed size system. Therefore, the authors of [87] first showed that for a specific set of assertions, called R-assertions, there is a number n_0, such that an R-assertion is inductive in every system P^{n_1} with $n_1 \leq n_0$ if and only if it is inductive for any system P^n with $n \geq 1$. Then, they generate a candidate invariant by computing symbolically the set of all reachable states for the fixed size system P^{n_0}. Assuming that the system is symmetric, the set of reachable states set is projected on one of the processes, say B_1. Then by replacing each instance of 1 with j, the candidate invariant $\forall j \varphi(j)$ is computed. Finally the invariant is checked for inductiveness in P^{n_0}. The approach is denoted by *invisible invariant* due to the fact that the generated candidate invariants are checked for inductiveness without been inspected by a user. This approach is in general not complete.

1.3.6 Regular Model Checking

Regular model checking [22], or short RMC, is a paradigm for verification of parameterized systems with linear or ring-formed topology. RMC can be used for systems in which states can be represented by *words* (strings) of arbitrary finite length, where each letter in the word describes the state of one process in the system. The term regular comes from the fact that sets of states can be represented by regular expressions or equivalently finite state automatons, and hence the technique is symbolic. Transitions between words are represented by finite state *transducers* where a finite state transducer can be seen as a finite state automaton that additionally produces output. The aim of RMC is to apply existing automaton theoretic algorithms for manipulating regular sets.

The technique starts by constructing a transducer that recognizes the transitive closure of the regular relation[2] between global states of the system. The construction might not terminate (the transducer might not be finite) unless we

[2] A set of pairs of words is regular if it can be recognized by a finite state transducer.

pose some restrictions on the system transitions [71]. The transducer is finite if actions of the system have a bounded *local depth*. An action has a local depth d if each position in the word (global state) is changed at most d times in any sequence of executions of that action. Then we can compute the reachability set by applying the constructed transducer relation iteratively to the set of initial states represented by a regular expression. Furthermore we can verify liveness properties [5].

1.3.7 Restricted Systems

The undecidability of the parameterized model checking problem is a general result and therefore researchers work on identifying subclasses of concurrent systems for which the problem is decidable. These subclasses are usually obtained by imposing restrictions on the organization and synchronization of processes. In the following we briefly explain five interesting subclasses for which there exists decidability results. This is often achieved by using one of the techniques mentioned above.

Broadcast Systems

In a broadcast system if a process sends a message, all other processes must receive the message, *everybody must listen*. Thus, in a communication transition of the system all processes make a step. Figure 1.4 sketches a simple process template with broadcast communication. The transition's label $m_1!!$ indicates sending the message m_1 and the label $m_1??$ indicates receiving the message m_1. For instance, suppose we have process p_0 in state q_0, p_1 in state q_1, and p_2 in state q_2. Then if p_2 sends the message m_1, it moves to q_1, p_0 and p_1 receive m_1 and move to q_2 and q_0, respectively. Esparza et al. [48] proved that reachability in broadcast systems is decidable by showing that these protocols are WSTS. According to Schmitz and Schnoebelen [88] the time complexity of the coverability problem for broadcast protocols grows faster than any primitive recursive function. On the other hand, Esparza et al. [48] showed that the model checking problem for liveness properties is undecidable. Their proof was based on a reduction from the halting problem on two counter machines which is known to be undecidable.

Pairwise Rendezvous

In a pairwise rendezvous system if a process sends a message, exactly one other process receives it, *somebody must listen*. In other words, in a communication transition of the system exactly two processes take a step and all other processes remain idle. Consider the process template in Figure 1.5. Suppose we have process p_0 in state q_0, p_1 in state q_1, and p_2 in state q_2. Then if p_2 sends the message m_1, it moves to q_1, and either p_0 receives m_1 and moves to q_2 or p_1 receives m_1 and moves to q_0. Pairwise rendezvous systems and asynchronous shared memory systems with locks are equally expressive [49]. The complexity of the coverability problem for systems with one controller and an arbitrary number of user processes $A\|B^n$ is EXPSPACE-complete [49, 59]. On the other hand for systems without a controller the problem is in PTIME [59]. Furthermore

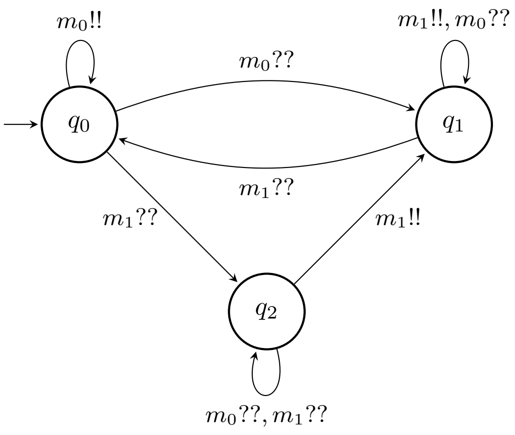


Figure 1.4: Process Template with Broadcast Synchronization

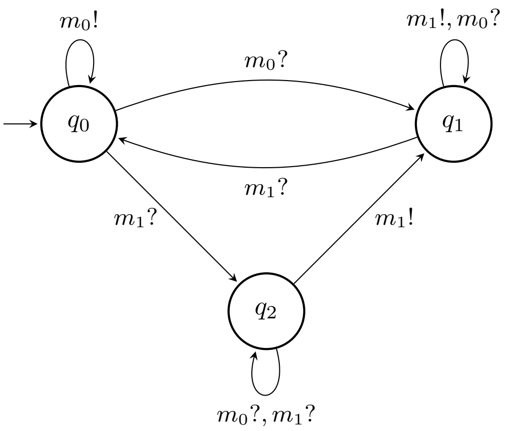


Figure 1.5: Process Template with Pairwise Synchronization

German and Sistla [59] showed the model checking problem for liveness properties is decidable. Another interesting result is due to Aminof et al. [7], who showed that systems without controller have no cutoff in general for properties in $\mathsf{LTL}\backslash\mathbf{X}$.

Global Guards.

Guarded protocols are asynchronous systems characterized by their guarded transitions. Each transition is labeled with a guard g, where g is a statement over other processes' states. For instance, in Figure 1.6, the transition from state q_0 to state q_1 is labeled with the guard $\forall(q_0 \vee q_2) \wedge \exists q_2$. That is, a process p in q_0 can move to q_1 if and only if all other processes in the system are either in q_0 or in q_2 and at least one other process is in q_2. Unfortunately the parameterized model checking problem for guarded protocols is undecidable in general [44]. However for systems restricted to only *disjunctive* guards the

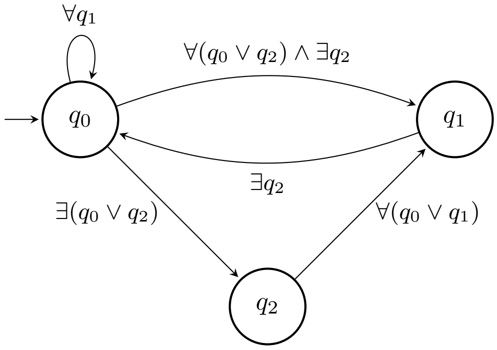


Figure 1.6: Process Template with Global Guards

problem is decidable, where a disjunctive guard is any guard of the form $\exists(q_{i_0} \vee q_{i_1} \vee \ldots \vee q_{i_m})$. A disjunctive guard is enabled (i.e. can be executed) if there exists some process that satisfies the guard. In Figure 1.6, the transitions (q_1, q_0), (q_0, q_2) are labeled with disjunctive guards. The guards of transitions (q_0, q_0), (q_0, q_1), and (q_2, q_1) are not disjunctive.
Furthermore, Emerson and Kahlon [43] showed that disjunctive systems have a cutoff. Additionally, Aminof et al. [7] showed that for a disjunctive system that consists of one A process and an arbitrary number of B processes there exists a constructible Büchi automaton of size $O(|A| \times 2^{|B|})$ that accepts exactly the destuttered executions. On the other hand, Emerson and Kahlon proved that the PMCP for systems with conjunctive guards is undecidable [44], where a conjunctive guard is any guard of the form
$\forall(q_{i_0} \vee q_{i_1} \vee \ldots \vee q_{i_m})$. A conjunctive guard is enabled if and only if all processes satisfy the guard. In Figure 1.6, the transitions (q_2, q_1), (q_0, q_0) are labeled with conjunctive guards. All other guards are not conjunctive. However, the authors later showed [43] that the problem becomes decidable if we add some restrictions on the process templates, and they also determined a cutoff. One of these restrictions requires that the initial state of the process template is in every guard.

Token-passing.

In a token passing systems (or short TPS) processes communicate through a token. Each process has three types of transitions, a synchronous transition for the send/receive of the token, asynchronous transition that can be taken if and only if the process possesses the token, and asynchronous transition that can be taken at any time no matter which process has the token. For the synchronous transition exactly two processes are involved, one to send the token and another to receive the token. Clarke et al. [31], and Emerson et al. [45, 46] showed that the parameterized model checking problem for token passing systems is decidable and has a small cutoff that depends only on the property but not on the process template.

1.4 Contributions

Chapter 3 We present our new cutoff results for guarded protocols (with global guards):

- For conjunctive systems, we extend the class of process templates that are supported by cutoff results, giving cutoff results for local deadlock detection in new classes of templates, and include examples. Although we do not solve the general problem, we show that a cutoff for arbitrary conjunctive systems has to be at least quadratic in the size of the template.

- For both conjunctive and disjunctive systems, we show that by more thorough analysis of process templates, specifically the number and the form of transition guards, we can have smaller cutoffs in many cases. This circumvents the tightness results of Außerlechner et al. [10], which state that no smaller cutoffs can exist for the class of all processes of a given size.

- For disjunctive systems, we also extend both the class of process templates and the class of specifications that are supported by cutoff results. We show that systems with finite conjunctions of disjunctive guards are also supported by variations of the existing proof methods, and get cutoff results for these systems. Furthermore, we provide cutoffs that support checking the simultaneous reachability (and repeated reachability) of a target set by *all* processes in a disjunctive system.

Chapter 4 Prompt-LTL is a linear time logic that extends LTL with a *prompt eventually* operator that is satisfied when the desired event happens at *some* time in the future, and there exist a *bound* on the time that can pass before it happens. Prompt-LTL$\backslash\mathbf{X}$ is not a stutter-insensitive logic, since unbounded stuttering could invalidate a promptness property. Therefore we define the notion of *bounded stutter equivalence*, and prove that Prompt-LTL$\backslash\mathbf{X}$ is *bounded stutter insensitive*. Furthermore, we establish a connection between bounded fairness, bounded stutter equivalence, and the satisfaction of Prompt-LTL$\backslash\mathbf{X}$ formulas. Relying on these findings, we investigate existing approaches that solve parameterized model checking by the *cutoff* method and show that in many cases, we can modify these approaches to obtain correctness guarantees for specifications in Prompt-LTL$\backslash\mathbf{X}$. The types of systems for which we prove these results include *guarded protocols*, as introduced by Emerson and Kahlon [43], and *token-passing systems*, as introduced by Emerson and Namjoshi [46] for uni-directional rings, and by Clarke et al. [31] for arbitrary topologies.

Chapter 5 We present our results that enable the repair of concurrent systems with parameterized correctness guarantees:

- We show how to model disjunctive systems as well-structured transition systems. This facilitates parameterized model checking with respect to reachability of error configurations, and also parameterized deadlock detection for these systems.

- We present a general algorithm that employs a parameterized model checker and constraint-based search for candidate solutions to solve the parameterized repair problem. We show how correctness of the algorithm follows from correctness of its parts.

- We show how to employ the algorithm to repair disjunctive systems with respect to reachability (or: coverability) properties. We give a concrete parameterized model checking algorithm and a concrete encoding of constraints based on the results of model checking, guiding the search for candidate repairs.

- We show how the given concrete algorithm can be extended in two orthogonal directions: from reachability to arbitrary safety properties (and even liveness, in some cases), and from disjunctive systems to other types of systems, like pairwise rendez-vous, and broadcast protocols.

Chapter 6 In this chapter we present two lazy synthesis algorithms: a SAT based algorithm which can be used for concurrent systems, and a BDD-based symbolic algorithm that can be used only for monolithic systems. These lazy synthesis algorithms combine a search for candidate solutions with backward model checking of these candidates. The main contribution in this chapter is the BDD-based algorithm which employs a forward/backward technique to search for candidate solutions. This technique allows us to detect small subsets of the winning region that are sufficient to define a winning strategy. As a result, it produces less permissive solutions than the standard approach and can solve certain classes of problems more efficiently. We evaluate a prototype implementation of our BDD-based algorithm on three sets of benchmarks, including the benchmark set of the Reactive Synthesis Competition (SYNTCOMP) 2017 [62]. We show that on many benchmarks our algorithm detects remarkably small subsets of the winning region that are sufficient to solve the synthesis problem: on the benchmark set from SYNTCOMP 2017, the biggest measured difference is by a factor of 10^{68}. Moreover, it solves a number of problem instances that have not been solved by any participant in SYNTCOMP 2017. Finally, we observe a relation between our algorithm and the approach of Dallal et al. [36] for systems with perturbations, and provide the first implementation of their algorithm as a variant of our algorithm. On the SYNTCOMP benchmark set, we show that whenever a given benchmark admits controllers that give stability guarantees under perturbations, then our lazy algorithm will terminate after exploring a small subset of the winning region and can provide quantitative safety guarantees similar to those of Dallal et al. without any additional cost.

Chapter 7 In this chapter, we present AIGEN, a tool for the generation of random transition systems in a symbolic representation. Instead of generating a monolithic transition relation for the system, the tool generates a partitioned system, where each partition is a Boolean update function for a state variable. To generate a Boolean update function randomly, the tool relies on the fact that ROBDDs are canonical representations of Boolean formulas, and utilizes a method that is inspired by data structures used to implement ROBDDs. Alternatively, AIGEN includes an option to generate Boolean functions relying on the canonical DNF (CDNF) representation. To ensure diversity, the tool allows for a uniformly random sampling over the space of all Boolean functions with a given number of variables, and transition systems constructed from these

Boolean functions. Furthermore it gives the user several parameters that can be used to restrict the random generation to certain types of Boolean functions or transition systems. AIGEN can be used to test and evaluate applications that receive Boolean functions (respectively, propositional formulas) or transition systems as input, e.g. SAT solvers, QBF solvers, finite-state model checkers, reactive synthesis tools, or other tools that analyze qualitative or quantitative properties of transition systems.

1.5 Research Papers

The thesis is based on the following research papers:

Peer-Reviewed Publications

- Swen Jacobs, Mouhammad Sakr. **AIGEN: Random Generation of Symbolic Transition Systems.** 33rd International Conference on Computer Aided Verification (CAV 2021).

- Swen Jacobs, Mouhammad Sakr, and Martin Zimmerman (2020). **Promptness and Bounded Fairness in Concurrent and Parameterized Systems.** In proceedings of the 21st International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2020.

- Swen Jacobs, Mouhammad Sakr (2020). **A Symbolic Algorithm for Lazy Synthesis of Eager Strategies.** In Acta Informatica volume 57 (2020).

- Swen Jacobs, Mouhammad Sakr (2018). **Analyzing Guarded Protocols: Better Cutoffs, More Systems, More Expressivity.** In proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2018.

- Swen Jacobs, Mouhammad Sakr (2018). **A Symbolic Algorithm for Lazy Synthesis of Eager Strategies.** In proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis, ATVA 2018.

Under Submission

- Swen Jacobs, Mouhammad Sakr. **Parameterized Repair of Concurrent Systems.**

Chapter 2

Guarded Protocols and Parameterized Model checking

In this chapter, we introduce a system model for a class of concurrent systems called guarded protocols. Furthermore, we define formally parameterized specifications, parameterized model checking, and other notions that will be used throughout the thesis. The results we present in this thesis are not restricted to guarded protocols, however some of our findings are of interest for other classes of concurrent and parameterized systems, e.g., broadcast protocols, pairwise rendezvous systems (Chapter 5), and token-passing systems (Chapter 4).

2.1 System Model

A *transition system* is a standard model in computer science to describe the behavior of a system. It is basically a directed graph where nodes model states, and edges represent transitions, i.e., state changes. A state describes information about a system at a specific moment, and a transition describes how a system evolves from one state to another. For instance, in a synchronous hardware circuit, a state represents the current values of the registers and the input bits, and a transition models the change of the registers and output bits on the current set of inputs. We say that a transition system is finite if the number of its states is finite. On the other hand a concurrent system consists of many processes running in parallel where each process may implement a different transition system, which we denote in this context by a *process template.*

Let Q be a finite set of states, and let $\mathcal{G} \subseteq \{\exists, \forall\} \times 2^Q$ be a set of guards. **Processes.** A *process template* is a transition system $U = (Q_U, \mathsf{init}_U, \delta_U)$ with[1]

- $Q_U \subseteq Q$ is a finite set of states including the initial state init_U,

[1] In contrast to Außerlechner et al. [10], for simplicity we only consider closed process templates. However, our results extend to open process templates in the same way as explained there.

- $\delta_U : Q_U \times 2^Q \times Q_U$ is a guarded transition relation.

Define the size of U as $|U| = |Q_U|$. An instance of template U will be called a *U-process.*

Concurrency and interleaving. A concurrent system can be modeled as a single sequential non-deterministic system with *interleaving semantics.* In such a model, instead of perceiving the system as a set of standalone processes, it would be modeled as a single global system, where a global state is composed of the current individual states of the different processes. Additionally, actions are interleaved. i.e. actions of different processes do not occur at the same time except for rendezvous or broadcast actions. Hence, at each global step of the system one or more processes are involved but not necessarily all of them.

Guarded Protocols. Guarded protocols are systems that comprise an arbitrary number of processes, where each is an instance of a finite-state process template. The process templates can be conceived as synchronization skeletons [42], that is, they only need to model the characteristics of the system components that are relevant for their synchronization. Processes use global guards for communication, where guards are statements about other processes that are construed either disjunctively ("there exists at least one process that satisfies the guard") or conjunctively ("all other processes satisfy the guard"). Formally, a *guarded protocol* is a system $A\|B^n$, consisting of one copy of a process template A (denoted A-process) and n copies of a process template B (denoted B-processes) running in parallel asynchronously (interleaving semantics). For this section, we assume that n is a fixed positive integer. By similar arguments as in Emerson and Kahlon [43], our results can be extended to systems with an arbitrary number of process templates. We use the templates as indices in order to identify items that belong to these templates. For instance, for a process template $U \in \{A, B\}$, Q_U is the set of states of U. We assume that $Q = Q_A \dot{\cup} Q_B$. Different copies of process template B are distinguished by subscript, i.e., for $i \in [1..n]$, B_i is the ith instance of B, and q_{B_i} is a state of B_i. A state of the A-process is denoted by q_A. We denote the set $\{A, B_1, \ldots, B_n\}$ as $\mathcal{P}$, and write p for a process in $\mathcal{P}$. For $U \in \{A, B\}$, we write $\mathcal{G}_U$ for the set of non-trivial guards that are used in δ_U, i.e., guards that are not in $\{\exists, \forall\} \times \{Q, \emptyset\}$. We assume that $\mathcal{G} = \mathcal{G}_A \cup \mathcal{G}_B$.

The semantics of $A\|B^n$ is given by the transition system $(S, \mathsf{init}_s, \Delta)$, where

- $S = Q_A \times (Q_B)^n$ is the set of (global) states,
- $\mathsf{init}_S = (\mathsf{init}_A, \mathsf{init}_B, \ldots, \mathsf{init}_B)$ is the global initial state, and
- $\Delta \subseteq S \times S$ is the global transition relation. Δ will be defined by local guarded transitions of the process templates A and B in the following.

For a global state $s \in S$ and $p \in \mathcal{P}$, let the *local state of p in s* be the projection of s onto that process, denoted $s(p)$. Then a local transition (q, g, q') of process $p \in \mathcal{P}$ is *enabled* in global state s if $s(p) = q$ and either

- $g = (\exists, G)$ and $\exists p' \in \mathcal{P} \setminus \{p\} : s(p') \in G$, or
- $g = (\forall, G)$ and $\forall p' \in \mathcal{P} \setminus \{p\} : s(p') \in G$.

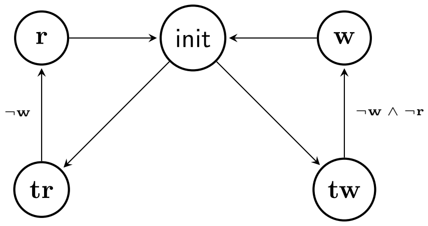


Figure 2.1: Conjunctive Reader-Writer protocol

Finally, $(s, s') \in \Delta$ if there exists $p \in \mathcal{P}$ such that $(s(p), g, s'(p)) \in \delta_p$ is enabled in s, and $s(p') = s'(p')$ for all $p' \in \mathcal{P} \setminus \{p\}$. We say that the transition (s, s') *is based* on the local transition $(s(p), g, s'(p))$ of p. Let $\mathsf{Set}(s) = \{q \in Q \mid \exists p \in \mathcal{P} : s(p) = q\}$, and let $s(p_1, \ldots, p_k)$ be the projection of s onto the processes $p_1, \ldots, p_k \in \mathcal{P}$. A process is *enabled* in s if at least one of its transitions is enabled in s, otherwise it is *disabled*.

Disjunctive and Conjunctive Systems. We distinguish disjunctive and conjunctive systems, as defined by Emerson and Kahlon [43]. In a *disjunctive process template*, every guard is of the form $(\exists, G)$ for some $G \in 2^Q$. In a *conjunctive process template*, every guard is of the form $(\forall, G)$, and $\{\mathsf{init}_A, \mathsf{init}_B\} \subseteq G$, i.e., initial states act as neutral states for all transitions. A *disjunctive system* consists of only disjunctive process templates. A *conjunctive system* consists of only conjunctive process templates. Like Emerson and Kahlon [43], we assume that in conjunctive systems init_A and init_B are contained in all guards, i.e., they act as neutral states. For conjunctive systems, we call a guard *k-conjunctive* if it is of the form $Q \setminus \{q_1, \ldots, q_k\}$ for some $q_1, \ldots, q_k \in Q$. A state q is *k-conjunctive* if all non-trivial guards of transitions from q are k'-conjunctive with $k' \leq k$. A conjunctive system is *k-conjunctive* if every state is k-conjunctive. Whenever the type (conjunctive or disjunctive) of the considered system is stated or clear the quantification symbol will be omitted from the guard and therefore guards will be considered as set of states, and $\mathcal{G}$ will be treated as the power set of Q.

Example 2. Consider the conjunctive system in Figure 2.1. It simulates a reader-writer protocol that models access to data shared between processes. A process that wants to read the data enters state tr ("try-read"). From tr, it can move to the reading state r. However, this transition is guarded by a statement $\neg w$. Formally, guards are sets of states, $\neg w$ stands for the set of all states except w (equivalent to $(\forall, \{r, tr, init, tw\})$), and $\neg w \wedge \neg r$ stands for the set of all states except w and r (equivalent to $(\forall, \{tr, init, tw\})$). Furthermore, this is a conjunctive system, which means that a guard is interpreted as "all other processes have to be in the given set of states". Thus, to take the transition from tr to r, no other process should currently be in state w, i.e., writing the data. Similarly, a process that wants to enter w has to go through tw, but the transition into w is only enabled if no state is reading or writing.

Consider the disjunctive system $W \| R^n$, where W is a writer process (Figure 2.3), and R is a reader process (Figure 2.2). A process in the "non-read" state nr can move to the "read" state r if and only if there is a process in the "non-write" state nw. it can stay in r as long as nw is occupied otherwise it will have to leave the "read" state. Furthermore a process can stay in the "write" state w as long as r is occupied by another process.

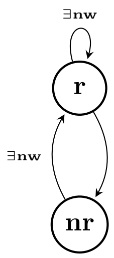


Figure 2.2: Reader

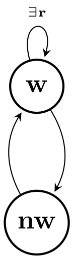


Figure 2.3: Writer

2.1.1 Runs

A *path* of a system $A\|B^n$ is a sequence of global states $x = s_0, s_1, \dots$ such that for all $m < |x|$ there is a transition $(s_m, s_{m+1}) \in \Delta$ based on a local transition of some process p. We say that process p *moves* at *moment* m. A system *run* is a path starting in the initial state init_S, and cannot be extended, namely, it is either infinite or ends in a global state where no local transition is enabled, also called a *deadlock.* We say that a run is *initializing* if every process that moves infinitely often also visits its initial state infinitely often.

Given a system path $x = s_0, s_1, \dots$ and a process p, the *local path* of p in x is the projection $x(p) = s_1(p), s_2(p), \dots$ of x onto local states of p. A local path $x(p)$ is a *local run* if x is a run. Moreover, we denote by $x(p_1, \dots, p_k)$ the projection $s_0(p_1, \dots, p_k)s_1(p_1, \dots, p_k)\dots$ of x onto the processes $p_1, \dots, p_k \in \mathcal{P}$.

Deadlocks. A run is *globally deadlocked* if it ends in a global state where no local transition is enabled, i.e. it is finite. A run is *locally deadlocked* if it is infinite, and there exists a process p and a time m such that p is disabled in $s_{m'}$ for all $m' \geq m$. A run is *deadlocked* if it is either locally or globally deadlocked. A system *has a (local/global) deadlock* if it has a (locally/globally) deadlocked run. Note that if a system has no local deadlocks then it has no global deadlocks, however the other way around is not true.

Fairness. We consider four notions of fairness. A run is called

- *strongly fair* if for every process p, if p is enabled infinitely often, then p moves infinitely often.

- *unconditionally fair*, denoted u-fair(x), if every process moves infinitely often.

- *globally b-bounded fair*, denoted b-gfair(x), for some $b \in \mathbb{N}$, if

$$\forall p \in \mathcal{P}\ \forall m \in \mathbb{N}\ \exists j \in \mathbb{N} : m \leq j \leq m + b \text{ and } p \text{ moves at moment } j.$$

- *locally b-bounded fair* for $E \subseteq \mathcal{P}$, denoted b-lfair(x, E), if it is unconditionally fair and

$$\forall p \in E\ \forall m \in \mathbb{N}\ \exists j \in \mathbb{N} : m \leq j \leq m + b \text{ and } p \text{ moves at moment } j.$$

Remark 1. We analyze these different notions of fairness for the following reason: we are interested in unconditionally fair runs of the system, which requires an assumption about scheduling. However, directly assuming unconditional fairness is too strong, since any run with a local deadlock will violate the assumption, and therefore satisfy the overall specification. Thus, we will consider strong fairness as an assumption on the scheduler, and absence of local deadlocks as a property of the system that has to be proved. Together, they imply unconditional fairness. Furthermore bounded fairness is essential when we analyze systems against specifications that impose a bound on the on the satisfaction of the eventuality (check Chapter 4).

Additional Definitions. Fix a run $x = s_0 s_1 ...$ of the disjunctive system $A \| B^n$. Our constructions are based on the following definitions, where $q \in Q_B$:

- $\mathsf{appears}^{B_i}(q)$ is the set of all moments in x where process B_i is in state q: $\mathsf{appears}^{B_i}(q) = \{m \in \mathbb{N} \mid s_m(B_i) = q\}$.

- $\mathsf{appears}(q)$ is the set of all moments in x where at least one B-process is in state q: $\mathsf{appears}(q) = \{m \in \mathbb{N} \mid \exists i \in \{1, \ldots, n\} : s_m(B_i) = q\}$.

- f_q is the first moment in x where q appears: $f_q = min(\mathsf{appears}(q))$, and $\mathsf{first}_q \in \{1, \ldots, n\}$ is the index of a B-process where q appears first, i.e., with $s_{f_q}(B_{\mathsf{first}_q}) = q$.

- if $\mathsf{appears}(q)$ is finite, then $l_q = max(\mathsf{appears}(q))$ is the last moment where q appears, and $\mathsf{last}_q \in \{1, \ldots, n\}$ is a process index with $s_{l_q}(B_{\mathsf{last}_q}) = q$

- let $\mathsf{Visited}(x)$ be the set of B-states that appeared in the run x. Formally, $\mathsf{Visited}(x) = \{q \in Q_B \mid \mathsf{appears}(q) \neq \emptyset\}$.

- let $\mathsf{Visited}_F(x)$ be the set of states in $F \subseteq Q_B$ that appeared in the run x. Formally, $\mathsf{Visited}_F(x) = \{q \in \mathsf{Visited}(x) \mid q \in F\}$.

- let $\mathsf{Visited}^{inf}(x)$ be the set of B-states that appeared infinitely often in the run x. Formally, $\mathsf{Visited}^{inf}(x) = \{q \in Q_B \mid \exists B_i \in \{B_2, \ldots, B_n\} : \mathsf{appears}^{B_i}(q) \text{ is infinite}\}$.

- let $\mathsf{Visited}^{fin}(x)$ be the set of B-states that appeared finitely often in the run x. Formally, $\mathsf{Visited}^{fin}(x) = \{q \in Q_B \mid \forall B_i \in \{B_2, \ldots, B_n\} : \mathsf{appears}^{B_i}(q) \text{ is finite}\}$.

- $\mathsf{Set}(s_i)$ is the set of all states that are visited by some process at moment i: $\mathsf{Set}(s_i) = \{q | q \in (Q_A \cup Q_B) \text{ and } \exists p \in \mathcal{P} : s_i(p) = q\}$.

2.2 Specifications

A specification is a set of properties to be satisfied by a process or a system. These properties are usually used as input to a theorem prover, a model checker, or a synthesis tool. In this thesis we are concerned with temporal properties like reachability, safety, liveness, and fairness.

2.2.1 Temporal logic

Temporal logic is used for specifying properties over time. Basically it extends classical propositional logic with operators that refer to the behavior of systems over time. It allows for the specification of a range of important system properties such as functional correctness (does the system behave as it is supposed?), reachability (is it possible to reach a deadlock state?), safety ("something bad never occurs"), liveness ("something good must eventually happen").

Propositional Logic. The set of propositional logic formulas over a set of atomic propositions AP is defined by the following grammar:

$$\phi ::= true \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2, \text{ where } a \in AP$$

A propositional formula is a proposition that might hold or not, depending on which of the atomic propositions are assumed to hold. The formula a states that the proposition a holds. The symbol $\wedge$ is conjunction, that is, $\phi_1 \wedge \phi_2$ holds if and only if both ϕ_1 and ϕ_2 hold. The symbol $\neg$ stands for negation, i.e., $\neg\phi$ holds if and only if ϕ does not hold. The constant *true* is a proposition that holds in any context, independent of the valuation of other atomic propositions. Additionally, we use the constant $false$ for $\neg true$.

Furthermore, we assume that time is discrete, and we differentiate between two types of temporal logic, linear and branching. In *linear temporal logic* at each moment in time there is a single successor moment, thus it has the ability to reason about a single timeline. Whereas in *branching temporal logic* time may split into alternative courses, hence it can reason about multiple timelines.

Linear Temporal Logic (or short LTL) extends propositional logic with a set of operators for constructing linear time properties. A linear time property can be viewed as a set of infinite words, where each word describes a valid run. LTL formulas over a set of atomic proposition AP are defined according to the following grammar:

$$\varphi ::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2, \text{ where } a \in \mathsf{AP}$$

An LTL formula can be satisfied by a word over 2^{AP}. This word can be seen as a run (infinite sequence of states) of a system. The symbol $\mathbf{X}$ denotes the "next" operator, then a run x satisfies $\mathbf{X}\varphi$ (denoted by $x \models \mathbf{X}\varphi$) if and only if φ holds at the next state. Also the symbol $\mathbf{U}$ denotes the "until" operator, that is, $x \models \varphi_1 \mathbf{U} \varphi_2$ if and only if φ_1 holds in x at least until φ_2 holds. The until operator allows to derive the temporal operators $\mathbf{F}$ ("eventually", sometimes in the future) and $\mathbf{G}$ ("always", from now until forever) as follows: $\mathbf{F}\varphi = true\mathbf{U}\varphi$, and $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$ (or $\mathbf{G}\varphi = \varphi\mathbf{U}false$). We use the notation $\mathsf{LTL}\backslash\mathbf{X}$ to refer to LTL without the "next" operator $\mathbf{X}$. A full formal definition for LTL is omitted and can be found in [85].

Branching Temporal logic. In LTL, we can state properties over all possible runs that start in a specific state, but not about a proper subset of the runs. To overcome such problem, branching temporal logic extends LTL with path quantifier operators. The existential path quantifier (denoted E) and the universal path quantifier (denoted A). Computation Tree logic (or CTL) is a branching time logic where its formulas are defined over a set of atomic proposition AP according to the following grammar:

$$\varphi ::= \mathit{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{E}\,\mathbf{X}\varphi \mid \mathsf{E}[\varphi_1 \mathbf{U} \varphi_2],\ \text{where } a \in \mathsf{AP}$$

For instance, the property $\mathsf{E}\,\mathbf{X}\varphi$ denotes that there exists a run along which φ holds at the next state. In other words, it states that there is at least one possible run in which φ holds in the next state. The formula $\mathsf{A}\,\mathbf{X}\varphi$ is equivalent to $\neg\,\mathsf{E}\,\mathbf{X}\neg\varphi$. Branching time means that at each moment there may be several different possible futures. Therefore the semantics of a branching temporal logic is interpreted in terms of an infinite directed tree of states rather than an infinite sequence. A full formal definition for CTL is omitted and can be found in [42].

Büchi automaton. For every LTL formula φ, there exists a Büchi automaton that accepts exactly all words that satisfy φ [94]. A (non-deterministic) *Büchi automaton.* is a tuple $\mathcal{A} = (\Sigma, Q_{\mathcal{A}}, \delta, a_0, \alpha)$, where:

- Σ is a finite alphabet,
- $Q_{\mathcal{A}}$ is a finite set of states,
- $\delta : Q_{\mathcal{A}} \times \Sigma \to 2^{Q_{\mathcal{A}}}$ is a transition function,
- $a_0 \in Q_{\mathcal{A}}$ is an initial state, and
- $\alpha \subseteq Q_{\mathcal{A}}$ is a Büchi acceptance condition.

$\mathcal{A}$ accepts all the words (runs) in which at least one of the infinitely often occurring states is in α.

We denote by $\mathcal{A}_\varphi$ the Büchi automaton that accepts exactly all words (runs) that satisfy φ.

Parameterized Specifications. A *parameterized specification* is a temporal logic formula with indexed atomic propositions and quantification over indices. Let $\varphi(A, B_{i_1}, \ldots, B_{i_k})$ be a temporal logic formula over atomic propositions from Q_A and indexed propositions from $Q_B \times \{i_1, \ldots, i_k\}$. A *k-indexed formula* is of the form $\forall i_1, \ldots, i_k . \varphi(A, B_{i_1}, \ldots, B_{i_k})$. For given $n \geq k$, by symmetry of guarded protocols (cp. Emerson and Kahlon [43]) we have

$$A \| B^n \models \forall i_1, \ldots, i_k . \varphi(A, B_{i_1}, \ldots, B_{i_k}) \quad \text{iff} \quad A \| B^n \models \varphi(A, B_1, \ldots, B_k).$$

The latter formula is denoted by $\varphi(A, B^{(k)})$, and we will use it instead of the original $\forall i_1, \ldots, i_k . \varphi(A, B_{i_1}, ..., B_{i_k})$.

Example. Consider again the conjunctive system in Figure 2.1, then the temporal specification below states that on all paths, it is always the case that whenever a process wants to read it will eventually read and whenever a process wants to write it will eventually be granted the access to the write state.

$$\forall i.\, \mathsf{A}\,\mathbf{G}\left((tr_i \to \mathbf{F} r_i) \wedge (tw_i \to \mathbf{F} w_i)\right),$$

2.3 Model Checking Problems and Cutoffs

For a given system $A \| B^n$ and a temporal logic formula $\varphi(A, B^{(k)})$ with $n \geq k$,

- the *model checking problem* is to decide whether $A \| B^n \models \varphi(A, B^{(k)})$,

- the (global/local) *deadlock detection problem* is to decide whether $A\|B^n$ has (global/local) deadlocks,
- the *parameterized model checking problem* (PMCP) is to decide whether $\forall m \geq n :\ A\|B^m \models \varphi(A, B^{(k)})$, and
- the *parameterized (local/global) deadlock detection problem* is to decide whether for some $m \geq n$ the system $A\|B^m$ has (global/local) deadlocks.

We will add later different notions of fairness to these definitions. According to Remark 1 about fairness, we are interested in proving the absence of local deadlocks under the assumption of strong fairness, which implies unconditional fairness and therefore allows us to separately prove the satisfaction of a temporal logic specification under the assumption of unconditional fairness.

Cutoffs. We define cutoffs with respect to a class of systems (either disjunctive or conjunctive), a class of process templates T, and a class of properties, which can be k-indexed formulas for some $k \in \mathbb{N}$ or the existence of (local/global) deadlocks. A *cutoff* for a given class of properties and a class of systems with processes from T is a number $c \in \mathbb{N}$ such that for all $A, B \in T$, all properties φ in the given class, and all $n \geq c$:

$$A\|B^n \models \varphi \;\Leftrightarrow\; A\|B^c \models \varphi.$$

Like the problem definitions above, cutoffs may additionally be flavored with different notions of fairness.

Cutoffs and Decidability. Note that the existence of a cutoff implies that the parameterized model checking and parameterized deadlock detection problems are *decidable* iff their non-parameterized versions are decidable.

Chapter 3

Better Cutoffs for Guarded Protocols

The parameterized model checking problem for systems that combine disjunctive and conjunctive guards is in general undecidable [47]. For this reason, research in the literature has considered systems that have a single type of guard, i.e. conjunctive or disjunctive systems. Emerson and Kahlon [43, 44] have studied these classes of systems and showed that they have cutoffs that depend on the number of states in the process templates. These results are for specifications of the form $\forall \bar{p}.\ \mathsf{A}\,\Phi(\bar{p})$, where $\Phi(\bar{p})$ is an $\mathsf{LTL}\backslash\mathbf{X}$ property over the local states of one or more processes $\bar{p}$. Außerlechner et al. [10] have extended and enhanced these results, but a number of issues remain open.

For instance, consider the reader-writer protocol in Figure 3.1 and the following parameterized safety specifications:

$$\forall i \neq j.\, \mathsf{A}\,\mathbf{G}\,(\neg(w_i \wedge w_j) \wedge \neg(w_i \wedge r_j))\,,$$

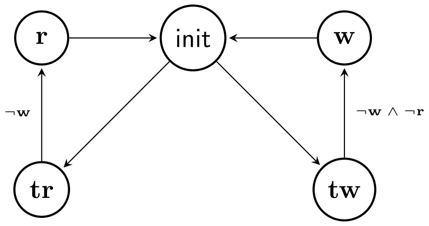
As explained earlier, indices i and j refer to processes in the system. This condition states that it is never the case that two processes are writing at the same time and it is never the case that two processes are reading and writing simultaneously. Emerson and Kahlon [43] proved that, for conjunctive systems, properties from $\mathsf{LTL}\backslash\mathbf{X}$ have a cutoff of $k{+}1$ for properties with k index variables. Furthermore, they proved that for deadlock detection, a cutoff linear in the size of the process template is sufficient.

However, Außerlechner et al. [10] observed that for liveness properties such as

$$\forall i.\, \mathsf{A}\,\mathbf{G}\,((tr_i \rightarrow \mathsf{F}\,r_i) \wedge (tw_i \rightarrow \mathsf{F}\,w_i))\,,$$

an explicit treatment of fairness assumptions on the scheduling of processes is required. They noticed that, in guarded protocols, local deadlock detection has to be considered to adequately treat liveness properties although the cutoff results for $\mathsf{LTL}\backslash\mathbf{X}$ properties still holds under fairness assumptions. They showed that for local deadlock detection a cutoff that is linear in the size of the process template is enough, but under a major restriction where the cutoff only supports systems with 1-conjunctive guards, i.e., where each guard can only exclude a single state. Unfortunately, the reader-writer example in Figure 3.1 is not supported by these results, since the guard $\neg w \wedge \neg r$ excludes 2 states.

Figure 3.1: Conjunctive Reader-Writer protocol



Another drawback of the existing results is that they take into consideration only minimal knowledge about the process templates, e.g., their size and the interpretation of guards. Therefore many cutoffs rely directly on the size of the process template. Intuitively, for the cutoff, the *communication* between processes should be equally or even more important than the process templates internal state space. This can also be noticed in the example above: only 2 out of the 5 states can be observed by the other processes, and can hence affect their behavior. In this chapter, we will explore the idea of cutoffs that depend on the type and number of guards in the process templates.

In this chapter, we expand, for conjunctive systems, the class of process templates that have cutoff results, providing cutoff results for local deadlock detection in classes of templates that are not 1-conjunctive, and include systems like the one in Figure 3.1. Although we do not solve the general problem, we show that a cutoff for an arbitrary conjunctive system is at least quadratic in the size of the process template. Additionally, for conjunctive and disjunctive systems, we give a cutoff that is linear in the number of transition guards in the process template which in many cases can be smaller than the cutoff obtained by Emerson and Kahlon [43] and Außerlechner et al. [10]. This circumvents the tightness results of Außerlechner et al. [10], which state that no smaller cutoffs can exist for the class of all processes of a given size. Furthermore, we show that these new cutoff results can be extended to the class of process templates with a conjunction of disjunctive guards and to a new class of specifications.

Outline of the Chapter. This chapter is organized as follows: In Section 3.1 we present our new results for global and local deadlock detection in conjunctive systems, and compare them to previously existing results. Using these new results we show in Section 3.2 how we can verify the reader-writer example in Figure 3.1. In Section 3.3 we state our new cutoff results for disjunctive systems, and we show how these results can be extended to the class of process templates with a conjunction of disjunctive guards and to a new class of specifications. Finally, Section 3.4 draws some conclusion.

3.1 New Cutoff Results for Conjunctive Systems

In this section, we state our new results for conjunctive systems, and compare them to the previously known results in Table 3.1. We give improved cutoffs for global deadlock detection in general (Section 3.1.1), and for local deadlock detection for the restricted case of 1-conjunctive systems (Section 3.1.2). After that, we explain why local deadlock detection in general is hard, and identify a number of cases where we can solve the problem even for systems that are not 1-conjunctive (Sections 3.1.3 and 3.1.4). We do not improve on the cutoffs for $\mathsf{LTL}\backslash\mathbf{X}$ properties, since they are already very small for conjunctive systems and only depend on the number of index variables in the specification.

Additional Definitions.

To analyze deadlocks in a given conjunctive system $A \| B^n$, we introduce additional definitions. A *deadset* of a local state q is a minimal set $D \subseteq Q$ that block all outgoing transitions of any process that is currently in local state q. Formally, we say that $D \subseteq Q$ is a *deadset* of $q \in Q$ if:

i) $\forall (q, g, q') \in \delta_U : \exists q'' \in D : q'' \notin g$,

ii) D contains at most one state from Q_A, and

iii) there is no D' that satisfies i) and ii) with $D' \subset D$.

For a given local state q, $dead^{\wedge}_q$ is the set of all deadsets of q:

$$dead^{\wedge}_q = \{D \subseteq Q \mid D \text{ is a deadset of } q\}.$$

If $dead^{\wedge}_q = \emptyset$, then we say q is *free*. We say that a state q is *non-blocking* if $\forall q' \in Q \, \forall D \in dead^{\wedge}_{q'} : q \notin D$. Informally, q is non-blocking if it does not appear in $dead^{\wedge}_{q'}$ for any $q' \in Q$. We say that a state q is *not self-blocking* if $\forall D \in dead^{\wedge}_q : q \notin D$. Informally q is not self-blocking if it does not appear in $dead^{\wedge}_q$.

In these terms, a globally deadlocked run is a run that ends in a global state s such that for every local state q of s, there is a process p deadlocked in q, and there exists some $D \in dead^{\wedge}_q$ such that $D \subseteq \mathsf{Set}(s(\mathcal{P} \setminus p))$. Similarly, a locally deadlocked run is a run such that one process p will eventually always remain in state q, and from some point on, we always have $D \subseteq \mathsf{Set}(s(\mathcal{P} \setminus p))$ for some $D \in dead^{\wedge}_q$. Note that in this case, it can happen that there does not exist a single deadset D that is contained in $\mathsf{Set}(s)$ all the time, but the run may *alternate* between different deadsets of q that are contained in $\mathsf{Set}(s)$ at different times. We say that a locally deadlocked run is *alternation-bounded* if it does not alternate infinitely often between different deadsets. We say that a system *has alternation-bounded local deadlocks* if, whenever there exists a locally deadlocked run, there also exists an alternation-bounded locally deadlocked run.

From now on we say that a run is $non\text{-}fair$ if it is not necessarily fair.

3.1.1 Global Deadlock Detection

For global deadlock detection, we show how to obtain improved cutoffs based on the number of free, non-blocking, and not self-blocking states in a given process template.

Theorem 1. *For conjunctive systems and process templates A, B, let*

- *$k_1 = |D_1|$, where $D_1 \subseteq Q_B$ is the set of free states in B,*
- *$k_2 = |D_2 \setminus D_1|$, where $D_2 \subseteq Q_B$ is the set of non-blocking states in B, and*

- $k_3 = |D_3 \setminus (D_1 \cup D_2)|$, *where* $D_3 \subseteq Q_B$ *is the set of not self-blocking states in* B.

Then for $c = 2|B| - 2k_1 - 2k_2 - k_3$ *we have:*
$(\forall n \geq c : A||B^n$ *has no global deadlock* $) \Leftrightarrow A||B^c$ *has no global deadlock.*

Proof Sketch. In the following, we prove that a globally deadlocked run of a large system $A\|B^n$ can be simulated in the cutoff system $A\|B^c$ for all $n \geq c$.

In order to simulate a globally deadlocked run $x = s_0, s_1, \ldots, s_m$ of a large system by a run y in the cutoff system, by Emerson and Kahlon [43] the following is sufficient. We analyze the set of local states $q \in Q$ that are present in the final state s_m of x, and distinguish whether any such q appears once in s_m, or multiple times. If q appears once, we identify one local run of x that ends in q, and replicate it in the cutoff system. If q appears multiple times, we do the same for two local runs of x that end in q. This construction ensures that in the resulting global run $x' = s'_0, \ldots, s'_m$ of the cutoff system, for any point in time t, we have $\mathsf{Set}(s'_t) \subseteq \mathsf{Set}(s_t)$, and for any deadlocked process at time t we have $\mathsf{Set}(s'_t(\mathcal{P} \setminus p)) \subseteq \mathsf{Set}(s_t(\mathcal{P} \setminus p))$. Therefore, all transitions in x' will be enabled, and x' is deadlocked in s'_m. If x' does not contain all local runs of x then there are stuttering steps in x', where no process moves. By removing these stuttering steps, we obtain the desired run y.

The construction of Emerson and Kahlon assumes that in the worst case all local states of B appear in the deadlocked state s_m. However, if $D_1 \subseteq Q_B$ are *free* local states, then we know that no state from D_1 can ever appear in s_m, and thus the cutoff is reduced by $2|D_1|$. Similarly, if $D_2 \subseteq Q_B$ are *non-blocking* states, then we know that no state from D_2 can be necessary for the deadlock in s_m, and therefore the construction will also work if we remove the local runs ending in D_2. This also reduces the cutoff by $2|D_2|$. Moreover, the original construction assumes that all local states q may be self-blocking, which requires the second local run that ends in q. If we know that $D_3 \subseteq Q_B$ are *not self-blocking*, then we only need one local run for each of these states, reducing the cutoff by $|D_3|$. If we combine all three cases, we get the statement of the theorem. □

Note that the sets of free, non-blocking, and not self-blocking states can be identified by a simple analysis of a single process template, and the cost of this analysis is negligible compared to the cost of a higher cutoff in verification of the system.

3.1.2 Local Deadlock Detection in 1-conjunctive Systems

For local deadlock detection, we first show that smaller cutoffs can be found by taking into account the transitions and guards of the process template. For a 1-conjunctive process template $U \in \{A, B\}$, let $\mathcal{G}_{U,B}$ be the set of guards of U that exclude one of the states of B, i.e., that are of the form $g = Q \setminus \{q\}$ for some $q \in Q_B$ (guards not in this set exclude only states of A). Furthermore, let $\mathsf{maxD}_U = max\{|D \cap Q_B| \mid D \in dead^{\wedge}_q \text{ for some } q \in Q_U\}$ be the maximal number of states from B that appear in any deadset of a state in U.

Theorem 2. *For conjunctive systems with process templates A, B, if process template $U \in \{A, B\}$ is 1-conjunctive, then the following are cutoffs for local deadlock detection in a U-process in non-fair runs:*

- $\mathsf{maxD}_U + 2$, *and*
- $|\mathcal{G}_{U,B}| + 2$.

Proof. To show the existence of a cutoff c, we prove that a locally deadlocked run of the cutoff system $A\|B^c$ can be simulated in a large system $A\|B^n$ for all $n \geq c$ (i.e., monotonicity), and we prove that a locally deadlocked run of a large system $A\|B^n$ can be simulated in the cutoff system $A\|B^c$ for all $n \geq c$ (i.e., bounding).

Monotonicity. A globally deadlocked run $x = s_0, s_1, \ldots, s_m$ of $A\|B^n$ can be simulated by a run y of $A\|B^{n+1}$ as follows: Let y replicate the run x, and keeps the new process stuttering in the initial state. Let m be the moment when the deadlock happens in x, then let the new process behave arbitrarily after the moment m. In the remaining cutoff proofs of this chapter we omit the monotonicity part as they are all similar to this one.

Bounding. In order to simulate a locally deadlocked run $x = s_0, s_1, \ldots$ of a large system by a run y in the cutoff system, the following construction has been presented by Außerlechner et al. [9] for non-fair runs. Suppose process p is locally deadlocked in local state q after the system has entered state s_m. We first replicate (copy) the local runs of A and p. Since the system is 1-conjunctive, every local state has a unique deadset. For each q' in the deadset D of q, we copy a local run from x that is in q' at time m, and modify it such that it stays in q' forever after this point in time. Thus, the process in q is locally deadlocked because all states in D will be present at any time after m. Finally, we copy one additional local run of a process that moves infinitely often in x (this process should be different than those that have been copied before). As in the proof of Theorem 1, all transitions of the resulting global run x' will be enabled, and we can obtain the desired run y by de-stuttering.

Note that the original proof uses one process for every state in the unique deadset D of the deadlocked local state q, and assumes that in the worst case we have $D \supseteq Q_B$, resulting in the cutoff of $|Q_B|+2$ for all process templates with a given set of states Q_B. However, if we take into account the guards of transitions and the individual deadsets, we can obtain smaller cutoffs: in particular, instead of assuming that the size of some deadset is $|Q_B|$, we can compute the maximal size of actual deadsets maxD_U, and replace $|Q_B|$ by maxD_U to obtain a cutoff of $\mathsf{maxD}_U + 2$. Further, note that (since the system is 1-conjunctive) maxD_U is bounded by $|\mathcal{G}_{U,B}|$, so $|\mathcal{G}_{U,B}| + 2$ also is a cutoff. □

Theorem 3. *For conjunctive systems and process templates A, B, if process template $U \in \{A, B\}$ is 1-conjunctive, then $2|\mathcal{G}_{U,B}|$ is a cutoff for local deadlock detection in a U-process in strongly fair runs.*

Proof Sketch. For fair runs, the construction by Außerlechner et al. [9] is similar as in the previous proof, but additionally we need to ensure that all processes either move infinitely often or are locally deadlocked. We explain the original construction in a new way that highlights our insight.

Let $x = s_0, s_1, \ldots, s_m$ be a locally deadlocked run of a large system, then we construct a run y in the cutoff system as follows: First, identify all states

$q' \in Q_B$ such that there exists a locally deadlocked local run in x that eventually stays in q'. For each of these states, copy this local run, and if q' is self-blocking then also copy another local run from x that eventually visits the state q' and stays there (note that if q' is self-blocking then there is at least 2 local runs that are deadlocked in q'). To ensure that these local runs are locally deadlocked also in the constructed run, add the states in their deadsets to a set of states D. Note that only states that are excluded in one of the (1-conjunctive) guards can be added to D. Note also that for each state, in which we have a deadlocked process, we have copied up to two local runs from x. Thus, the size of D is bounded by $|\mathcal{G}_{U,B}|$, and in the worst case we have added $2|\mathcal{G}_{U,B}|$ processes until now.

Then, let $D' \subseteq D$ be the set of states for which no process has been added thus far, and let m' be the time when all local runs that have been added until now are locally deadlocked. Copy for each of the states $q' \in D'$ one local run from x that is in q' at time m', and add a process that stays in init_B until time m'. Then after moment m' we can let all processes that are in D' move in the following way: (i) choose some $q' \in D'$ (ii) let the process that is in init_B move to q' (iii) let the process that was waiting in q' move to init_B (iv) repeat with fair choices of $q' \in D'$. Since each of these states must appear in x at any time after m' without a process being locally deadlocked in the state, there must be a local path from this state to itself in one of the local runs in x. Since for fair conjunctive systems we assume that they are initializing, this path must go through init_B, and the construction is guaranteed to work.

Note that overall, for each state in D we have copied either one or two local runs from x, so the bound for the number of these processes is still $2|\mathcal{G}_{U,B}|$. Also note that the additional process that waits in init_U is only needed if at least one of the other processes is not locally deadlocked, thus it does not increase the needed number of processes. Finally, for the original locally deadlocked process we can distinguish two cases: i) if we have added $2|\mathcal{G}_{U,B}|$ processes thus far, then the original process is deadlocked in a state that does not block any transition, and we can remove it since the run will exhibit a local deadlock regardless, or ii) if this is not the case, then even with the original process we need at most $2|\mathcal{G}_{U,B}|$ processes overall. □

Note that in a 1-conjunctive process template U, we have $|\mathcal{G}_{U,B}| \leq |Q_B| - 1$ (the initial state can never be excluded from a guard). Thus, our new cutoffs are always smaller or equal to the known cutoff from Außerlechner et al. [9].

3.1.3 Local Deadlock Detection: Beyond 1-conjunctive Systems

While Theorems 2 and 3 improve on the local deadlock detection cutoff for conjunctive systems in some cases, the results are still restricted to 1-conjunctive process templates. The reason for this restriction is that when going beyond 1-conjunctive systems, the local deadlock detection cutoff (even without considering fairness) can be shown to grow at least quadratically in the number of states or guards, and it becomes very hard to determine a cutoff.

To analyze these cases, define the following:

- Given a process template $U \in \{A, B\}$, a sequence of local states $q_1, \ldots, q_n$ is *connected* if $\forall q_i \in \{q_1, \ldots, q_n\} : \exists (q_i, g_i, q_{i+1}) \in \delta_U$.

Table 3.1: Cutoff Results for Conjunctive Systems

		EK [43]	AJK [10]	our work
k-indexed LTL\X	non-fair	$k+1$	$k+1$	unchanged
k-indexed LTL\X	fair	-	$k+1$	unchanged
Local Deadlock	non-fair	-	$\lvert B\rvert + 1^*$	$\lvert\mathcal{G}_{U,B}\rvert + 2^*$
Local Deadlock	fair	-	$2\lvert B\rvert - 2^{**}$	$2\lvert\mathcal{G}_{U,B}\rvert^{**}$
Global Deadlock		$2\lvert B\rvert + 1$	$2\lvert B\rvert - 2$	$2\lvert B\rvert - 2k_1 - 2k_2 - k_3$

* : systems need to have alternation-bounded local deadlocks (see Sect. 3.1.4)
** : systems need to be initializing and have alternation-bounded local deadlocks
k_1: number of free states
k_2: number of non-blocking states that are not free
k_3: number of not self-blocking states that are not free or non-blocking

- A *cycle* is a connected sequence of states $q, q_1, \ldots, q_n, q$ such that $\forall q_i, q_j \in \{q_1, \ldots, q_n\} : q_i \neq q_j$. We denote such a cycle by C_q.
- We abuse the notation and use C_q for the set of states on the cycle C_q.
- A *cycle* C_q is called *free* if $\forall q' \in C_q \setminus q \; \forall g \in \mathcal{G}_{C_q} : q' \in g$.

Example 3. If we consider the process template in Figure 3.2 without the dashed parts, then it exhibits a local deadlock in state q_l for 9 processes (in a strongly fair run), but not for 8 processes: one process has to move to q_l, which has four deadsets: $\{a, c\}$, $\{a, d\}$, $\{b, c\}$, and $\{b, d\}$. To preserve a deadlock in q_l, the processes need to alternate between different deadsets while always at least covering one of them. To achieve this, for each cycle that starts and ends in states a, b, c, d, we need 2 processes that move along the cycle to keep all guards of q_l covered at all times. Intuitively, one process per cycle has to be in the state of interest, or ready to enter it, and the other process is traveling on the cycle, waiting until the guards are satisfied.

Now, consider the modified template (including the dashed parts) where we i) add two states e, f in a similar way as a, b, c, d, ii) add a new state connected to q_l with guard $\neg e \wedge \neg f$, and iii) change the guards in the sequence from u_1 to init to $\neg a \wedge \neg c \wedge \neg e$ and $\neg b \wedge \neg d \wedge \neg f$, respectively. Then we have 6 cycles that need 2 processes each, and we need 13 processes to reach a local deadlock in q_l.

Moreover, consider the modified template where we increase the length of the path from u_1 to init by adding states u_3 and u_4, such that we obtain a sequence $(u_1, u_2, u_3, u_4, \mathsf{init})$, where transitions alternate between the two guards from the original sequence. Then, for every cycle we need 3 processes instead of 2, as otherwise they cannot traverse the cycle fast enough to ensure that the local deadlock is preserved infinitely long. That is, the template with both modifications now needs 19 processes to reach a local deadlock.

Observe that by increasing the height of the template, we increase the necessary number of processes without increasing the number of different guards. Moreover, when increasing both the width and height of the template, the number of processes that are necessary for a local deadlock increases quadratically with the size of the template.

This example leads us to the following result.

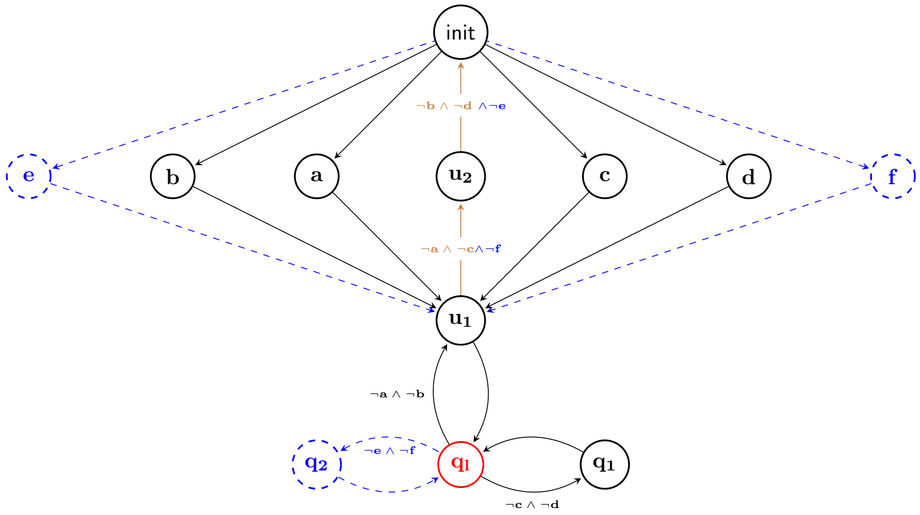


Figure 3.2: Process Template with Quadratic Cutoff for Local Deadlocks

Theorem 4. *A cutoff for local deadlock detection for the class of all conjunctive systems must grow at least quadratically in the number of states. Furthermore, it cannot be bounded by the number of guards at all.*

Proof Sketch. For a system that does exhibit a local deadlock for some size n, but not for $n-1$, the cutoff cannot be smaller than n. Thus, the example shows that a cutoff for local deadlock detection in general is independent of the number of guards, and must grow at least quadratic in the size of the template. □

Cutoffs that can in the best case be bounded by $|B|^2$ will not be very useful in practice. Therefore, instead of solving the general problem, we identify in the following a number of cases where the cutoff remains linear in the number of states or guards.

3.1.4 Systems with Alternation-bounded Local Deadlocks

When comparing the proof of Theorem 2 to Example 3, we note that the reason that the cutoff in Theorem 2 does not apply is the following: while in 1-conjunctive systems every state has a unique deadset, in the general case a state may have many deadsets, and the structure of the process template may require infinitely many alternations between different deadsets to preserve the local deadlock. Moreover, as shown in the example, the number of processes needed to alternate between deadsets may increase with the size of the template, even if the set of guards (and thus, the number of different deadsets) remains the same.

However, we can still obtain small cutoffs in some cases, based on the following observation: even if states have multiple deadsets, an infinite alternation between them may not be necessary to obtain a local deadlock. In the following, we will first show that for systems where infinite alternation between different deadsets is not necessary, the cutoff for 1-conjunctive systems applies, and then give a number of sufficient conditions to identify such systems.

Alternation-bounded Local Deadlocks. We say that a run $x = s_0, s_1, \ldots$ where process p is locally deadlocked in state q *is alternation-bounded* if there is a moment m and a single set $D \subseteq Q_B$ such that for all $m' > m$: $D \subseteq \mathsf{Set}(s_{m'}(\mathcal{P} \setminus q))$ and for some $q_A \in Q_A$, $D \cup q_A$ is a deadset of q. Intuitively, this means the B-states in the deadset that preserves the deadlock only change finitely often.

For $q \in Q$, we say that q *has alternation-bounded local deadlocks for* $c \in \mathbb{N}$ if the following holds for all $n \geq c$:

if $\quad A \| B^n$ has a local deadlock in q
then $\quad A \| B^n$ has an alternation-bounded local deadlock in q.

Let $\mathcal{G}_{U,B*}$ be the set of guards of U that exclude at least one state of B, i.e., that are of the form $g = Q \setminus \{Q_1\}$ with $Q_1 \subseteq \{Q_A \cup Q_B\}$ and $Q_1 \cap Q_B \neq \emptyset$.

Theorem 5. *For conjunctive systems and process templates A, B, the cutoffs for non-fair runs is $|\mathcal{G}_{U,B*}| + 2$, and the cutoff for strongly fair runs is $2|\mathcal{G}_{U,B*}|$ if every $q \in Q$ has alternation-bounded local deadlocks for the cutoff value. In particular, this implies that the parameterized local deadlock detection problem is decidable.*

Proof Sketch. Suppose in run x of $A\|B^n$, with n greater than the cutoff value, process p is locally deadlocked in local state $q \in Q$, and q has alternation-bounded local deadlocks. Then there exists an alternation-bounded run x' of $A\|B^n$ in which p is locally deadlocked in q. That is, either the local deadlock in x' eventually is preserved by a sequence of deadsets with unique restriction to B-states, or a number of processes that is bounded by the size of the largest deadset is sufficient to preserve the local deadlock in q. In the latter case, we are done. In the former case, based on the set D, the run x' can be simulated with the same constructions as in the proofs of Theorems 2 and 3. □

Sufficient Conditions for Alternation-bounded Local Deadlocks. In the following, we will identify four sufficient conditions that imply that a state q has alternation-bounded local deadlocks, and that can easily be checked directly on the process template.

Effectively 1-conjunctive states. We say that a state q is *effectively* 1-*conjunctive* if it either has only 1-conjunctive guards or is free.

Lemma 1. *If $q \in Q$ is effectively* 1*-conjunctive, then it has alternation-bounded local deadlocks for $c = 1$.*

Proof Sketch. If q is 1-conjunctive, then it has alternation-bounded local deadlocks since it has only a single deadset. If q is free, then a local deadlock in q is not possible, so the condition holds vacuously. □

In the reader-writer example of Figure 3.1, all states except tw are effectively 1-conjunctive.

Relaxing 1-conjunctiveness. For $q \in Q$, let $\mathcal{G}_q$ be the set of non-trivial guards in transitions from q. We say that state q is *relaxed* 1*-conjunctive* if $\mathcal{G}_q$ only contains guards of the form $Q \setminus \{q_1, \ldots, q_k\}$, where either

- at most one of the q_i is from Q_B, or
- whenever more than one q_i is from Q_B, then $\mathcal{G}_q$ must also contain a guard of the form $Q \setminus \{q'_1, \ldots, q'_k, q_i\}$ for one of these q_i and where all q'_j are from Q_A.

Lemma 2. *If $q \in Q$ is relaxed 1-conjunctive, then it has alternation-bounded local deadlocks for $c = 1$.*

Proof Sketch. Note that the guards we allow on transitions from a relaxed 1-conjunctive state each have at most one state from Q_B that can block the transition. Thus q does not necessarily have a unique deadset, but for each deadset D the restriction to states of B is unique. Thus, every run that is locally deadlocked in q will be alternation-bounded. □

Alternation-free. We say that a state $q \in Q$ *is alternation-free* if the following condition holds: if $D = \{q' \mid \forall g \in \mathcal{G}_\sigma \ : q' \notin g\}$, i.e., if D is the set of local states that disables the k-conjunctive guards with $k > 1$ in transitions from q, then there is at most one $q' \in D$ for which the following does not hold:

For all cycles $C_{q'} = q', \ldots, q' \in U$:

- $q \in C_{q'}$, or
- for all $g \in \mathcal{G}_{C_{q'}} : q' \notin g$, or there is a $g' \supseteq g$ with $g' \in \mathcal{G}_q$.

Intuitively, this means that there is at most one state $q' \in D$ such that you can leave it and return to it without breaking the local deadlock — and at least two such states would be needed to alternate between different deadsets.

In the reader-writer example of Figure 3.1 state tw is alternation-free: i) $\{w, r\}$ is the set of states that disables the only guard that is not 1-conjunctive, and ii) all cycles that start and end in w contain also tw.

The following lemma directly follows from the explanation above.

Lemma 3. *If $q \in Q$ is alternation-free, then it has alternation-bounded local deadlocks for $c = 1$.*

Process templates with freely traversable lassos. While the three conditions above guarantee the existence of a fair alternation-bounded run if the original run was fair, the following condition in general returns a run that is not strongly fair. A *lasso lo* is a connected sequence of local states $q_0, \ldots, q_i, \ldots, q_k$ such that $q_0 = \mathsf{init}$ and $q_i, \ldots, q_k$ is a cycle. We denote by $\mathcal{G}_{lo}$ the set of guards of the transitions on lo. We say that a lasso lo is *freely traversable* with respect to a state $q \in Q$ if it does not contain q and for every deadset D of q, every $g \in \mathcal{G}_{lo}$ contains $D \cup \{q\}$. Intuitively, these conditions ensure that lo can be executed after the system has reached any of the (minimal) deadlock configurations for q.

Lemma 4. *If there exists a freely traversable lasso in B with respect to $q \in Q$, then q has alternation-bounded local deadlocks in non-fair runs for $c = \mathsf{maxD}_U + 2$.*

Proof Sketch. Suppose there exists a freely traversable lasso with respect to q, and x is a run where process p is locally deadlocked in q, where p is not enabled anymore after time m. Then we obtain an alternation-bounded locally deadlocked run x' by picking a deadset D of q with $D \subseteq \mathsf{Set}(s_m(\mathcal{P} \setminus p))$ and for every $q' \in D$ a local run from x that is in q' at time m. Since $n \geq c = \mathsf{maxD}_U + 2$, there is at least one other process in $A \| B^n$. We replace the local run of this process with a local run that stays in init_B until m, and after m is the only process that moves, along the freely traversable lasso we assumed to exist. Any further local runs stay in init_U forever. □

Theorem 5 and Lemmas 1 to 4 allow us to analyze the process templates, state by state, and to conclude the existence of a small cutoff for local deadlock detection in certain cases. The lemmas provide sufficient but not necessary conditions for the existence of alternation-bounded cutoffs. They provide a template for obtaining small cutoffs in certain cases, and for a given application they may be refined depending on domain-specific knowledge.

3.1.5 Local Deadlock Detection under Infinite Alternation

For systems that do not have alternation-bounded local deadlocks, it is very difficult to obtain cutoff results. For example, for systems that have no restrictions on the guards, one can show that a cutoff based on the number of guards in general cannot exist (Example 3). Moreover, the cutoff grows at least linearly in the number of states, or, more precisely, in the number of alternations between different deadsets that are necessary to traverse a cycle C_q for a state q from a deadset.

3.2 Verification of the Reader-Writer Example

We consider the reader-writer in Figure 3.1, and show how our new results allow us to check correctness, find a bug, and check a fixed version.

With our results, we can for the first time check the given liveness property in a meaningful way, i.e., under the assumption of fair scheduling. Since all states in the process template have alternation-bounded local deadlocks for $c = 1$, by Theorems 5 the local deadlock detection cutoff for the system is $2|\mathcal{G}_{B,B}| = 4$. No cutoff for this problem was known before. Moreover, compared to previous results we reduce the cutoff for global deadlock detection by recognizing that $k_1 = 3$ states can never be deadlocked, and $k_2 = 2$ additional states never appear in any guard. This reduces the cutoff to $2|B| - 2k_1 - 2k_2 = 10 - 6 - 4 = 0$, i.e., we detect that there are no global deadlocks without further analysis.

However, checking the system for local deadlocks shows that a local deadlock is possible: a process may forever be stuck in tw if the other processes move in a loop $(\mathsf{init}, tr, r)^\omega$ (and always at least one process is in r). To fix this, we can add an additional guard $\neg tw$ to the transition from init to tr, as shown in the process template in Figure 3.3. For the resulting system, our results give a local deadlock detection cutoff of $2|\mathcal{G}_{B,B}| = 6$, and a global deadlock detection cutoff of $2|B| - 2k_1 - 2k_2 - k_3 = 10 - 6 - 2 - 1 = 1$ (where k_3 is the number of states that do appear in guards and could be deadlocked themselves, but do not have a transition that is blocked by another process in the same state).

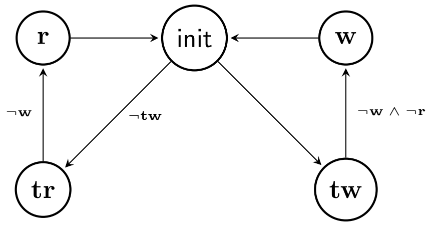


Figure 3.3: Error-Free Reader Writer Protocol

3.3 New Cutoff Results for Disjunctive Systems

In this section, we state our new cutoff results for disjunctive systems, and compare them to the previously known results in Table 3.2. Moreover, we show two extensions of the class of problems for which cutoffs are available:

1. systems where transitions are guarded with a conjunction of disjunctive guards (Section 3.3.4), and
2. two important classes of specifications that cannot be expressed in prenex indexed temporal logic (Section 3.3.5).

To state our results, we need the following additional definitions. Fix process templates A, B. Let $B_{\mathcal{G}} = \{q \in Q_B \mid \exists g \in \mathcal{G} : q \in g\}$, i.e., the set of B-states that appear on a guard. For a state $q \in Q_B$ in a disjunctive system, define $\mathsf{Enable}_q = \{q' \in Q \mid \exists (q, g, q'') \in \delta_B : q' \in g\}$, i.e., the set of states of A and B that enable a transition from q.

3.3.1 Linear-Time Properties

In this section we consider *k-indexed properties* Φ_k of the form

$$\forall i_1, \ldots, i_k.\, \mathsf{A}\, h(A, B_{i_1}, \ldots, B_{i_k})$$

or

$$\forall i_1, \ldots, i_k.\, \mathsf{E}\, h(A, B_{i_1}, \ldots, B_{i_k})$$

where $h(A, B_{i_1}, \ldots, B_{i_k})$ is an $\mathsf{LTL}\backslash\mathbf{X}$ formula over atomic propositions from Q_A and indexed propositions from $Q_B \times \{i_1, \ldots, i_k\}$. As mentioned in Chapter 2 by symmetry of guarded protocols we have:

$$A \| B^n \models \forall i_1, \ldots, i_k.\, \mathsf{A}\, h(A, B_{i_1}, \ldots, B_{i_k}) \quad \text{iff} \quad A \| B^n \models \mathsf{A}\, h(A, B_1, \ldots, B_k).$$

Theorem 6. *For disjunctive systems, process templates A, B, and k-indexed properties Φ_k:*

- *$|\mathcal{G}| + k + 1$ and $|B_{\mathcal{G}}| + k + 1$ are cutoffs in non-fair runs,*
- *$|B_{\mathcal{G}}| + |\mathcal{G}| + k$ and $2|B_{\mathcal{G}}| + k$ are cutoffs in unconditionally fair runs.*

We prove the theorem by proving two lemmas. We will use some of the notations that are defined in Section 2.1.1.

Lemma 5 (Monotonicity Lemma)**.** *Let A, B be disjunctive process templates, $n \geq k$, and Φ_k a k-indexed property. Then:*

$$(A\|B^n \not\models \Phi_k) \Rightarrow \big(A\|B^{n+1} \not\models \Phi_k\big)$$

Proof. Let x be a run of $A\|B^n$ that violates Φ_k, then we construct a run y of $A\|B^{n+1}$ that also violates Φ_k as follows: Let $y(A) = x(A)$ and $y(B_j) = x(B_j)$ for all $B_j \in \{B_1, \ldots, B_n\}$ and let the new process B_{n+1} copy one of the B-processes of $A\|B^n$, i.e., $y(B_{n+1}) = x(B_i)$ for some $i \in \{1, \ldots, n\}$. Copying a local run violates the interleaving semantics as two processes will be moving at the same time. To solve this problem, we split every transition (s'_l, s'_{l+1}) where the interleaving semantics is violated by B_i and B_{n+1} executing local transitions (q_i, g, q'_i) and (q_{n+1}, g, q'_{n+1}), respectively. To do this, replace (s'_l, s'_{l+1}) with two consecutive transitions $(s'_l, u)(u, s'_{l+1})$, where (s'_l, u) is based on the local transition (q_i, g, q'_i) and (u, s'_{l+1}) is based on the local transition (q_{n+1}, g, q'_{n+1}). Note that both of these local transitions are enabled in the constructed run y since the transition (q_i, g, q'_i) is enabled in the original run x. Hence, we have $y \in A\|B^{n+1}$ and $A\|B^{n+1} \not\models \Phi_k$ Moreover, we have run y is unconditional fair if and only if the run x is unconditional fair. □

Lemma 6 (Bounding Lemma)**.** *Let A, B be disjunctive process templates, $n \geq c$, and Φ_k a k-indexed property. Then:*

$$(A\|B^n \not\models \Phi_k) \Rightarrow (A\|B^c \not\models \Phi_k)$$

Proof. Let x be a run of $A\|B^n$ that violates Φ_k. Emerson and Kahlon [43] showed how to construct a non-fair run y in the cutoff system $(A\|B^c)$ that does not satisfy Φ_k. The run y includes the local runs $x(A), x(B_1), \ldots, x(B_k)$, and additional runs that ensure that all the transitions are enabled: for every state $q \in Q_B$ that appears in x and the local run that first visits q, we add the prefix of that local run up to q, and then let it stay in q forever. One additional local run may have to be copied from x to ensure that the resulting run is infinite. Thus, the produced cutoff is $c = |B| + k + 1$ in non-fair runs. □

Based on the above and an analysis of the process template B, we can find better cutoffs for non-fair runs:
Let x be a run of $A\|B^n$. We show first how to construct a non-fair run y of $A\|B^c$ such that $x(A, B_1, \ldots, B_k)$ and $y(A, B_1, \ldots, B_k)$ are equivalent up to stuttering.

Construction 1:

1. $y(A) = x(A)$, and $\forall l \in \{1, \ldots, k\}$ $y(B_l) = x(B_l)$.
2. **(Flooding)**: To every $q \in \mathsf{Visited}_{B_\mathcal{G}}(x)$, devote one process B_{i_q} that copies B_{first_q} until the time f_q, then stutters in q forever. Formally:

$$y(B_{i_q}) = x(B_{\mathsf{first}_q})[0 : \infty]$$

3. Establish interleaving semantics.

After copying the runs needed to reproduce the counter example in step 1, step 2 statically checks which states of the run do appear in a guard ($\mathsf{Visited}_{B_\mathcal{G}}(x)$)), and conclude that only those need to be copied. This reduces the cutoff to $c = |B_{\mathcal{G}}| + k + 1$. Furthermore, as a second option and since our goal is to enable all transitions, it is also sufficient to only copy a local run for one *representative* state of each guard (the one that is visited first in x). In this way, we need at most one additional process per guard in B, i.e., $c = |\mathcal{G}| + k + 1$ also is a cutoff for non-fair runs. Note that the last step (Establish interleaving semantics) is required due to the fact that we might copy a local run of a process multiple times.

The following construction, introduced by Außerlechner et al. [10], builds on the steps explained above, and additionally preserves unconditional fairness in a given run.

Construction 2:

1. $y(A) = x(A)$, and $\forall l \in \{1, \ldots, k\}$ $y(B_l) = x(B_l)$.
2. **(Flooding with evacuation)**: To every $q \in \mathsf{Visited}^{fin}(x)$, devote one process B_{i_q} that copies B_{first_q} until the time f_q, then stutters in q until time l_q where it starts copying B_{last_q} forever. Formally:
$$y(B_{i_q}) = x(B_{\mathsf{first}_q})[0 : f_q].(q)^{l_q - f_q}.x(B_{last_q})[l_q + 1 : \infty]$$
3. **(Flooding with fair extension)**: For every $q \in \mathsf{Visited}^{inf}(x)$, let B_q^{inf} be a process that visits q infinitely often in x. We devote to q two processes $B_{i_{q_1}}$ and $B_{i_{q_2}}$ that both copy B_{first_q} until the time f_q, and then stutter in q until B_q^{inf} reaches q for the first time. After that, let $B_{i_{q_1}}$ and $B_{i_{q_2}}$ copy B_q^{inf} in turns as follows: $B_{i_{q_1}}$ copies B_q^{inf} until it reaches q while $B_{i_{q_2}}$ stutters in q, then $B_{i_{q_2}}$ copies B_q^{inf} until it reaches q while $B_{i_{q_1}}$ stutters in q and so on.
4. Establish interleaving semantics as in the proof of Lemma 7.

This construction gives a cutoff of $c = 2|B| + k - 1$, since in the worst case all states appear infinitely often and we need two copies for each, but at least one of them must also appear infinitely often in the k processes that have to satisfy the specification.

However, similarly to the non-fair case, an analysis of the template gives us better cutoffs. As a first approximation, we can again limit the construction to states in $B_{\mathcal{G}}$, and obtain the cutoff $c = 2|B_{\mathcal{G}}| + k$ (now we can not assume that one of the infinite states in the constructed run also appears in the k processes). Moreover, from the states in $B_{\mathcal{G}}$ that appear infinitely often we can again choose one representative for each guard, and only add two local runs for each representative. This does not work for the processes that are visited finitely often, since we need to move them into an infinitely visited state to ensure fairness, and then need a different representative. To compute the cutoff, suppose f states from $B_{\mathcal{G}}$ are visited finitely often, and i states infinitely often. From the latter, there are r states for which we added two local runs, with $r \leq |\mathcal{G}|$ and $r \leq i$. Then we need at most $f + 2r + k$ local runs (including the k processes that satisfy the specification). However, we have $f \leq |B_{\mathcal{G}}| - i$, and therefore $f + 2r + k \leq |B_{\mathcal{G}}| - i + 2r + k \leq |B_{\mathcal{G}}| + r + k \leq |B_{\mathcal{G}}| + |\mathcal{G}| + k$.

3.3.2 Global Deadlock Detection

Let $\mathcal{N} = \{q \in Q_B \mid q \in \mathsf{Enable}_q\}$, and let $\mathcal{N}^*$ be the maximal subset (wrt. number of elements) of $\mathcal{N}$ such that $\forall q_i, q_j \in \mathcal{N}^* : q_i \notin \mathsf{Enable}_{q_j} \land q_j \notin \mathsf{Enable}_{q_i}$.

Theorem 7. *For disjunctive systems and process templates A, B, $|B_\mathcal{G}| + |\mathcal{N}^*|$ is a cutoff for global deadlock detection.*

Proof Sketch. To construct a globally deadlocked run in the cutoff system, for each state from $\mathcal{N}$ that appears in the deadlock, we copy the according local run. To simulate the remaining part of x, we use the same construction as for fair runs in the proof of Theorem 6, except that local states that appear in the deadlock are considered to be visited infinitely often (and we don't need the fair extension of runs after reaching the state). Thus, the resulting run will be globally deadlocked, and all transitions up to the deadlock will be enabled. The number of local runs is bounded by $|\mathcal{N}| + f + i$, where i is the number of states from $|B_\mathcal{G}|$ that appear in the deadlock and are not in $\mathcal{N}$, and f is the number of states from $|B_\mathcal{G}|$ that appear in the run, but not in the deadlock. Since $f + i \leq |B_\mathcal{G}|$ and $\mathcal{N}^*$ is the maximal subset of $\mathcal{N}$ that can appear together in a global deadlock, the number of needed local runs is bounded by $|\mathcal{N}^*| + |B_\mathcal{G}|$. □

Remark. To compute $\mathcal{N}^*$ exactly, we need to find the smallest set of states in $\mathcal{N}$ that do not satisfy the additional condition. This amounts to finding the minimum vertex cover (MVC) for the graph with vertices from $\mathcal{N}$ and edges from q_i to q_j if $q_i \in \mathsf{Enable}_{q_j}$.

3.3.3 Local Deadlock Detection

Theorem 8. *For disjunctive systems and process templates A, B:*

- *$m + |\mathcal{G}| + 1$ is a cutoff for local deadlock detection in non-fair runs, where $m = \max_{q \in Q_B^*}\{|\mathsf{Enable}_q|\}$ for $Q_B^* = \{q \in Q_B \mid |\mathsf{Enable}_q| < |B|\}$,*
- *$|B_\mathcal{G}| + |\mathcal{G}| + 1$ and $2|B_\mathcal{G}| + 1$ are cutoffs for local deadlock detection in unconditionally fair runs.*

Proof Sketch. Based on what we have already shown, the fair case is simpler: we copy the local runs of A and the deadlocked process, and for the other processes use the same construction as in the fair case of Theorem 6. The local deadlock is preserved since states that appear finitely often in the original run also appear finitely often in the constructed run, and the cutoffs are $2|B_\mathcal{G}| + 1$ and $|B_\mathcal{G}| + |\mathcal{G}| + 1$.

For the non-fair case, we use a combination of the constructions for the fair and non-fair case from Theorem 6: if in run x a process is locally deadlocked in local state q, then for states in Enable_q that appear in x we use the construction for finitely appearing states in fair runs. For the remaining states, we use the non-fair construction, i.e., we find one representative per guard and stay there forever, except that representatives now can never be from Enable_q. The construction ensures that all transitions that are taken are enabled, and eventually all transitions from q are disabled. Since m gives a bound on the number of states that can be in Enable_q, the cutoff we get is $m + |\mathcal{G}| + 1$. □

Table 3.2: Cutoff Results for Disjunctive Systems with $m < |B|$, $|\mathcal{N}^*| < |B|$, and $min = min(|\mathcal{G}|, |B_\mathcal{G}|)$

		EK [43]	AJK [10]	our work
k-indexed $LTL \setminus X$	non-fair	$\lvert B\rvert + k + 1$	$\lvert B\rvert + k + 1$	$min + k + 1$
k-indexed $LTL \setminus X$	fair	-	$2\lvert B\rvert + k - 1$	$\lvert B_\mathcal{G}\rvert + min + k$
Local Deadlock	non-fair	-	$\lvert B\rvert + 2$	$m + \lvert\mathcal{G}\rvert + 1$
Local Deadlock	fair	-	$2\lvert B\rvert - 1$	$\lvert B_\mathcal{G}\rvert + min + 1$
Global Deadlock		-	$2\lvert B\rvert - 1$	$\lvert B\rvert + \lvert\mathcal{N}^*\rvert$

3.3.4 Systems with Conjunctions of Disjunctive Guards

We consider systems where a transition can be guarded by a set of sets of states, interpreted as a conjunction of disjunctive guards. I.e., a guard $\{D_1, \ldots, D_n\}$ is satisfied in a given global state if for all $i = 1, \ldots, n$, there exists another process in a state from D_i.

We observe that for this class of systems, most of the original proof ideas still work. For results that depend on the number of guards, we have to count the number of different conjuncts in guards.

Theorem 9. *For systems with conjunctions of disjunctive guards, cutoff results for disjunctive systems that do not depend on the number of guards still hold (first and second column of results in Table 3.2, and cutoffs in the third column that only refer to $|B|_\mathcal{G}$ and constants).*

Cutoff results that depend on the number of guards (last column of Table 3.2) hold if we consider the number of conjuncts in guards instead. For results that additionally refer to some measure of the sets of enabling states (m and $|\mathcal{N}^*|$, respectively), we obtain a valid cutoff for systems with conjunctions of disjunctive guards if we replace this measure by $|B| - 1$.

In particular, the existence of a cutoff implies that the respective PMCP and parameterized deadlock detection problems are decidable.

Proof Ideas. The cutoff results that are independent of the number of guards still hold since all of the original proof constructions still work. To simulate a run x of a large system in a run y the cutoff system, one task is to make sure that all necessary transitions are enabled in the cutoff system. The construction that is used to do this works for conjunctions of disjunctive guards just as well. By a similar argument, deadlocks are preserved in the same way as for disjunctive systems.

For cutoffs that depend on the number of guards, transitions with conjunctions of disjunctive guards require us to use one representative for each conjunct in a guard, in the construction explained in the proof of Theorem 6.

Finally, the reductions of the cutoff based on the analysis of states that can or cannot appear together in a deadlock do not work in these extended systems, and we have to replace m and $|\mathcal{N}^*|$ by $|B| - 1$ in the cutoffs. The reason is that Enable_q is now not a set of states anymore, but a set of sets of states. A more detailed analysis based on this observation may be possible, but is still open. □

3.3.5 Simultaneous Reachability of Target States

An important class of properties for parameterized systems asks for the reachability of a global state where all processes of type B are in a given local state q (compare Delzanno et al. [37]). This can be written in indexed $\mathsf{LTL}\backslash\mathbf{X}$ as $\mathsf{F}\,\forall i.q_i$, but is not expressible in the fragment where index quantifiers have to be in prenex form. We denote this class of specifications as TARGET. Similarly, repeated reachability of q by all states simultaneously can be written $\mathsf{GF}\,\forall i.q_i$, and is also not expressible in prenex form. We denote this class of specifications as REPEAT-TARGET.

Theorem 10 (Disjunctive TARGET and REPEAT-TARGET)**.** *For disjunctive systems: $|B|$ is a cutoff for checking* TARGET *and* REPEAT-TARGET.

In particular, the PMCP with respect to TARGET *and* REPEAT-TARGET *in disjunctive systems is decidable.*

Proof Ideas. We can simulate a run x in a large system where all processes are in q at time m in the cutoff system by first moving one process into each state that appears in x before m, in the same order as in x. To make all processes reach q, we move them out of their respective states in the same order as they have moved out of them in x. For this construction, we need at most $|B|$ processes.

If in x the processes are repeatedly in q at the same time, then we can simulate this also in the cutoff system: if $m' > m$ is a point in time where this happens again, then we use the same construction as above, except that we consider all states that are visited between m and m', and we move to these states from q instead from init. The correctness argument is the same, however.

Finally, if the run with REPEAT-TARGET should be fair, then we do not select *any* m' with the property above, but we choose it such that all processes move between m and m'. If the original run x is fair, then such an m' must exist. □

3.4 Conclusion

We have shown that better cutoffs for guarded protocols can be obtained by analyzing properties of the process templates, in particular the number and form of transition guards. We have further shown that cutoff results for disjunctive systems can be extended to a new class of systems with conjunctions of disjunctive guards, and to specifications TARGET and REPEAT-TARGET, that have not been considered for guarded protocols before.

For conjunctive systems, previous works have treated local deadlock detection only for the restricted case of systems with 1-conjunctive guards. We have considered the general case, and have shown that it is very difficult — the cutoffs grow independently of the number of guards, and at least quadratically in the size of the process template. To circumvent this worst-case behavior, we have identified a number of conditions under which a small cutoff can be obtained even for systems that are not 1-conjunctive.

By providing cutoffs for several problems that were previously not known to be decidable, we have in particular proved their decidability.

Our work is inspired by applications in parameterized synthesis [18, 61], where the goal is to automatically *construct* process templates such that a given specification is satisfied in systems with an arbitrary number of components. In this setting, deadlock detection and expressive specifications are particularly important, since *all* relevant properties of the system have to be specified.

Chapter 4

Promptness and Bounded Fairness

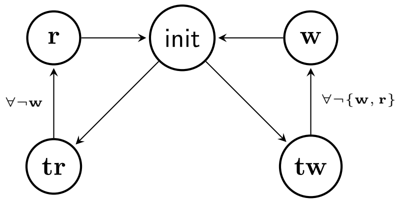
The majority of the existing approaches for parameterized model checking only handle safety properties, or their support for liveness properties is restricted. One of the main reasons is that global fairness properties are either not considered or can't be expressed within the supported logic (cp. Außerlechner et al. [10]). In this chapter we consider liveness properties, including a quantitative version of liveness called *promptness* and we investigate cases in which we can guarantee that a parameterized system satisfies such properties. The idea of promptness is that a desired event shouldn't just occur at *some* time in the future, but there need to exist a *bound* on the time that may pass before it happens.

We consider specifications in Prompt-LTL [76], which is a logic that extends LTL with the prompt eventuality operator that puts a symbolic bound on the satisfaction of the eventuality. Additionally, the model checking problem with respect to Prompt-LTL specifications checks if there exists a value for the symbolic bound such that the property is guaranteed to be satisfied with respect to this value.

In several settings, adding promptness comes totally free in terms of asymptotic complexity [76], for instance, model checking and synthesis of fixed-size systems [68]. It should be noted that Prompt-LTL can be viewed as a fragment of parametric LTL, a logic introduced by Alur et al. [6]. However, given that many decision problems for parametric LTL, including model checking, can be reduced to those for Prompt-LTL, we can limit our focus to the simpler logic. Hence, in this chapter we study *parameterized* model checking for Prompt-LTL and demonstrate that in several cases adding promptness is totally free for this problem.

Particularly, since it is common in the analysis of concurrent systems, we abstract concurrency by an interleaving semantics. For this reason, we restrict our specifications to Prompt-LTL$\backslash\mathbf{X}$, an extension of the stutter-insensitive logic LTL$\backslash\mathbf{X}$ that doesn't include the next-time operator (similarly, in Chapter 3, we considered LTL$\backslash\mathbf{X}$ instead of LTL). The behavior of parameterized concurrent systems with respect to Prompt-LTL$\backslash\mathbf{X}$ specifications has not been investigated in detail before and brings new challenges.

Figure 4.1: Conjunctive Reader-Writer protocol



Example. For instance, consider the reader-writer conjunctive system in Figure 4.1 which simulates access to shared data between processes.

For such systems, as explained in Chapter 3, there are cutoff results for parameterized verification of properties from LTL**X**, e.g.,

$$\forall i. \mathbf{G}\left((tr_i \rightarrow \mathbf{F} r_i) \wedge (tw_i \rightarrow \mathbf{F} w_i) \right),$$

In this chapter we investigate whether the same cutoffs still hold if we replace the LTL eventually operator **F** above with the prompt-eventually operator $\mathbf{F_p}$, while imposing a bounded fairness assumption on the scheduler.

Outline of the Chapter. This chapter is organized as follows: In Section 4.1, we introduce Prompt-LTL**X** and show that it is stutter sensitive. Then we propose the notion *Bounded Stutter Equivalence* and show that Prompt-LTL**X** is bounded stutter insensitive. In Section 4.2, we present our cutoff results for disjunctive systems with respect to specifications in Prompt-LTL**X** and in LTL**X**. In Section 4.3, we investigate cutoff results for conjunctive systems under bounded fairness and stutter-insensitive specifications with promptness. In Section 4.4, we introduce a system model for token passing systems and then show how to obtain cutoff results under Prompt-LTL**X**. Finally, Section 4.5 draws some conclusions.

4.1 Prompt-LTL\X and Bounded Stutter Equivalence

We consider concurrent systems that are represented as an interleaving composition of finite-state transition systems, possibly with synchronizing transitions where multiple processes take a step at the same time. In such systems, a process may stay in the same state for many global transitions while other processes are moving. From the perspective of that process, these are *stuttering steps.*

Stuttering is a well-known phenomenon, and temporal languages that include the next-time operator **X** are *stutter sensitive*: they can require some atomic proposition to hold at the next moment in time, and the insertion of a stuttering step may change whether the formula is satisfied or not. On the other hand, LTL**X**, which does not have the **X** operator, is stutter-insensitive: two words that only differ in stuttering steps cannot be distinguished by the logic [11].

In the following, we introduce Prompt-LTL**X**, an extension of LTL**X**, and investigate its properties with respect to stuttering.

4.1.1 Prompt-LTL\X

Let AP be the set of atomic propositions. The syntax of Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formulas over AP is given by the following grammar:

$$\varphi ::= a \mid \neg a \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{F_p}\varphi \mid \varphi\mathbf{U}\varphi \mid \varphi\mathbf{R}\varphi, \text{ where } a \in AP$$

The semantics of Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formulas is defined over infinite words $w = w_0w_1 \ldots \in (2^{AP})^\omega$, positions $i \in \mathbb{N}$, and bounds $k \in \mathbb{N}$. The semantics of the prompt-eventually operator $\mathbf{F_p}$ is defined as follows:

$$(w, i, k) \models \mathbf{F_p}\varphi \text{ iff there exists } j \text{ such that } i \leq j \leq i + k \text{ and } (w, j, k) \models \varphi.$$

All other operators ignore the bound k and have the same semantics as in LTL, moreover we define $\mathbf{F}$ and $\mathbf{G}$ in terms of $\mathbf{U}$ and $\mathbf{R}$ as usual.

Example Let's examine meticulously the reader-writer protocol in Figure 4.1 (in this example we are considering the fixed size system that consists of a single reader and a single writer), it is easy to see that the system satisfies the Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formula $\mathbf{G}(tw \rightarrow \mathbf{F_p}w)$. However the system doesn't satisfy the formula $\mathbf{F_p}w$ due to the fact that, no matter what bound k we choose, a run of the protocol can keep traversing the sequence of states $(init \rightarrow tr \rightarrow r \rightarrow init)$ for at least $k + 1$ steps and then move to tw and w.

4.1.2 Prompt-LTL and Stuttering

Our first observation is that Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ is stutter sensitive: to satisfy the formula $\varphi = \mathbf{GF_p}q$ with respect to a bound k, q has to appear at least once in every k steps. Given a word w that satisfies φ for some bound k, we can construct a word that does not satisfy φ for any bound k by introducing an increasing (and unbounded) number of stuttering steps between every two appearances of q. In the following, we show that Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ is stutter insensitive if and only if there is a bound on the number of consecutive stuttering steps.

Bounded Stutter Equivalence. A finite word $w \in (2^{AP})^+$ is a *block* if $\exists\alpha \subseteq AP$ such that $w = \alpha^{|w|}$. Two blocks $w, w' \in (2^{AP})^+$ are *d-compatible* if $\exists\alpha \subseteq AP$ such that $w = \alpha^{|w|}, w' = \alpha^{|w'|}$, $|w| \leq d \cdot |w'|$ and $|w'| \leq d \cdot |w|$. Two infinite sequences of blocks $w_0w_1w_2 \ldots$, $w'_0w'_1w'_2 \ldots$ are *d-compatible* if w_i, w'_i are d-compatible for all i.

Two words $w, w' \in (2^{AP})^\omega$ are *d-stutter equivalent*, denoted $w \equiv_d w'$, if they can be written as d-compatible sequences of blocks. They are *bounded stutter equivalent* if they are d-stutter equivalent for some d. We denote by $\hat{w}$ a sequence of blocks that corresponds to a word w.

Example. Consider the two infinite local runs w and w' of the reader-writer protocol depicted in Figure 4.1:

$$w = init, tr, r, r, (init, tw, w)^\omega$$

$$w' == init, init, tr, tr, r, r, r, r, (init, init, tw, tw, w, w)^\omega$$

Then w and w' are 2-stutter equivalent.

Given an infinite sequence of blocks $\hat{w} = w_0, w_1, w_2 \ldots$, let $N_i^{\hat{w}} = \{\sum_{l=0}^{i-1} |w_l|, \ldots, \sum_{l=0}^{i-1} |w_l| + |w_i| - 1\}$ be the set of positions of the ith block. Given a position $n \in \mathbb{N}$, there is a unique i such that $n \in N_i^{\hat{w}}$.

To prove that Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ is *bounded stutter insensitive*, i.e., it cannot distinguish two words that are bounded stutter equivalent, we define a function that maps between positions in two d-compatible sequences of blocks: given two infinite d-stutter equivalent words w, w' such that $\hat{w}, \hat{w}'$ are d-compatible, define the function $f : \mathbb{N} \to 2^{\mathbb{N}}$ where: $f(j) = N_i^{\hat{w}'} \Leftrightarrow j \in N_i^{\hat{w}}$. Note that $\forall j' \in f(j)$ we have $w(j) = w'(j')$, where $w(i)$ denotes the ith symbol in w. For an infinite word w, let $w[i, \infty)$ denote its suffix starting at position i, and $w[i : j]$ its infix starting at i and ending at j. Then we can state the following.

Remark 2. Given two words w and w', if $w \equiv_d w'$, then $\forall j \in \mathbb{N}\ \forall j' \in f(j) : w[j, \infty) \equiv_d w'[j', \infty)$.

Now, we can state our first theorem.

Theorem 11 (Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ is Bounded Stutter Insensitive)**.** *Let w, w' be d-stutter equivalent words, φ a Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formula, and f as defined above. Then $\forall i, k \in \mathbb{N}$:*

$$(w, i, k) \models \varphi \ \Rightarrow\ \forall j \in f(i) : (w', j, d \cdot k) \models \varphi$$

Proof. The proof works inductively over the structure of φ. Let $\hat{w} = w_0, w_1, \ldots$ and $\hat{w}' = w_0', w_1', \ldots$ be two d-compatible sequences of w and w'. We denote by n_i, m_i the number of elements inside $N_i^w, N_i^{w'}$ respectively.

Case 1: $\varphi = a$**.** $(w, i, k) \models \varphi \Leftrightarrow a \in w(i)$. By definition of f we have $\forall j \in f(i) : w(i) = w'(j)$, and thus $\forall j \in f(i) : (w', j, d \cdot k) \models \varphi$.

Case 2: $\varphi = \neg a$**.** $(w, i, k) \models \varphi \Leftrightarrow a \notin w(i)$. By definition of f we have $\forall j \in f(i) : w(i) = w'(j)$, and thus $\forall j \in f(i) : (w', j, d \cdot k) \models \varphi$.

Case 3: $\varphi = \varphi_1 * \varphi_2$ **with** $* \in \{\wedge, \vee\}$**.** $(w, i, k) \models \varphi \Leftrightarrow (w, i, k) \models \varphi_1 * (w, i, k) \models \varphi_2$. By induction hypothesis we have: $\forall j \in f(i)\ (w', j, d \cdot k) \models \varphi_1 * \forall j \in f(0)\ (w', j, d \cdot k) \models \varphi_2 \Leftrightarrow (w', j, d \cdot k) \models \varphi$.

Case 4: $\varphi = \mathbf{F_p}\varphi$**.** $(w, i, k) \models \mathbf{F_p}\varphi \Leftrightarrow \exists e, x : i \leq e \leq i + k,\ e \in N_x^w$, and $(w, e, k) \models \varphi$ where $(\sum_{l=0}^{x-1} n_l) \leq e < (\sum_{l=0}^{x} n_l)$. Then by induction hypothesis we have: $\forall j \in f(e)\ (w', j, d \cdot k) \models \varphi$. Let s be the smallest position in $f(e)$, then $s = \sum_{l=0}^{x-1} m_l$. There exists $y \in \mathbb{N}$ s.t. $i \in N_y^w$ then $s = \sum_{l=0}^{y-1} m_l + \sum_{l=y}^{x-1} m_l \leq \sum_{l=0}^{y-1} m_l + \sum_{l=y}^{x-1} n_l.d \leq \sum_{l=0}^{y-1} m_l + d.(\sum_{l=y}^{x-1} n_l) \leq \sum_{l=0}^{y-1} m_l + k \cdot d$ (note that $i \in N_y^w$ and $(w, i, k) \models \mathbf{F_p}\varphi$). As $\sum_{l=0}^{y-1} m_l$ is the smallest position in $f(i)$, then $\forall j \in f(i) : (w', j, d \cdot k) \models \mathbf{F_p}\varphi$.

Case 5: $\varphi = \varphi_1 \mathbf{U} \varphi_2$**.** $(w, i, k) \models \varphi_1 \mathbf{U} \varphi_2 \Leftrightarrow \exists j \geq i : (w, j, k) \models \varphi_2$ and $\forall e < j : (w, e, k) \models \varphi_1$. Then, by induction hypothesis we have: $\forall e < j$ $\forall l \in f(e) : (w', l, d \cdot k) \models \varphi_1$ and $\forall l \in f(j) : (w', l, d \cdot k) \models \varphi_2$, therefore $\forall j \in f(i) : (w', j, d \cdot k) \models \varphi_1 \mathbf{U} \varphi_2$

Case 6: $\varphi = \varphi_1 \mathbf{R} \varphi_2$**.** $(w, i, k) \models \varphi$ then either $\forall e \geq i\ (w, e, k) \models \varphi_2$ or $\exists e \geq i : (w, e, k) \models \varphi_1 \wedge \forall j \leq e\ (w, j, k) \models \varphi_2$

- **Subcase:** $\forall e \geq i\ (w, e, k) \models \varphi_2$. By induction hypothesis we have $\forall e \geq i$ $\forall j \in f(e) : (w', j, d \cdot k) \models \varphi_2$ then $\forall j \in f(i) : (w', j, d \cdot k) \models \varphi$
- **Subcase:** $\exists e \geq i : (w, e, k) \models \varphi_1 \wedge \forall j \leq e\ (w, j, k) \models \varphi_2$. Then, by induction hypothesis , we have: $\forall l \in f(e) : (w', l, d \cdot k) \models \varphi_1$ and $\forall j \leq e$ $\forall l \in f(e) : (w', l, d \cdot k) \models \varphi_2$, therefore $\forall j \in f(0) : (w', j, d \cdot k) \models \varphi$

□

Our later proofs will be based on the existence of counterexamples to a given property, and will use the following consequence of Theorem 11.

Corollary 1. *Let w, w' be d-stutter equivalent words, φ a Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$ formula, and f as defined above. Then $\forall k \in \mathbb{N}$:*

$$(w, i, k) \not\models \varphi \ \Rightarrow\ \forall j \in f(i) : (w', j, k/d) \not\models \varphi$$

4.1.3 Bounded-Fair Systems with Prompt-LTL Specifications

We consider systems that keep track of bounded fairness explicitly by running in parallel to $A\|B^n$ one counter for each process. In a step of the system where process p moves, the counter of p is reset, and all other counters are incremented. If one of the counters exceeds the bound b, the counter goes into a failure state from which no transition is enabled. We call such a system a *bounded-fair system*, and denote it $A\|_b B^n$.

A *path* of a bounded-fair system $A\|_b B^n$ is given as $x = (s_0, b_0)(s_1, b_1)\ldots$, and extends a path of $A\|B^n$ by valuations $b_i \in \{0, \ldots, b\}^{n+1}$ of the counters. Note that a run (i.e., a maximal path) of $A\|_b B^n$ is finite iff either it is deadlocked (in which case also its projection to a run of $A\|B^n$ is deadlocked) or a failure state is reached. Thus, the projection of all infinite runs of $A\|_b B^n$ to $A\|B^n$ are exactly the globally b-bounded fair runs of $A\|B^n$.

From now on, we write $x \in A\|B^n$ to denote that x is a run of $A\|B^n$.

Prompt-LTL$\backslash$X Specifications. Given a system $A\|B^n$, we consider specifications over $AP = Q_A \cup (Q_B \times \{1, \ldots, n\})$, i.e., states of processes are used as atomic propositions. For $i_1, \ldots, i_c \in \{1, \ldots, n\}$, we write $\varphi(A, B_{i_1}, \ldots, B_{i_c})$ for a formula that contains only atomic propositions from $Q_A \cup (Q_B \times \{i_1, \ldots, i_c\})$.

In the absence of fairness considerations, we say that $A\|B^n$ *satisfies* φ if

$$\exists k \in \mathbb{N}\ \forall x \in A\|B^n :\ (x, 0, k) \models \varphi.$$

We say that $A\|B^n$ *satisfies* $\varphi(A, B_1, \ldots, B_c)$ *under global bounded fairness*, written $A\|B^n \models_{gb} \varphi(A, B_1, \ldots, B_c)$, if

$$\forall b \in \mathbb{N}\ \exists k \in \mathbb{N}\ \forall x \in A\|B^n :\ b\text{-gfair}(x) \Rightarrow (x, 0, k) \models \varphi(A, B_1, \ldots, B_c).$$

Finally, for local bounded fairness we usually require bounded fairness for all processes that appear in the formula $\varphi(A, B_1, \ldots, B_c)$. Thus, we say that $A\|B^n$ *satisfies* $\varphi(A, B_1, \ldots, B_c)$ *under local bounded fairness*, written $A\|B^n \models_{lb} \varphi(A, B_1, \ldots, B_c)$, if

$$\forall b \in \mathbb{N}\ \exists k \in \mathbb{N}\ \forall x \in A\|B^n :\ b\text{-lfair}(x, \{1, \ldots, c\}) \Rightarrow (x, 0, k) \models \varphi(A, B_1, \ldots, B_c).$$

Parameterized Specifications. A *parameterized specification* is a Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formula with quantification over the indices of atomic propositions. A *h-indexed formula* is of the form $\forall i_1, \dots, \forall i_h.\varphi(A, B_{i_1}, \dots, B_{i_h})$. Let $f \in \{gb, lb\}$, then for given $n \geq h$,

$$A \| B^n \models_f \forall i_1, \dots, \forall i_h.\varphi(A, B_{i_1}, \dots, B_{i_h})$$

$$\Leftrightarrow$$

$$\forall j_1 \neq \dots \neq j_h \in \{1, \dots, n\} : \; A \| B^n \models_f \varphi(A, B_{j_1}, \dots, B_{j_h}).$$

By symmetry of guarded protocols, this is equivalent (cp. [43]) to $A \| B^n \models_f \varphi(A, B_1, \dots, B_h)$. The latter formula is denoted by $\varphi(A, B^{(h)})$, and we often use it instead of the original $\forall i_1, \dots, \forall i_h.\varphi(A, B_{i_1}, ..., B_{i_h})$.

Cutoffs. A *cutoff* for a given class of systems with processes from T, a fairness notion $f \in \{lb, gb\}$ and a set of Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formulas Φ is a number $c \in \mathbb{N}$ such that

$$\forall A, B \in T \; \forall \varphi \in \Phi \; \forall n \geq c : \; A \| B^n \models_f \varphi \; \Leftrightarrow \; A \| B^c \models_f \varphi.$$

Decidability. As mentioned before, the existence of a cutoff implies that the PMCP is decidable if the model checking problem for the cutoff system $A \| B^c$ is decidable. Decidability of model checking for finite transition systems with specifications in Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ and bounded fairness follows from the fact that bounded fairness can be expressed in Prompt-$\mathsf{LTL}\backslash\mathbf{X}$, and from results on decidability of assume-guarantee model checking for Prompt-LTL (cf. Kupferman et al. [76] and Faymonville and Zimmermann [51][Lemmas 8, 9]).

4.2 Cutoffs for Disjunctive Systems

In this section, we prove cutoff results for disjunctive systems under bounded fairness and stutter-insensitive specifications with or without promptness. To this end, in Section 4.2.1 we prove two lemmas that show how to simulate, up to bounded stuttering, local runs from a system of given size n in a smaller or larger disjunctive system. We then use these two lemmas in Subsections 4.2.2 and 4.2.3 to obtain cutoffs for specifications in $\mathsf{LTL}\backslash\mathbf{X}$ and Prompt-$\mathsf{LTL}\backslash\mathbf{X}$, respectively.

Moreover for the proofs of these two lemmas we utilize the same construction techniques that were used in Chapter 3 and in [9, 10, 43, 65], but in addition we analyze their effects on bounded fairness and bounded stutter equivalence. Note that we will only consider formulas of the form $\varphi(A, B^{(1)})$, however, similarly to the results in Chapter 3, our results extend to specifications over an arbitrary number h of B-processes.

Table 4.1 summarizes the results of this section: for specifications in $\mathsf{LTL}\backslash\mathbf{X}$ and Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ we obtain a cutoff that depends on the size of process template B, as well as on the number h of quantified index variables. The table states generalizations of Theorems 12 and 13 from the 2-indexed case to the h-indexed case for arbitrary $h \in \mathbb{N}$. For one of the cases we were not able to obtain a cutoff result (as explained in Section 4.2.3).

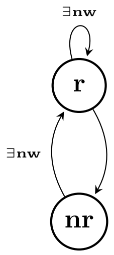


Figure 4.2: Reader

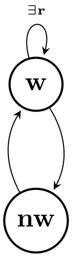


Figure 4.3: Writer

Simple Reader-Writer Example. Consider the disjunctive system $W\|R^n$, where W is a writer process (Figure 4.3), and R is a reader process (Figure 4.2). Let the specification φ be $\forall i\, \mathbf{G}(\mathbf{w} \rightarrow \mathbf{F_p}[(\mathbf{w} \wedge \mathbf{nr}_i)])$, i.e., if process W is in state $\mathbf{w}$, then eventually all the R processes will be in state $\mathbf{nr}$, while W is in $\mathbf{w}$. According to Table 4.1, the cutoff for checking whether $W\|R^n \models_{lb} \varphi$ is 5.

Table 4.1: Cutoffs for Disjunctive Systems

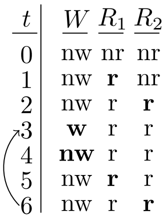
	Local Bounded Fairness	Global Bounded Fairness
h-indexed $\mathsf{LTL}\backslash\mathbf{X}$	$\lvert B\rvert + min(\lvert\mathcal{G}\rvert, \lvert B\rvert) + h$	$\lvert B\rvert + min(\lvert\mathcal{G}\rvert, \lvert B\rvert) + h$
h-indexed Prompt-$\mathsf{LTL}\backslash\mathbf{X}$	$\lvert B\rvert + min(\lvert\mathcal{G}\rvert, \lvert B\rvert) + h$	-

4.2.1 Simulation up to Bounded Stutter Equivalence

Our first lemma states that any behavior of processes A and B_1 in a system $A\|B^n$ can be simulated up to bounded stuttering in a system $A\|B^{n+1}$.

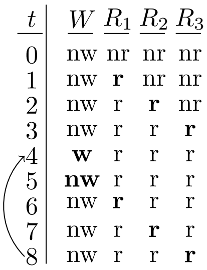
Lemma 7 (Monotonicity Lemma for Bounded Stutter Equivalence)**.** *Let A, B be disjunctive process templates, $n \geq 2$, $b \in \mathbb{N}$ and $x \in A\|B^n$ with b-lfair$(x, \{A, B_1\})$. Then there exists $y \in A\|B^{n+1}$ with $y = s'_0, s'_1, \ldots,$ $2b$-lfair$(y, \{A, B_1\})$ and $x(A, B_1) \equiv_2 y(A, B_1)$.*

Proof. Let x be a run of $A\|B^n$ where b-lfair$(x, \{A, B_1\})$. Let $y(A) = x(A)$ and $y(B_j) = x(B_j)$ for all $B_j \in \{B_1, \ldots, B_n\}$ and let the new process B_{n+1} copy one of the B-processes of $A\|B^n$, i.e., $y(B_{n+1}) = x(B_i)$ for some $i \in \{1, \ldots, n\}$. Copying a local run violates the interleaving semantics as two processes will be moving at the same time. To solve this problem, we split every transition (s'_l, s'_{l+1}) where the interleaving semantics is violated by B_i and B_{n+1} executing local transitions (q_i, g, q'_i) and (q_{n+1}, g, q'_{n+1}), respectively. To do this, replace (s'_l, s'_{l+1}) with two consecutive transitions $(s'_l, u)(u, s'_{l+1})$, where (s'_l, u) is based on the local transition (q_i, g, q'_i) and (u, s'_{l+1}) is based on the local transition (q_{n+1}, g, q'_{n+1}). Note that both of these local transitions are enabled in the constructed run y since the transition (q_i, g, q'_i) is enabled in the original run x. Moreover, run y inherits unconditional fairness from x. Finally, it is easy to see that for every local transition of process B_i in x, establishing interleaving semantics has added one additional stuttering step to every local run in y



t	W	R_1	R_2
0	nw	nr	nr
1	nw	**r**	nr
2	nw	r	**r**
3	**w**	r	r
4	**nw**	r	r
5	nw	**r**	r
6	nw	r	**r**

Figure 4.4: Run: $W||R^2$



t	W	R_1	R_2	R_3
0	nw	nr	nr	nr
1	nw	**r**	nr	nr
2	nw	r	**r**	nr
3	nw	r	r	**r**
4	**w**	r	r	r
5	**nw**	r	r	r
6	nw	**r**	r	r
7	nw	r	**r**	r
8	nw	r	r	**r**

Figure 4.5: Run: $W||R^3$

including processes A and B_1. Therefore we have that $2b$-lfair$(y, \{A, B_1\})$ and $x(A, B_1) \equiv_2 y(A, B_1)$. □

Reader-Writer Example. Consider the run x of the system $W||R^2$ in Figure 4.4 where W and R are as defined in Figures 4.2 and 4.3. We construct a run y of the system $W||R^3$ (see Figure 4.5) such that $x(W, R_1) \equiv_2 y(W, R_1)$. The local run of process R_3 is obtained by (i) copying the run of R_2, and (ii) establishing the interleaving semantics as in the proof of Lemma 7.

As mentioned in the above construction, if a local run of x is d-bounded fair for some $d \in \mathbb{N}$, then the constructed run y will be $2d$-bounded fair. This observation leads to the following corollary.

Corollary 2. *Let A, B be disjunctive process templates, $n \geq 2$, $b \in \mathbb{N}$ and $x \in A\|B^n$ with b-gfair(x). Then there exists $y \in A\|B^{n+1}$ with $2b$-gfair(y) and $x(A, B_1) \equiv_2 y(A, B_1)$.*

Our second lemma is a *bounding lemma* which states that any behavior of processes A and B_1 in a disjunctive system $A\|B^n$ can be simulated up to bounded stuttering in a system $A\|B^c$, if c is chosen to be sufficiently large and $n \geq c$.

Lemma 8 (Bounding Lemma for Bounded Stutter Equivalence)**.** *Let A, B be disjunctive process templates, $c = |B| + min(|\mathcal{G}|, |B|) + 1$, $n \geq c$, $b \in \mathbb{N}$ and $x \in A\|B^n$ with b-lfair$(x, \{A, B_1\})$. Then there exists $y \in A\|B^c$ with $(b \cdot c)$-lfair$(y, \{A, B_1\})$ and $x(A, B_1) \equiv_c y(A, B_1)$.*

Proof. Let x be a run of $A\|B^n$ where b-lfair$(x, \{A, B_1\})$. We show how to construct a run y of $A\|B^c$ where $(b \cdot c)$-lfair$(y, \{A, B_1\})$ and $x(A, B_1) \equiv_c y(A, B_1)$.

The basic idea is that, in order to ensure that all transitions in y are enabled at the time they are taken, we "flood" every state q that is visited in x with one or more processes that enter q and stay there. Additionally, we need to take care of fairness, which requires a more complicated construction that allows every such process to move infinitely often. Therefore, some processes have to leave the state they have flooded (if that state only appears finitely often in the original run), and every process needs to eventually enter a loop that allows it to move infinitely often. In the following, we construct such runs formally.

Construction:

0 $y(A) = x(A)$, and $y(B_1) = x(B_1)$.

1. **(Flooding with evacuation)**: To every $q \in \mathsf{Visited}^{fin}(x)$, devote one process B_{i_q} that copies B_{first_q} until the time f_q, then stutters in q until time l_q where it starts copying B_{last_q} forever. Formally:

$$y(B_{i_q}) = x(B_{\mathsf{first}_q})[0 : f_q].(q)^{l_q - f_q}.x(B_{last_q})[l_q + 1 : \infty]$$

2. **(Flooding with fair extension)**: For every $q \in \mathsf{Visited}^{inf}(x)$, let B_q^{inf} be a process that visits q infinitely often in x. We devote to q two processes $B_{i_{q_1}}$ and $B_{i_{q_2}}$ that both copy B_{first_q} until the time f_q, and then stutter in q until B_q^{inf} reaches q for the first time. After that, let $B_{i_{q_1}}$ and $B_{i_{q_2}}$ copy B_q^{inf} in turns as follows: $B_{i_{q_1}}$ copies B_q^{inf} until it reaches q while $B_{i_{q_2}}$ stutters in q, then $B_{i_{q_2}}$ copies B_q^{inf} until it reaches q while $B_{i_{q_1}}$ stutters in q and so on.

3. Establish interleaving semantics as in the proof of Lemma 7.

After steps 1 and 2, the following property holds: at any time t we have that $\mathsf{Set}(s_t(\mathcal{P} \setminus B_1)) \subseteq \mathsf{Set}(s'_t(\mathcal{P} \setminus B_1))$, which guarantees that every transition along the run is enabled. Note that establishing the interleaving semantics preserves this property.

This construction, introduced by Außerlechner et al. [10], gives a cutoff of $2|B|+1$, since in the worst case all states appear infinitely often which requires two copies for each. In Chapter 3, we gave a construction that results in a better cutoff based on an analysis of a given template. The construction mainly changes step 2 where from the states in $\mathsf{Visited}^{inf}(x)$ we can choose one representative state for each guard (the one that is visited first in x), and only add two local runs for each representative. This does not work for the local runs constructed for states in $\mathsf{Visited}^{fin}(x)$ since we need to move them into an infinitely visited state to guarantee fairness, and then we would need a different representative. This construction gives a cutoff of $|B| + |\mathcal{G}| + 1$.

Finally, establishing interleaving semantics could introduce additional stuttering steps to the local runs of processes A and B_1 whenever steps 1 or 2 of the construction use the same local run from x more than once (e.g. if $\exists q_i, q_j \in Q_B$ with $\mathsf{first}_{q_i} = \mathsf{first}_{q_j}$). A local run of x can be used in the above construction at most $c-1$ times, therefore we have $x(A, B_1) \equiv_c y(A, B_1)$. Moreover, since the upper bound of consecutive stuttering steps in A or B_1 is $c \cdot b$, we get $(b \cdot c)$-lfair$(y, \{A, B_1\})$. □

4.2.2 Cutoffs for Specifications in LTL\X under Bounded Fairness

The PMCP for disjunctive systems with specifications from $\mathsf{LTL}\backslash\mathbf{X}$ has been considered in several previous works [10, 43, 65]. In the following we extend these results by proving cutoff results under bounded fairness.

Theorem 12 (Cutoff for $\mathsf{LTL}\backslash\mathbf{X}$ with Global Bounded Fairness)**.** *Let A, B be disjunctive process templates, $c = |B| + min(|\mathcal{G}|, |B|) + 1$, $n \geq c$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in \mathsf{LTL}\backslash\mathbf{X}$. Then:*

$$\left(\forall b \in \mathbb{N} : \; A\|_b B^n \models \varphi(A, B^{(1)})\right) \Leftrightarrow \left(\forall b' \in \mathbb{N} : \; A\|_{b'} B^c \models \varphi(A, B^{(1)})\right)$$

We prove the theorem by proving two lemmas, one for each direction of the equivalence.

Lemma 9 (Monotonicity Lemma for $\mathsf{LTL}\backslash\mathbf{X}$)**.** *Let A, B be disjunctive process templates, $n \geq 1$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in \mathsf{LTL}\backslash\boldsymbol{X}$. Then:*

$$\left(\exists b \in \mathbb{N} : \; A\|_{b}B^n \not\models \varphi(A, B^{(1)})\right) \Rightarrow \left(\exists b' \in \mathbb{N} : \; A\|_{b'}B^{n+1} \not\models \varphi(A, B^{(1)})\right)$$

Proof. Assume $\exists b \in \mathbb{N} : \; A\|_{b}B^n \not\models \varphi(A, B^{(1)})$. Then there exists a run x of $A\|B^n$ where x is b-gfair(x) and $x \not\models \varphi(A, B^{(1)})$. According to Corollary 2 there exists y of $A\|B^{n+1}$ where $2b$-gfair(y) and $x(A, B_1) \equiv_2 y(A, B_1)$, which guarantees that $y \not\models \varphi(A, B^{(1)})$. □

For the corresponding bounding lemma, our construction is based on that of Lemma 8. However, the local runs resulting from that construction might stutter in some local states for an unbounded time (e.g. local runs devoted for states in $\mathsf{Visited}^{fin}(x)$). To bound stuttering in such constructions, given an arbitrary run of a system $A\|B^n$, we first show that whenever there exists a bounded-fair run that violates a specification in $\mathsf{LTL}\backslash\mathbf{X}$, then there also exists an ultimately periodic run with the same property. Such a run can be extracted from the product of the automaton that represents the property and the system.

Run Graph. A *run graph* of a Büchi automaton $\mathcal{A}_\varphi = (Q_A \times Q_B^n, Q_{\mathcal{A}_\varphi}, \delta, a_0, \alpha)$ on a system $A\|_{b}B^n$ is a directed graph $\mathcal{G}_b^n(\varphi) = (V, E)$ where:

- $V \subseteq (Q_A \times Q_B^n) \times \{0, \ldots, b\}^{n+1} \times Q_{\mathcal{A}_\varphi}$
- $(s_0, b_0, a_0) \in V$, where b_0 denotes that all counters are set to 0.
- $((s, b, a), (s', b', a')) \in E$ iff $(s, s') \in \Delta$, $a' \in \delta(a, s)$, and b' results from b according to the rules for the counters.

An infinite path of the run graph $\pi = (s_0, b_0, a_0)(s_1, b_1, a_1) \ldots$ is an *accepting path* if it starts with (s_0, b_0, a_0), and visits a state $a_\alpha \in \alpha$ infinitely often.

Lemma 10 (Ultimately Periodic Counter-Example)**.** *Let $\varphi \in \mathsf{LTL}$ and $b \in \mathbb{N}$. If $A\|_{b}B^n \not\models \varphi$ then there exists a run $x = uv^\omega$ of $A\|B^n$ with b-gfair(x), and $x \not\models \varphi$, where u, v are finite paths, and $|u|, |v| \leq 2 \cdot |A| \cdot |B|^n \cdot b^{n+1} \cdot |Q_{\mathcal{A}_{\neg\varphi}}|$.*

Proof. Assume that $A\|_{b}B^n \not\models \varphi$. Then there exists an accepting path π' in the run graph $\mathcal{G}_b^n(\neg\varphi)$. We first construct out of π' a fair path $\pi = u_\pi v_\pi^\omega$, by detecting and extracting a lasso-shaped accepting path from π'. In π' there exists an infix $\pi'_i \ldots \pi'_j$ where $\pi'_i = \pi'_j$, and there exists $\pi'_l \in \{\pi'_{i+1}, \ldots, \pi'_{j-1}\}$ with $\pi'_l(Q_{\mathcal{A}_{\neg\varphi}}) \in \alpha$ (accepting state in the automaton). Therefore, we have $\pi'_0 \ldots \pi'_{i-1}(\pi'_i \ldots \pi'_{j-1})^\omega$ is an accepting path of $\mathcal{G}_b^n(\neg\varphi)$.

Let $u' = \pi'_0 \ldots \pi'_{i-1}$ and $v' = \pi'_i \ldots \pi'_{j-1}$, then we can construct u_π and v_π by detection and removal of cycles under some conditions: (i) let u_π be a finite path obtained form u' where we iteratively replace every infix $\pi'_s \ldots \pi'_t$ with π'_s if $\pi'_s = \pi'_t$. Then, since u_π does not contain repetitions, we have $u_\pi \leq |A| \cdot |B|^n \cdot b^{n+1} \cdot |Q_{\mathcal{A}_{\neg\varphi}}|$. (ii) let $\pi'_a \in \{\pi'_i \ldots \pi'_{j-1}\}$ where $\pi'_a(\mathcal{A}_{\neg\varphi}) \in \alpha$ and let v_π be a finite path obtained form v' after we iteratively replace every infix $\pi'_s \ldots \pi'_t$ with π'_s if $\pi'_s = \pi'_t$ and $s \geq a$ or $t < a$. Thus, we get $v_\pi \leq 2 \cdot |A| \cdot |B|^n \cdot b^{n+1} \cdot |Q_{\mathcal{A}_{\neg\varphi}}|$.

Finally, let $x = u_\pi(Q_A \times Q_B^n)\,(v_\pi(Q_A \times Q_B^n))^\omega$. By construction, x is a run of $A\|B^n$ with b-gfair(x) and $x \not\models \varphi$. □

Now, we have all the ingredients to prove the bounding lemma for the case of $\mathsf{LTL}\backslash\mathbf{X}$ specifications and (global) bounded fairness.

Lemma 11 (Bounding Lemma for $\mathsf{LTL}\backslash\mathbf{X}$)**.** *Let A, B be disjunctive process templates, $c = |B| + min(|\mathcal{G}|, |B|) + 1$, $n \geq c$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in \mathsf{LTL}\backslash\mathbf{X}$. Then:*

$$\left(\exists b \in \mathbb{N} : \ A\|_b B^n \not\models \varphi(A, B^{(1)})\right) \Rightarrow \left(\exists b' \in \mathbb{N} : \ A\|_{b'} B^c \not\models \varphi(A, B^{(1)})\right)$$

Proof. Assume $\exists b \in \mathbb{N} : A\|_b B^n \not\models \varphi(A, B^{(1)})$. Then by Lemma 10 there is a run $x = uv^\omega$ of $A\|B^n$, where b-gfair(x) and $|u|, |v| \leq 2 \cdot |A| \cdot |B|^n \cdot b^{n+1} \cdot |Q_{\mathcal{A}_{\neg\varphi}}|$. According to Lemma 8, we can construct out of x a run y of $A\|B^c$ where b''-lfair$(y, \{A, B_1\})$, and $x(A, B_1) \equiv_c y(A, B_1)$ with $b'' = c \cdot b$. The latter guarantees that $y \not\models \varphi(A, B^{(1)})$. We still need to show that b'-gfair(y) for some $b' \in \mathbb{N}$. As $x = uv^\omega$, we observe that the construction of Lemma 8 ensures the following:

- The number of consecutive stuttering steps per process introduced in step 1 is bounded by $|u|$.

- The number of consecutive stuttering steps introduced in step 2 for a given process is bounded by $|u| + 2|v|$ because B_q^{inf} needs up to $|u| + |v|$ steps to reach q, and one of the processes has to wait for up to $|v|$ additional global steps before it can move.

In addition to the stuttering steps introduced in step 1 and 2, if more than one of the constructed runs simulate the same local run of x then establishing the interleaving semantics would be required, which in turn introduces additional stuttering steps. Therefore the upper bound of consecutive stuttering steps introduced in step 3 of the construction is $c \cdot b$. Therefore b'-gfair(y) where $b' = c \cdot b + 6 \cdot |A| \cdot |B|^n \cdot b^{n+1} \cdot |Q_{\mathcal{A}_{\neg\varphi}}|$. □

Remark 3. With a more complex construction that uses a stutter-insensitive automaton $\mathcal{A}$ [50] to represent the specification and considers runs of the composition of system and automaton, we can obtain a much smaller b' that is also independent of n. This is based on the observation that if in y some process is consecutively stuttering for more than $|A\|B^c \times \mathcal{A}|$ steps, then there must be a repetition of states from the product in this time, and we can simply cut the infix between the repeating states from the constructed run y.

4.2.3 Cutoffs for Specifications in Prompt-$\mathsf{LTL}\backslash$X

LTL specifications cannot enforce boundedness of the time that elapses before a liveness property is satisfied. Prompt-LTL solves this problem by introducing the prompt-eventually operator explained in Section 4.1.1. Since we consider concurrent asynchronous systems, the satisfaction of a Prompt-LTL formula can also depend on the scheduling of processes. If scheduling can introduce unbounded delays for a process, then promptness can in general not be guaranteed. Hence, non-trivial Prompt-LTL specifications can *only* be satisfied under the assumption of bounded fairness, and therefore this is the only case we consider here.

Theorem 13 (Cutoff for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ with Local Bounded Fairness)**.** *Let A, B be disjunctive process templates, $c = |B| + min(|\mathcal{G}|, |B|) + 1$ $n \geq c$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$. Then:*

$$A\|B^c \models_{lb} \varphi(A, B^{(1)}) \;\Leftrightarrow\; A\|B^n \models_{lb} \varphi(A, B^{(1)}).$$

Again, we prove the theorem by proving a monotonicity and a bounding lemma. Note that $A\|B^n \not\models_{lb} \varphi(A, B^{(1)})$ iff

$$\exists b \in \mathbb{N}\; \forall k \in \mathbb{N}\; \exists x \in A\|B^n\colon b\text{-lfair}(x, \{A, B^{(1)}\}) \wedge (x, 0, k) \not\models \varphi(A, B^{(1)}).$$

Lemma 12 (Monotonicity Lemma for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$)**.** *Let A, B be disjunctive process templates, $n \geq 2$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$. Then:*

$$A\|B^n \not\models_{lb} \varphi(A, B^{(1)}) \;\Rightarrow\; A\|B^{n+1} \not\models_{lb} \varphi(A, B^{(1)}).$$

Proof. Assume $A\|B^n \not\models_{lb} \varphi(A, B^{(1)})$. Then there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run x of $A\|B^n$ where b-lfair$(x, \{A, B^{(1)}\})$, and $(x, 0, 2 \cdot k) \not\models \varphi(A, B^{(1)})$. Then according to Lemma 7 there exists y of $A\|B^{n+1}$ where $2b$-lfair$(y, \{A, B^{(1)}\})$ and $x(A, B_1) \equiv_2 y(A, B_1)$, which guarantees, according to Corollary 1, that $(y, 0, k) \not\models \varphi(A, B^{(1)})$. As a consequence there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run y of $A\|B^c$ where $2b$-lfair$(y, \{A, B^{(1)}\})$ and $(y, 0, k) \not\models \varphi(A, B^{(1)})$, thus $A\|B^c \not\models_{lb} \varphi(A, B^{(1)})$.

□

Using the same argument of the above proof but by using Corollary 2 instead of Lemma 7 to construct the globally bounded fair counter example, we obtain the following:

Corollary 3. *Let A, B be disjunctive process templates, $n \geq 2$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$. Then:*

$$A\|B^n \not\models_{gb} \varphi(A, B^{(1)}) \;\Rightarrow\; A\|B^{n+1} \not\models_{gb} \varphi(A, B^{(1)}).$$

Lemma 13 (Bounding Lemma for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$)**.** *Let A, B be disjunctive process templates, $c = |B| + min(|\mathcal{G}|, |B|) + 1$, $n \geq c$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$. Then:*

$$A\|B^n \not\models_{lb} \varphi(A, B^{(1)}) \;\Rightarrow\; A\|B^c \not\models_{lb} \varphi(A, B^{(1)}).$$

Proof. Assume $A\|B^n \not\models_{lb} \varphi(A, B^{(1)})$. Then there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run x of $A\|B^n$ where b-lfair$(x, \{A, B^{(1)}\})$ and $(x, 0, c \cdot k) \not\models \varphi(A, B^{(1)})$. According to Lemma 8 we can construct for every such x a run y of $A\|B^c$ where $(c \cdot b)$-lfair$(y, \{A, B^{(1)}\})$, and $x(A, B_1) \equiv_c y(A, B_1)$, which guarantees that $(y, 0, k) \not\models \varphi(A, B^{(1)})$ (see Corollary 1). Thus, there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run y of $A\|B^c$ where $(c \cdot b)$-lfair$(y, \{A, B^{(1)}\})$ and $(y, 0, k) \not\models \varphi(A, B^{(1)})$, thus $A\|B^c \not\models_{lb} \varphi(A, B^{(1)})$. □

The Absence of a Bounding Lemma under Global Bounded Fairness.

The reader will notice that we have no bounding lemma, and therefore no cutoff result for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ under global bounded fairness. The main reason is that the constructions we adopt do not allow us to determine a bound on the number of stuttering steps they generate. For instance, the proof of Lemma 11 depends on a bound on the time after which only infinitely visited states occur. Based on the existence of an ultimately periodic counterexample uv^ω, we can conclude that $|u|$ is sufficient as a bound. In case of Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ however, this technique is not sufficient: a Prompt-LTL counterexample consists of a fairness bound b such that for all k there is a non-satisfying run. Since the previously mentioned technique only produces a bound b that will depend on the run for a given k, it cannot solve our problem.

We show below how we can extract, from a run x that does not satisfy a Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formula φ, a lasso that does not satisfy φ either. Such extraction is not as simple as in the $\mathsf{LTL}\backslash\mathbf{X}$ case and therefore we first summarize the alternating color technique introduced by Kupferman et al. [76].

Alternating-Color Technique

A standard approach for model checking a system S against an LTL formula φ is to translate the negation of φ to a non-deterministic Büchi automaton $\mathcal{A}_{\overline{\varphi}}$, and then construct the Büchi graph $G = S \times \mathcal{A}_{\overline{\varphi}}$, the synchronous product of S and $\mathcal{A}_{\overline{\varphi}}$. An accepting initial path π in G witnesses a run of S that violates φ. For model checking a Prompt-LTL specification φ, Kupferman et al. [76] have shown an approach that encodes φ into a relativization $c(\varphi)$ in LTL with an additional atomic proposition (called a *color*), and uses a *colored* Büchi graph to decide the model checking problem.

Given a word $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$ and a proposition $r \notin \mathsf{AP}$, an *r-coloring* of w is a word $w^c = w_0^c w_1^c \ldots \in (2^{AP \cup \{r\}})^\omega$ where $\forall i \geq 0 \; w_i^c \cap AP = w_i$. We refer to the assignment to r as the color of location i and we call i a *color change point* if w_{i-1}^c and w_i^c have different colors. A subword $w_i^c \ldots w_j^c$ of w^c is an *r-block* if all the locations in the subword agree on r, and i and $j+1$ are color-change points. For $k \geq 0$, we say that w^c is *k-spaced*, *k-bounded*, and *k-tight* if w^c has infinitely many r-blocks, and all the blocks are of size at least k, at most k, and exactly k, respectively. Moreover let $alt_r = \mathbf{GF}r \wedge \mathbf{GF}\neg r$, and let $rel_r(\varphi)$ be the formula obtained from φ by replacing every instance of $\mathbf{F_p}\psi$ by $(r \rightarrow (r\mathbf{U}(\neg r\mathbf{U}\psi))) \wedge (\neg r \rightarrow (\neg r\mathbf{U}(r\mathbf{U}\psi)))$. We define $c(\varphi) = alt_r \wedge rel_r(\varphi)$ and $\bar{c}(\varphi) = alt_r \wedge \neg rel_r(\varphi)$ which are LTL formulas. Hence, a run of the system satisfies $c(\varphi)$ if it is partitioned into infinitely many blocks, and each prompt eventuality is satisfied within two blocks.

Lemma 14 (Promptness lemma [76]). *Given a Prompt-LTL formula φ, a word w, and a bound $k \geq 0$.*

1. *if $(w, 0, k) \models \varphi$ then for every k-spaced r-coloring w^c of w, we have $(w^c, 0) \models c(\varphi)$*

2. *if w^c is a k-bounded r-coloring of w such that $(w^c, 0) \models c(\varphi)$, then $(w, 0, 2k) \models \varphi$*

Colored Büchi Graph. A *colored Büchi graph* is a tuple $\mathcal{G} = (r, V, E, v_0, L, \phi)$, where r is a proposition, V is a set of vertices, E is a set of edges, $v_0 \in V$ is an initial vertex, $L : V \rightarrow \{r, \emptyset\}$ is the coloring function, and $\phi \subseteq V$ is a Büchi acceptance condition. A *path* of $\mathcal{G}$ is a sequence $\pi = (v_0, L(v_0))(v_1, L(v_1)) \ldots$, where $(v_i, v_{i+1}) \in E$ for all i. We say that a path π of $\mathcal{G}$ is *fair* if it visits ϕ infinitely often.

Given a Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formula φ, we define the product $A\|_{b}B^n \times \mathcal{A}_{\bar{c}(\varphi)}$ as the colored Büchi graph

$$\mathcal{P}_b^n(\varphi) = \left(r, (Q_A \times Q_B^n) \times [b]^{n+1} \times \{\{r\}, \emptyset\} \times Q_{\mathcal{A}_{\bar{c}(\varphi)}}, M, (s_0, b_0, r, a_0), L, \mathcal{F}\right),$$

where

- b_0 denotes that all counters are set to 0,
- $((s, b_i, r_i, a), (s', b_j, r_j, a')) \in M$ iff $(s, s') \in \Delta$, $a' \in \delta(a, s \cup r_i)$, and b_j results from b_i according to the rules for the counters,
- $L((s, b, r_i, a)) = r_i$, and
- $\mathcal{F} = (Q_A \times Q_B^n) \times [b]^{n+1} \times \{r, \emptyset\} \times \alpha$.

Given a path π of $\mathcal{P}_b^n(\varphi)$ and a process p we denote by $\pi(p)$ the local run of process p in π. If φ is a formula over a fixed number of processes (as is always the case in our specifications), say A and B_1, then $\delta(a_i, (s_i, r_i)) = \delta(a_i, (s_j, r_i))$ whenever $s_i(A, B_1) = s_j(A, B_1)$, i.e., δ depends only on the local states of processes that appear in φ.

Unlike the LTL case, a path π in the product $\mathcal{P}_b^n(\varphi)$ is not a witness that $A\|B^n$ does not satisfy φ, however π entails that only for some bound $k \in \mathbb{N}$ the formula φ is not satisfied in $A\|_{b}B^n$ with bound k. However from Theorem 4.2 in [76] and its proof, we obtain the following corollary:

Corollary 4. *Given a Prompt-LTL formula φ, and a system $A\|B^n$ we have:*

$$A\|B^n \not\models_{gb} \varphi \Leftrightarrow \exists b \in \mathbb{N}\ \exists x \in A\|B^n \text{ s.t. } b\text{-}gfair(x) \wedge (x, 0, k) \not\models \varphi$$

with $k \geq |\mathcal{P}_b^n(\varphi)|$

Lemma 15 (Ultimately Periodic Counter-Example)**.** *Given a Prompt-LTL formula φ, a run x of $A\|B^n$ with b-gfair(x) , and a bound $k \in \mathbb{N}$. if $(x, 0, 2k) \not\models \varphi$ then there exists a run x' of $A\|B^n$ where b-gfair(x'), $x' = uv^\omega$, $(x', 0, k) \not\models \varphi$, $|u| \leq k \cdot |\mathcal{P}_b^n(\varphi)|$, and $|v| \leq k \cdot |\mathcal{P}_b^n(\varphi)|$.*

Proof. Assume $(x, 0, 2k) \not\models \varphi$ then according to Lemma 14 every k-bounded r-coloring x^c of x we have $(x^c, 0) \not\models c(\varphi)$. Let x_1^c be a k-tight (therefore k-spaced) r-coloring of x, and let π_1 be the fair path of $\mathcal{P}_b^n(\varphi)$ that corresponds to x_1^c. We can construct out of π_1 a k-spaced fair path π_2 of the product $\mathcal{P}_b^n(\varphi)$ where $\pi_2 = u'v'^\omega$, $|u'| \leq k \cdot |\mathcal{P}_b^n(\varphi)|$, and $|v'| \leq |\mathcal{P}_b^n(\varphi)|$ by detecting and extracting a lasso shape fair path from π_1. In π_1 there exists an infix $v_i \dots v_j$ where i and j are color change points, $v_i = v_j$, and there exists $v_l \in \{v_{i+1}, \dots, v_{j-1}\}$ with $v_l(\mathcal{A}_{\bar{c}(\varphi)}) \in \alpha$ (accepting state in the automaton). Therefore $v_0 \dots v_{i-1}(v_i \dots v_{j-1})^\omega$ is a fair path of the product $\mathcal{P}_b^n(\varphi)$. Let $u_\pi = v_0 \dots v_{i-1}$ and $v_\pi = v_i \dots v_{j-1}$, then we can construct u' and v' by detecting and removing cycles under some conditions: let u' be a prefix obtained form u_π after replacing every infix $v_s \dots v_t$ with v_s iff $v_s = v_t$ and s and t are color change points. Let $v_a \in \{v_i \dots v_{j-1}\}$ where $v_a(\mathcal{A}_{\bar{c}(\varphi)}) \in \alpha$ and $\forall g \in \mathbb{N}$ s.t. $i \leq g < a$ we have $v_g(\mathcal{A}_{\bar{c}(\varphi)}) \notin \alpha$. Let v' be the suffix obtained form v_π after replacing every infix $v_s \dots v_t$ with v_s iff $s \geq g$,

$v_s = v_t$, s and t are color change points, and for all $v_l \in \{v_{s+1}, \dots, v_{t-1}\}$ we have $v_l(\mathcal{A}_{\bar{c}(\varphi)}) \notin \alpha$. Let $u^c = u'(A\|B^n \times \{r, \emptyset\})$ and let $v^c = v'(A\|B^n \times \{r, \emptyset\})$ then $x'^c = u^c(v^c)^\omega$ is k-spaced and $x'^c \not\models c(\varphi)$. Let $x' = x'^c(A\|B^n)$, then x' is a run of $A\|B^n$ and according to Lemma 14 we have $(x', 0, k) \not\models \varphi$. Moreover we have b-gfair(x') as it was obtained from a product fair path. □

As an alternative approach to the ultimately periodic counterexample, we tried a technique based on the algorithm for solving the model checking problem for Prompt-LTL by Kupferman et al. [76]. Their method is based on the detection of a *pumpable path* in the product of a system S and a specification automaton $\mathcal{A}_\varphi$. However, when constructing a pumpable path for $A\|B^c$ out of a pumpable path of $A\|B^n$, we run into the problem that in certain cases the value of c depends on n, and therefore no cutoff can be detected with this technique.

4.3 Cutoffs for Conjunctive Systems

In this section we investigate cutoff results for conjunctive systems under bounded fairness and specifications in Prompt-LTL$\backslash\mathbf{X}$. Table 4.2 summarizes the results of this section, as generalizations of Theorems 14 and 15 to h-indexed specifications. Note that for results marked with a $*$ we require processes to be *bounded initializing*, i.e., that every cycle in the process template contains the initial state.[1]

Table 4.2: Cutoffs for Conjunctive Systems

	Local Bounded Fairness	Global Bounded Fairness
h-indexed LTL$\backslash\mathbf{X}$	$h+1$	$h+1^*$
h-indexed Prompt-LTL$\backslash\mathbf{X}$	$h+1$	$h+1^*$

4.3.1 Cutoffs under Local Bounded Fairness

Theorem 14 (Cutoff for Prompt-LTL$\backslash\mathbf{X}$ with Local Bounded Fairness)**.** *Let A, B be conjunctive process templates, $n \geq 2$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-LTL$\backslash\boldsymbol{X}$. Then:*

$$A\|B^2 \models_{lb} \varphi(A, B^{(1)}) \;\Leftrightarrow\; A\|B^n \models_{lb} \varphi(A, B^{(1)}).$$

We prove the theorem by proving two lemmas, one for each direction of the equivalence. Note that $A\|B^n \not\models_{lb} \varphi(A, B^{(1)})$ iff $\exists b \in \mathbb{N}\ \forall k \in \mathbb{N}\ \exists x \in A\|B^n :\ b\text{-gfair}(x) \wedge (x, 0, k) \not\models \varphi(A, B^{(1)})$.

Lemma 16 (Monotonicity Lemma, Prompt-LTL$\backslash\mathbf{X}$ with Local Bounded Fairness)**.** *Let A, B be conjunctive process templates, $n \geq 2$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-LTL$\backslash\boldsymbol{X}$. Then:*

$$A\|B^n \not\models_{lb} \varphi(A, B^{(1)}) \;\Rightarrow\; A\|B^{n+1} \not\models_{lb} \varphi(A, B^{(1)}).$$

[1] This is only slightly more restrictive than the assumption that they are initializing, as stated in the definition of conjunctive systems in Section 2.1.

Proof. Assume $A \| B^n \not\models_{lb} \varphi(A, B^{(1)})$. Then there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run x of $A \| B^n$ where b-gfair(x) and $(x, 0, k) \not\models \varphi(A, B^{(1)})$. For every such x, we construct a run y of $A \| B^{n+1}$ with b-lfair(y) and $(y, 0, k) \not\models \varphi(A, B^{(1)})$. Let $y(A) = x(A)$ and $y(B_j) = x(B_j)$ for all $B_j \in \{B_1, \dots, B_n\}$ and let the new process B_{n+1} "share" a local run $x(B_i)$ with an existing process B_i of $A \| B^{n+1}$ in the following way: one process stutters in $init_B$ while the other makes transitions from $x(B_i)$, and whenever $x(B_i)$ enters $init_B$ the roles are reversed. Since this changes the behavior of B_i, B_i cannot be a process that is mentioned in the formula, i.e. we need $n \geq 2$ for a formula $\varphi(A, B^{(1)})$. Then we have b-lfair$(y, \{A, B_1\})$ as the run of B_{n+1} inherits the unconditional fairness behavior from the local run of the process B_i in x. Note that it is not guaranteed that the local runs $y(B_i)$ and $y(B_{n+1})$ are bounded fair as the time between two occurrences of $init_B$ in $x(B_i)$ is not bounded. Moreover we have $x(A, B_1) \equiv_1 y(A, B_1)$, which according to Corollary 1 implies $(y(A, B_1), k) \not\models \varphi(A, B^{(1)})$. □

Lemma 17 (Bounding Lemma, Prompt-LTL\X, Local Bounded Fairness)**.** *Let A, B be conjunctive process templates, $n \geq 1$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-LTL\\$\boldsymbol{X}$. Then:*

$$A \| B^n \not\models_{lb} \varphi(A, B^{(1)}) \;\Rightarrow\; A \| B^1 \not\models_{lb} \varphi(A, B^{(1)}).$$

Proof. Assume $A \| B^n \not\models_{lb} \varphi(A, B^{(1)})$. Then there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run x of $A \| B^n$ where b-gfair(x), and $(x, 0, b \cdot k) \not\models \varphi(A, B^{(1)})$. For every such x, we construct a run y in the cutoff system $A \| B^1$ by copying the local runs of processes A and B_1 in x and deleting stuttering steps. It is easy to see that b-gfair(y) then we have $x(A, B_1) \equiv_b y(A, B_1)$, and by Corollary 1 $(y(A, B_1), k) \not\models \varphi(A, B^{(1)})$. □

Note that this is the same proof construction as in Außerlechner et al. [10], and we simply observe that this construction preserves bounded fairness.

4.3.2 Cutoffs under Global Bounded Fairness

As mentioned before, to obtain a result that preserves global bounded fairness, we need to restrict process template B to be bounded initializing.

Theorem 15 (Cutoff for Prompt-LTL\X with Global Bounded Fairness)**.** *Let A, B be conjunctive process templates, where B is bounded initializing, $n \geq 2$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-LTL\\$\boldsymbol{X}$. Then:*

$$A \| B^2 \models_{gb} \varphi(A, B^{(1)}) \;\Leftrightarrow\; A \| B^n \models_{gb} \varphi(A, B^{(1)}).$$

Again, the theorem can be separated into two lemmas.

Lemma 18 (Monotonicity Lemma, Prompt-LTL\X, Global Bounded Fairness)**.** *Let A, B be conjunctive process templates, where B is bounded initializing, $n \geq 2$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-LTL\\$\boldsymbol{X}$. Then:*

$$A \| B^n \not\models_{gb} \varphi(A, B^{(1)}) \;\Rightarrow\; A \| B^{n+1} \not\models_{gb} \varphi(A, B^{(1)}).$$

Proof. Assume $A\|B^n \not\models_{gb} \varphi(A, B^{(1)})$. Then there exists $b \in \mathbb{N}$ such that $\forall k \in \mathbb{N}$ there is a run x of $A\|B^n$ where b-gfair(x), and $(x, 0, (b + |B|) \cdot k) \not\models \varphi(A, B^{(1)})$. For every such x, we construct a run y of $A\|B^{n+1}$ in the same way we did in the proof of Lemma 16. Then we have b'-gfair(y) with $b' = b + |B|$ as $init_B$ is on every cycle of the process template B. Moreover we have $x(A, B_1) \equiv_1 y(A, B_1)$ which according to Corollary 1 implies that $(y(A, B_1), k) \not\models \varphi(A, B^{(1)})$. □

Lemma 19 (Bounding Lemma, Prompt-$\mathsf{LTL}\backslash\mathbf{X}$, Global Bounded Fairness)**.** *Let A, B be conjunctive process templates, where B is bounded initializing, $n \geq 1$, and $\varphi(A, B^{(1)})$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\mathbf{X}$. Then:*

$$A\|B^n \not\models_{gb} \varphi(A, B^{(1)}) \;\Rightarrow\; A\|B^1 \not\models_{gb} \varphi(A, B^{(1)}).$$

Proof. Under the given assumptions, we can observe that the construction from Lemma 17 also preserves global bounded fairness. □

4.4 Token Passing Systems

In this section, we first introduce a system model for token passing systems and then show how to obtain cutoff results for this class of systems.

4.4.1 System Model

Processes. A *token passing process* is a transition system $T = (Q_T, I_T, \Sigma_T, \delta)$ where

- $Q_T = \overline{Q_T} \times \{0, 1\}$ is a finite set of states. $\overline{Q_T}$ is a finite non-empty set. The Boolean component $\{0, 1\}$ indicates the possession of the token.
- I_T is the set of initial states with $I_T \cap (\overline{Q_T} \times \{0\}) \neq \emptyset$ and $I_T \cap (\overline{Q_T} \times \{1\}) \neq \emptyset$.
- $\Sigma_T = \{\epsilon, rcv, snd\}$ is the set of actions, where ϵ is an asynchronous action, and $\{rcv, snd\}$ are the actions to receive and send the token.
- $\delta_T = Q_T \times \Sigma_T \times Q_T$ is a transition relation, such that $((q, b), a, (q', b')) \in \delta_T$ if all of the following hold:
  - $a = \epsilon \;\Rightarrow\; b = b'$.
  - $a = snd \;\Rightarrow\; b = 1$ and $b' = 0$
  - $a = rcv \;\Rightarrow\; b = 0$ and $b' = 1$

Token Passing System. Let $G = (V, E)$ be a finite directed graph without self loops where $V = \{1, \ldots, n\}$ is the set of vertices, and $E \subseteq V \times V$ is the set of edges. Given a token passing process T, the *token passing system* T^n_G is a concurrent system containing n instances of process T where the only synchronization between the processes is the sending/receiving of a token according to the graph G. Formally, $T^n_G = (S, init_S, \Delta)$ with:

- $S = (Q_T)^n$.
- $init_S = \{s \in (I_T)^n$ such that exactly one process holds the token $\}$,

- $\Delta \subseteq S \times S$ such that $((q_1, \ldots, q_n), (q'_1, \ldots, q'_n)) \in \Delta$ iff:
  - **Asynchronous Transition**. $\exists i \in V$ such that $(q_i, \epsilon, q'_i) \in \delta_T$, and $\forall j \neq i$ we have $q_j = q'_j$.
  - **Synchronous Transition**. $\exists (i,j) \in E$ such that $(q_i, snd, q'_i) \in \delta_T$, $(q_j, rcv, q'_j) \in \delta_T$, and $\forall z \in V \setminus \{i,j\}$ we have $q_z = q'_z$.

Runs. A *configuration* of a system T^n_G is a tuple (s, ac) where $s \in S$, and either $ac = a_i$ with $a \in \Sigma_{\scriptscriptstyle T}$, and $i \in V$ is a process index, or $ac = (snd_i, rcv_j)$ where $i,j \in V$ are two process indices with $i \neq j$. A run is an infinite sequence of configurations $x = (s_0, ac_0)(s_1, ac_1)\ldots$ where $s_0 \in init_S$ and s_{i+1} results from executing action ac_i in s_i. Additionally we denote by $x(T_i, \ldots, T_j)$ the projection $(s_0(T_i, \ldots, T_j), ac_0(T_i, \ldots, T_j))(s_1(T_i, \ldots, T_j), ac_1(T_i, \ldots, T_j))\ldots$ where $s_e(T_i, \ldots, T_j)$ is the projection of s_e on the local states of $(T_i, \ldots, T_j)$ and

$$ac(T_i, \ldots, T_j) = \begin{cases} \perp & \text{if } ac = a_m \text{ and } m \notin \{i, \ldots, j\} \\ \perp & \text{if } ac = (snd_m, rcv_n) \text{ and } m,n \notin \{i, \ldots, j\} \\ ac & otherwise \end{cases}$$

Bounded Fairness. A run x of a token passing system T^n_G is b-bounded globally fair, denoted b-gfair(x) if for every moment m and every process T_i, T_i receives the token at least once between moments m and $m + b$.

Cutoffs for Complex Networks. In the presence of different network topologies, represented by the graph G, we define a cutoff to be a bound on the size of G that is sufficient to decide the PMCP. Note that, in order to obtain a decision procedure for the PMCP, we not only need to know the size of the graphs, but also which graphs of this size we need to investigate. This is straightforward if the graph always falls into a simple class, such as rings, cliques, or stars, but is more challenging if the graph can become more complex with increasing size.

4.4.2 Cutoff Results for Token Passing Systems

Table 4.3 summarizes the results of this section, generalizing Theorem 16 to the case of h-indexed specifications. Similar to previous sections, the specifications are over states of processes. The results for local bounded fairness follow from the results for global bounded fairness.

To prove the results of this section, we need some additional definitions.

Table 4.3: Cutoff Results for Token Passing Systems

	Local Bounded Fairness	Global Bounded Fairness
h-indexed $\mathsf{LTL}\backslash\mathbf{X}$	$2h$	$2h$
h-indexed Prompt-$\mathsf{LTL}\backslash\mathbf{X}$	$2h$	$2h$

Connectivity vector [31]. Given two indices $i, j \in V$ in a finite directed graph G, we define the connectivity vector $v(G, i, j) = (u_1, u_2, u_3, u_4, u_5, u_6)$ as follows:

- $u_1 = 1$ if there is a non-empty path from i to i that does not contain j. $u_1 = 0$ otherwise.

- $u_2 = 1$ if there is a path from i to j via vertices different from i and j. $u_2 = 0$ otherwise.
- $u_3 = 1$ if there is a direct edge from i to j. $u_3 = 0$ otherwise.
- u_4, u_5, u_6 are defined like u_1, u_2, u_3, respectively where i is replaced by j and vice versa.

Immediately Sends. Given a token passing process T, we fix two local states q^{snd} and q^{rcv}, such that there is (i) a local path $q^{\mathsf{init}}, \ldots, q^{rcv}$ where $q^{\mathsf{init}} \in I_T \cap (\overline{Q_T} \times \{0\})$, (ii) a local path $q^{rcv}, \ldots, q^{snd}$ that starts with a receive action, and (iii) a local path $q^{snd}, \ldots, q^{rcv}$ that starts with a send action. We assumed here that such a path exists as otherwise all the runs of the system will be finite.

When constructing a local run for a process T_i that is currently in local state q^{rcv}, we say that T_i *immediately sends the token* if and only if:

1. T_i executes consecutively all the actions on a simple path $q^{rcv}, \ldots, q^{snd}$, then sends the token, and then executes consecutively all the actions on a simple path $q^{snd}, \ldots, q^{rcv}$.
2. All other processes remain idle until T_i reaches q^{rcv} (obviously, except those processes that need to synchronize with q^{snd} and q^{rcv}).

Note that, when T_i *immediately* sends the token, it executes at most $|T|$ actions, since the two paths cannot share any states except q^{rcv} and q^{snd}.

Theorem 16 (Cutoff for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$). *Let T^n_G be a token-passing system, $g, h \in V$, and $\varphi(T_g, T_h)$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$. Then there exists a system $T^4_{G'}$ with $G' = (V', E')$ and $i, j \in V'$ such that $v(G, g, h) = v(G', i, j)$, $|V'| = 4$, and*

$$T^n_G \models_{gb} \varphi(T_g, T_h) \Leftrightarrow T^4_{G'} \models_{gb} \varphi(T_i, T_j).$$

We prove the theorem by proving two lemmas, one for each direction of the equivalence. Note that $T^n_G \not\models_{gb} \varphi(T_g, T_h)$ iff $\exists b \in \mathbb{N}\, \forall k \in \mathbb{N}\, \exists x \in T^n_G :$ b-gfair$(x) \wedge$ $(x, 0, k) \not\models \varphi(T_g, T_h)$.

Lemma 20 (Monotonicity Lemma). *Let T^n_G be a token-passing system with $n \geq 3$ and $g, h \in V$, and $\varphi(T_g, T_h)$ a specification with $\varphi \in$ Prompt-$\mathsf{LTL}\backslash\boldsymbol{X}$. Then there exists a system $T^{n+1}_{G'}$ with $G' = (V', E')$ and $i, j \in V'$ such that $v(G, g, h) = v(G', i, j)$ and*

$$T^n_G \not\models_{gb} \varphi(T_g, T_h) \;\Rightarrow\; T^{n+1}_{G'} \not\models_{gb} \varphi(T_i, T_j).$$

Proof. Let a be a vertex of G with $a \notin \{g, h\}$. Then we construct G' from G as follows: Let $V' = V \cup \{n+1\}$, and $E' = (E \cup \{(n+1, m) | (a, m) \in E$ for some $m \in V\} \cup \{(a, n+1)\}) \setminus \{(a, m) | (a, m) \in E$ for some $m \in V\}$, i.e. we copy all the outgoing edges of a to the vertex $n+1$, and replace all the outgoing edges of a by one outgoing edge to $n+1$.

Assume $T^n_G \not\models_{gb} \varphi(T_g, T_h)$. Then there exists $b \in \mathbb{N}$ such that $\forall k' \in \mathbb{N}$ there is a run x of T^n_G where b-gfair(x), and $(x, 0, |T| \cdot k') \not\models \varphi(T_g, T_h)$. Let $b' = b + (b - n + 2) \cdot |T|$, and $d = |T| + 1$. We will construct for every such run x

a run y of $T_{G'}^{n+1}$ where b'-gfair(y), and $x(T_g, T_h) \equiv_d y(T_i, T_j)$ which guarantees that $(y, 0, k') \not\models \varphi(T_i, T_j)$ (see Corollary 1).

Construction. The construction is such that we keep the local paths of the n existing processes up to bounded stuttering, and we add a process T_{n+1} that always immediately sends the token after receiving it, with q^{rcv}, q^{snd} and the corresponding paths as defined above. In the following, as a short-hand notation, if $s = (q_1, \ldots, q_n)$ is a global state of T_G^n and $q \in Q_T$, we write (s, q) for $(q_1, \ldots, q_n, q)$.

Let $x = (s_0, ac_0)(s_1, ac_1)\ldots$ and $y' = ((s_0, q^{rcv}), ac_0)((s_1, q^{rcv}), ac_1)\ldots$. Note that y' is a sequence of configurations of $T_{G'}^{n+1}$, but not a run. To obtain a run, first let

$$y'' = ((s_0, q^{\mathsf{init}}), \epsilon)\ldots((s_0, q^{rcv}), ac_0)((s_1, q^{rcv}), ac_1)\ldots$$

Finally, replace every occurrence of a pair of consecutive configurations $((s, q^{rcv}), (snd_a, rcv_z))$, $((s', q^{rcv}), ac')$, where $s, s' \in Q_T^n, z \in V, ac' \in \Sigma$, with the sequence

$$((s, q^{rcv}), (snd_a, rcv_{n+1}))\ldots((s, q^{snd}), (snd_{n+1}, rcv_z))\ldots((s', q^{rcv}), ac').$$

In other words, instead of sending the token to T_z, T_a sends the token to T_{n+1}, and T_{n+1} sends the token immediately to T_z. Furthermore, in x between moments t and $t+b$, T_a can send the token at most $b-n+1$ times, and whenever T_{n+1} receives the token, it takes at most $|T|$ steps before reaching q^{rcv} again. Finally, note that the number of steps T_{n+1} takes to reach q^{rcv} for the first time is also bounded by $|T|$. Therefore we have b'-gfair(y) and $x(T_g, T_h) \equiv_d y(T_i, T_j)$ (as $b' \leq b \cdot d$) which by Corollary 1 implies that $(y, 0, k') \not\models \varphi(T_i, T_j)$. □

Lemma 21 (Bounding Lemma). *Let T_G^n be a system with $n \geq 4$ and $g, h \in V$, and $\varphi(T_g, T_h)$ a specification with $\varphi \in$ Prompt-LTL$\backslash\boldsymbol{X}$. Then there exists a system $T_{G'}^4$ with $G' = (V', E')$ and $i, j \in V'$ such that $v(G, g, h) = v(G', i, j)$ and*

$$T_G^n \not\models_{gb} \varphi(T_g, T_h) \;\Rightarrow\; T_{G'}^4 \not\models_{gb} \varphi(T_i, T_j).$$

Proof. Let i, j, k, and l be the processes indices in $T_{G'}^4$. $G' = (V', E')$ is any graph where $V' = \{i, j, k, l\}$, $v(G, g, h) = v(G', i, j)$, and $(k, j), (l, i) \in E'$. According to [31] such a graph always exists.

Assume $T_G^n \not\models_{gb} \varphi(T_g, T_h)$. Then there exists $b \in \mathbb{N}$ such that $\forall k' \in \mathbb{N}$ there is a run x of T_G^n where b-gfair(x), and $(x, 0, k' \cdot (|T| + 1)) \not\models \varphi(T_g, T_h)$. Let $d = |T| + 1$, and $b' = 2|T| + b + (b - n + 2) \cdot |T|$. We show how to construct for every such x a run y of $T_{G'}^4$ where b'-gfair(y), $x(T_g, T_h) \equiv_d y(T_i, T_j)$.

Construction. Let $x = (s_0, ac_0)(s_1, ac_1)\ldots$ and

$$y' = ((s_0(T_g, T_h), q^{rcv}, q^{rcv}), ac_0(T_g, T_h))((s_1(T_g, T_h), q^{rcv}, q^{rcv}), ac_1(T_g, T_h))\ldots$$

The word y' is a sequence of configurations of $T_{G'}^4$, where we assign the local runs of T_g, T_h into the local runs of T_i and T_j. Note that y' is not a run, hence to obtain a run, first let

$$y'' = ((s_0(T_g, T_h), q^{\mathsf{init}}, q^{\mathsf{init}}), \epsilon)\ldots((s_0(T_g, T_h), q^{rcv}, q^{rcv}), ac_0(T_g, T_h))$$
$$((s_1(T_g, T_h), q^{rcv}, q^{rcv}), ac_1(T_g, T_h))\ldots$$

If neither T_g nor T_h has the token in the initial state of x, then, if T_g has the token first in x before T_h, we replace the pair of consecutive configurations

$$((s(T_g,T_h),q^{rcv},q^{rcv}),(snd_z,rcv_i))((s'(T_g,T_h),q^{rcv},q^{rcv}),ac'(T_g,T_h))$$

with

$$((s(T_g,T_h),q^{rcv},q^{rcv}),\epsilon)\dots((s(T_g,T_h),q^{snd},q^{rcv}),(snd_i,rcv_i))$$
$$\dots((s'(T_g,T_h),q^{rcv},q^{rcv}),ac'(T_g,T_h))$$

where $z \in V$. Similarly we deal with the case where T_h has the token before T_g. Furthermore, for every occurrence of a pair of consecutive configurations

$$pair_i = ((s(T_g,T_h),q^{rcv},q^{rcv}),(snd_i,rcv_z))((s'(T_g,T_h),q^{rcv},q^{rcv}),ac'(T_g,T_h))$$

where $s, s' \in Q^n_{\scriptscriptstyle T}, z \in V \setminus \{j\}, ac' \in \Sigma$, then:

- If after $pair_i$ in y'' T_i executes the receive action without a receive action from T_j in between, then $(i,l),(l,i) \in E'$, and we replace $pair_i$ with the sequence:

  $$((s(T_g,T_h),q^{rcv},q^{rcv}),(snd_i,rcv_l))\dots((s(T_g,T_h),q^{snd},q^{rcv}),(snd_l,rcv_i))$$
  $$\dots((s'(T_g,T_h),q^{rcv},q^{rcv}),ac'(T_g,T_h))$$

  Informally we let the process T_l receive the token from T_i and send it immediately back to T_i.

- If after $pair_i$ in y'' T_j receives the token through some other process(es) (different than T_i and T_j), then $(i,k),(k,j) \in E'$, and we replace $pair_i$ with the sequence:

  $$((s(T_g,T_h),q^{rcv},q^{rcv}),(snd_i,rcv_k))\dots((s(T_g,T_h),q^{rcv},q^{snd}),(snd_k,rcv_j))$$
  $$\dots((s'(T_g,T_h),q^{rcv},q^{rcv}),ac'(T_g,T_h))$$

  Informally we let the process T_k receive the token from T_i and sends *immediately* back to T_j.

Next, we do the same for every occurrence of a pair of consecutive configurations

$$pair_j = ((s(T_g,T_h),q^{rcv},q^{rcv}),(snd_j,rcv_z))((s'(T_g,T_h),q^{rcv},q^{rcv}),ac'(T_g,T_h))$$

where $s, s' \in Q^n_{\scriptscriptstyle T}, z \in V \setminus \{i\}, ac' \in \Sigma$.

Furthermore, in x between moments t and $t+b$, T_g and T_h can send the token at most $b-n+2$ times, and whenever T_l or T_k receives the token, it takes at most $|T|$ steps before reaching q^{rcv} again. Finally, note that the number of steps T_l or T_k takes to reach q^{rcv} for the first time is also bounded by $|T|$. Therefore we have b'-gfair(y) and $x(T_g,T_h) \equiv_d y(T_i,T_j)$ $(b' \leq b \cdot d)$ which by Corollary 1 implies that $(y,0,k') \not\models \varphi(T_i,T_j)$. □

4.5 Conclusions

We have investigated the behavior of concurrent systems with respect to promptness properties specified in Prompt-$\mathsf{LTL}\backslash\mathbf{X}$. Our first important observation is that Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ is not stutter insensitive, so the standard notion of stutter equivalence is insufficient to compare traces of concurrent systems if we are interested in promptness. Based on this, we have defined *bounded stutter equivalence*, and have shown that Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ is *bounded stutter insensitive*.

We have shown how this allows us to obtain cutoff results for guarded protocols and token-passing systems, and have obtained cutoffs for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ (with locally or globally bounded fairness) that are the same as those that were previously shown for $\mathsf{LTL}\backslash\mathbf{X}$ (with unbounded fairness). This implies that, for the cases where we do obtain cutoffs, the PMCP for Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ has the same asymptotic complexity as the PMCP for $\mathsf{LTL}\backslash\mathbf{X}$.

Finally, we note that together with methods for distributed synthesis from Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ specifications [68], our cutoff results enable the synthesis of parameterized systems based on the *parameterized synthesis* approach [61] that has been used to solve challenging synthesis benchmarks by reducing them to systems with a small number of components [18, 74].

Chapter 5

Parameterized Repair of Concurrent Systems

When considering systems that are composed of an arbitrary number of processes, methods such as parameterized model checking can provide correctness guarantees that hold regardless of the number of processes. Nonetheless, if parameterized model checking reveals an error in a system, it does not say how the system can be repaired such that it satisfies the specifications. To repair the system, a user has to analyze thoroughly a counter example returned by a model checker in order to identify which behavior led to the error, and how to avoid such unwanted behavior. Both tasks may be nontrivial in a parameterized concurrent system, due to the fact that an error can be either an internal issue of one of the process templates or a problem in the synchronization between process templates' instances. In this chapter, given non-deterministic process descriptions, we present an approach that automatically detects errors in any system based on these processes, and automatically repairs errors by restricting the given non-determinism, such that any system based on the repaired processes is correct.

5.1 Basic Idea

Similarly to existing repair approaches [8, 69], we begin with a given non-deterministic implementation, in this case an implementation A of the controller process and an implementation B of the uniform user processes of our concurrent system. The non-determinism may have been added by a designer to "propose" possible repairs for a system that is known or suspected to be faulty. The goal is to automatically detect errors in any system based on these components, and automatically repair them by restricting the given non-determinism, such that any system based on the fixed components is correct. In contrast to similar approaches, we not only aim to find the right restrictions on the internal behavior of processes, but also on their communication. That is, the non-determinism may also include different options for synchronization between processes, and the algorithm will find an option that works. For instance, for a disjunctive system, a designer can start with a template that contains all possible synchronizations, i.e. all possible transitions and guards.

To automatically repair a parameterized system, we propose a method that interleaves the generation of candidate solutions (repairs) with parameterized model checking and parameterized deadlock detection approaches. For this sake we check thoroughly which information parameterized model checker and parameterized deadlock detector need to provide in order to direct the search for candidate repairs, and how this information can be encoded into propositional constraints in order to employ SAT solving to automatically find convenient repairs on the process implementations (templates). Figure 6.4 sketches the basic idea.

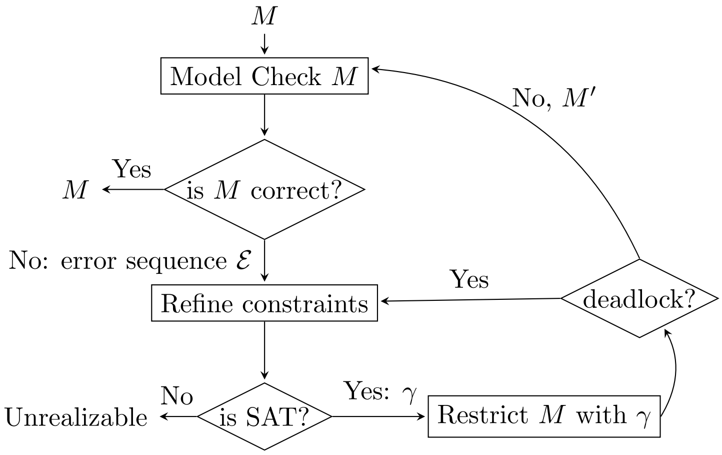


Figure 5.1: Control flow of the algorithm

As a concrete instantiation of the approach, we present an algorithm that is based on model checking of well-structured transition systems [3, 56]. This algorithm supports many classes of systems from the parameterized verification literature, including guarded protocols with disjunctive guards [43], pairwise rendezvous systems [59] and broadcast protocols [48].

Related Work

Despite the many automatic repair approaches that have been studied in the literature [8,38,57,60,69,83], to the best of our knowledge none of these approaches can give correctness guarantees for parameterized concurrent systems.

The repair problem has a close link with the synthesis problem [84, 86], and hence the approach we present in this chapter to *parameterized repair* builds on existing approaches for the synthesis of concurrent systems, e.g. *lazy synthesis* [53].

Many of the existing repair approaches [17,58,81,90,95] that have been proposed for synchronization synthesis and repair of concurrent systems are based on the counter example guided synthesis/repair principle or learning-based algorithms [58]. However none of the above handles parameterized systems, i.e., they are restricted to fixed size systems. In [17, 95] the authors present algorithms that restrict processes' execution interleaving by adding atomic sections. An atomic section defines program statements that need to be executed without intermediate context switch (interruption from another process). In *Assume-*

Guarantee-Repair [58], the authors also combine verification and repair where they use a learning-based assume-guarantee algorithm to find counter examples, and then restrict transitions guards to avoid these unwanted behaviors of the system. In contrast to our results, this approach cannot guarantee the termination of the repair process. However, the processes considered in [58], as well as in [17,95], are more general than ours, and may include constraints or variables over infinite domains under some restrictions. In [81] the authors introduce an algorithm for the synthesis of synchronizations among controller processes in a software-defined network. As a system model, they use event net (a special form of a Petri net) in which a place corresponds to a static network configuration. Their approach requires that a marking (distribution of tokens over the net's places) assigns at most one token to each place. Hence, the approach is restricted to fixed size systems.

Outline of the Chapter. In Section 5.2 we introduce our system model that can simulate infinite systems including parameterized systems. In Section 5.3 we show how to model disjunctive systems as well-structured transition systems in order to show that reachability analysis for such systems is decidable. Then we present a parameterized model checking algorithm and a parameterized deadlock detection algorithm and we prove their correctness. We present our repair algorithm in Section 5.4 and we prove its correctness. Finally we show how the repair algorithm can be extended in two orthogonal directions: from reachability to arbitrary safety properties (Sect. 5.5), and from disjunctive systems to other types of systems, like pairwise rendezvous and broadcast protocols (Sect. 5.6).

5.2 System Model

In this section we introduce a model that can simulate a system with an arbitrary number of processes. This model is restricted to disjunctive systems, however in later sections we present similar models that can simulate other type of systems.

Given a process template $U = (Q_U, \mathsf{init}_U, \delta_U)$, we denote by t_U a transition of U, i.e. $t_U = (q_U, g, q'_U) \in \delta_U$, and by $\delta_U(q_U)$ the set of all outgoing transitions of $q_U \in Q_U$. We assume that δ_U is *total*, i.e., for every $q_U \in Q_U$ there exists some transition $(q_U, g, q'_U) \in \delta_U$.

Counter System. A *configuration* of a system $A\|B^n$ is a tuple $(q_A, \vec{c})$, where $q_A \in Q_A$, and $\vec{c} : Q_B \to \mathbb{N}_0$. For $Q_B = \{q_0, \ldots, q_{|B|-1}\}$, we identify $\vec{c}$ with the vector $(\vec{c}(q_0), \ldots, \vec{c}(q_{|B|-1})) \in \mathbb{N}_0^{|B|}$, and also use $\vec{c}(i)$ to refer to $\vec{c}(q_i)$. Intuitively, $\vec{c}(i)$ indicates how many processes are in state q_i. We denote by $\vec{u_i}$ the vector given by $\vec{u_i}(i) = 1$ and $\vec{u_i}(j) = 0$ for $i \neq j$.

Given a configuration $\sigma = (q_A, \vec{c})$, we say that the guard g of a local transition $(q_U, g, q'_U) \in \delta_U$ is *satisfied in* σ, denoted $\sigma \models_{q_U} g$, if one of the following conditions hold:

(a) $q_U = q_A$, and $\exists q_i \in Q_B$ with $q_i \in g$ and $\vec{c}(i) \geq 1$
(A takes the transition, a B-process is in g)

(b) $q_U \neq q_A$, $\vec{c}(q_U) \geq 1$, and $q_A \in g$
(B-process takes the transition, A is in g)

(c) $q_U \neq q_A$, $\vec{c}(q_U) \geq 1$, and $\exists q_i \in Q_B$ with $q_i \in g$, $q_i \neq q_U$ and $\vec{c}(i) \geq 1$ (B-process takes the transition, another B-process is in different state in g)

(d) $q_U \neq q_A$, $q_U \in g$, and $\vec{c}(q_U) \geq 2$ (B-process takes the transition, another B-process is in same state in g)

We also say that the local transition (q_U, g, q'_U) is *enabled* in σ.

Informally, the guard g of a local transition (q_U, g, q'_U) is satisfied if and only if there is a process in q_U and there is another process (i.e., not the one taking the transition) in one of the local states of the guard g.

Then the *configuration space* of all systems $A \| B^n$, for fixed A, B but arbitrary $n \in \mathbb{N}$, is the transition system $M = (\Omega, \Omega_0, \Delta)$ where:

- $\Omega \subseteq Q_A \times \mathbb{N}_0^{|B|}$ is the set of states,
- $\Omega_0 = \{(init_A, \vec{c}) \mid \forall q \in Q_B : \vec{c}(q) = 0 \text{ if } q \neq init_B)\}$ is the set of initial states,
- Δ is the set of transitions $((q_A, \vec{c}), (q'_A, \vec{c'}))$ such that one of the following holds:
    1. $\vec{c} = \vec{c'} \wedge \exists (q_A, g, q'_A) \in \delta_A : (q_A, \vec{c}) \models_{q_A} g$ (transition of A)
    2. $\exists (q_i, g, q_j) \in \delta_B : \vec{c}(q_i) \geq 1 \wedge \vec{c'} = \vec{c} - \vec{u_i} + \vec{u_j} \wedge (q_A, \vec{c}) \models_{q_i} g$ (transition of some B)

We will also call M the *counter system* (of A and B), and will call configurations *states* of M, or *global states*.

A process is *enabled* in a global state σ if at least one of its transitions is enabled in σ, otherwise it is *disabled*.

Let $\sigma, \sigma' \in \Omega$ be states of M, and $U \in \{A \cup B\}$. For a transition $(\sigma, \sigma') \in \Delta$ we also write $\sigma \rightarrow \sigma'$. If the transition is based on the local transition $t_U = (q_U, g, q'_U) \in \delta_U$, we also write $\sigma \xrightarrow{t_U} \sigma'$ or $\sigma \xrightarrow{g} \sigma'$. Let $\Delta^{local}(\sigma) = \{t_U \mid \sigma \xrightarrow{t_U} \sigma'\}$, i.e., the set of all enabled outgoing local transitions from σ, and let $\Delta(\sigma, t_U) = \sigma'$ if $\sigma \xrightarrow{t_U} \sigma'$. If σ' can be reached from σ by executing one or more local transition, we write $\sigma \rightarrow^* \sigma'$.

The above system model is similar to the configuration space of a vector addition systems with states (VASS). That is, guarded protocols and pairwise rendezvous systems can directly be modeled as VASS, whereas broadcast synchronization requires an extension of the framework, e.g. to affine VASS [21].

Runs. A *path* of a counter system is a sequence of global states $x = \sigma_1, \sigma_2, \ldots$ such that for all $m < |x|$ there is a transition $\sigma_m \rightarrow \sigma_{m+1}$ based on some local transition. A path can be finite or infinite, and a *maximal path* is a path that cannot be extended. A system *run* is a maximal path starting in the initial configuration. Runs are either infinite, or they end in a configuration where no transition is enabled. We say that a run is *deadlocked* if it is finite. Note that every run $\sigma_1, \sigma_2, \ldots$ of the counter system corresponds to a run of a fixed system $A \| B^n$, i.e., the number of processes does not change during a run. Given a set of error states $E \subseteq \Omega$, an *error path* is a finite path that starts in an initial state and ends in E.

From now on we assume that each guard $g \in \mathcal{G}$ is a singleton. This is not a restriction as any local transition (q_U, g, q'_U) with $|g| > 1$ can be split into $|g|$ transitions $(q_U, g_1, q'_U), \ldots, (q_U, g_{|g|}, q'_U)$ where for all $i \leq |g| : g_i \in g$ is a singleton guard. Note that this does not affect the runs of the system: let $M = (\Omega, \Omega_0, \Delta)$ be a counter system, and let $M' = (\Omega, \Omega_0, \Delta')$ be obtained by splitting transitions as described above, then if $x = (q_{A_1}, \vec{c}_1), (q_{A_2}, \vec{c}_2), \ldots$ is a run of M, then x is also a run of M'. To see that let $(q_A, \vec{c}_i), (q'_A, \vec{c}_{i+1})$ be two consecutive configurations in x, then there exists a local transition $t_U = (q_U, g, q'_U)$ with $g = \{q_{g_1}, \ldots, q_{g_{|g|}}\}$ where $(q_A, \vec{c}_i) \xrightarrow{t_U} (q'_A, \vec{c}_{i+1}) \in \Delta$. We have $(q_A, \vec{c}_i) \models_{q_U} g$ then there exists $g_j = \{q_{g_j}\}$ with $(q_A, \vec{c}) \models_{q_U} g_j$ (check the definition above of $\models_{q_U}$). Thus, if the transition t_U is enabled at moment i in M, then there exists a local transition $t_{U_j} = (q_U, g_j, q'_U)$ that is also enabled at moment i in M'. Similarly if $x = (q_{A_1}, \vec{c}_1), (q_{A_2}, \vec{c}_2), \ldots$ is a run of M', then x is also a run of M.

5.3 Parameterized Model Checking of Disjunctive Systems

As a subprocedure of our parameterized repair algorithm, we will use a parameterized model checker. In the following, we provide a model checking algorithm for counter systems, based on the backward reachability algorithm by Abdulla et al. [3] for well-structured transition systems (WSTSs). That is, we first prove that a counter system is a WSTS (Sect. 5.3.1), then we introduce a concrete parameterized model checking algorithm for disjunctive systems (Sect. 5.3.2), and finally show how the algorithm can be modified to also check for the reachability of deadlocked states (Sect. 5.3.3).

5.3.1 Counter Systems as WSTS

Well-quasi-order. Given a set of states Ω, a binary relation $\preceq \subseteq \Omega \times \Omega$ is a *well-quasi-order* (wqo) if $\preceq$ is reflexive, transitive, and if any infinite sequence $\sigma_0, \sigma_1, \ldots \in \Omega^\omega$ contains a pair $\sigma_i \preceq \sigma_j$ with $i < j$. A subset $R \subseteq \Omega$ is an *antichain* if any two distinct elements of R are incomparable wrt. $\preceq$. Therefore, $\preceq$ is a wqo on Ω if and only if it is well-founded and has no infinite antichains.

Upward-closed Sets. Let $\preceq$ be a wqo on Ω. The *upward closure* of a set $R \subseteq \Omega$, denoted $\uparrow R$, is the set $\{\sigma \in \Omega \mid \exists \sigma' \in R : \sigma' \preceq \sigma\}$. We say that a set R is *upward-closed* if $\uparrow R = R$. Let R be an upward-closed set, then we call $B \subseteq S$ a *basis* of R if $\uparrow B = R$. From the definition of wqo it follows directly that any basis of R has a unique subset of minimal elements. We call this set the *minimal basis* of R, denoted by $minBasis(R)$.

Compatibility. Given a counter system $M = (\Omega, \Omega_0, \Delta)$, we say that a wqo $\preceq \subseteq \Omega \times \Omega$ is *compatible* with Δ if the following holds:

$$\forall \sigma, \sigma', r \in S : \text{ if } \sigma \to \sigma' \text{ and } \sigma \preceq r \text{ then } \exists r' \text{ with } \sigma' \preceq r' \text{ and } r \to^* r'$$

We say $\preceq$ is *strongly compatible* with Δ if the above holds with $r \to r'$ instead of $r \to^* r'$.

WSTS [3]. We say that $(M, \preceq)$ with $M = (\Omega, \Omega_0, \Delta)$ is a *well-structured transition system* if $\preceq$ is a wqo on Ω that is compatible with Δ.

Lemma 22. *Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system for guarded process templates A, B, and let $\lessapprox \subseteq \Omega \times \Omega$ be the binary relation defined by:*

$$(q_A, \vec{c}) \lessapprox (q'_A, \vec{d}) \;\Leftrightarrow\; \Big(q_A = q'_A \wedge \vec{c} \lesssim \vec{d}\Big)\,,$$

where $\lesssim$ is the component-wise ordering of vectors. Then $(M, \lessapprox)$ is a WSTS.

Proof. The partial order $\lessapprox$ is a wqo due to the fact that $\lesssim$ is a wqo. Moreover, we show that $\lessapprox$ is strongly compatible with Δ. Let $\sigma = (q_A, \vec{c}), \sigma' = (q'_A, \vec{c}'), r = (q_A, \vec{d}) \in \Omega$ such that $\sigma \xrightarrow{t_U} \sigma' \in \Delta$ and $\sigma \lessapprox r$. Since the transition t_U is enabled in σ, it is also enabled in r and $\exists r' = (q'_A, \vec{d}') \in \Omega$ with $r \xrightarrow{t_U} r' \in \Delta$. Then it is easy to see that $\sigma' \lessapprox r'$: either t_U is a transition of A, then we have $\vec{c} = \vec{c}'$ and $\vec{d} = \vec{d}'$, or t_U is a transition of B with $t_U = (q_i, g, q_j)$, then $q_A = q'_A$ and $\vec{c}' = \vec{c} - \vec{c}_i + \vec{c}_j \lesssim \vec{d} - \vec{c}_i + \vec{c}_j = \vec{d}'$. □

Predecessor, Effective *pred*-basis [56]. Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system and let $R \subseteq \Omega$. Then the set of *immediate predecessors* of R is

$$pred(R) = \{\sigma \in \Omega \mid \exists r \in R : \sigma \rightarrow r\}.$$

We say that a WSTS $(M, \lessapprox)$ has *effective pred-basis* if there exists an algorithm that takes as input any finite set $R \subseteq \Omega$ and returns a finite basis of $\uparrow pred(\uparrow R)$.

One can easily show the following connection between strong compatibility and the immediate predecessor of an upward-closed set:

Corollary 5. *Let $R \subseteq \Omega$ be upward-closed with respect to $\lessapprox$. Then $pred(R)$ is upward-closed iff $\preceq$ is strongly compatible with Δ.*

For backward reachability analysis, we want to compute $pred^*(R)$ as the limit of the sequence $R_0 \subseteq R_1 \subseteq \ldots$ where $R_0 = R$ and $R_{i+1} = R_i \cup pred(R_i)$. Note that if we have strong compatibility and effective pred-basis, we can compute $pred^*(R)$ for any upward-closed set R. If we can furthermore check intersection of upward-closed sets with initial states (which is easy for counter systems), then reachability of arbitrary upward-closed sets is decidable.

Lemma 23. *Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system for guarded process templates A, B. Then $(M, \lessapprox)$ has effective pred-basis.*

Proof. Let $R \subseteq \Omega$ be finite. By Corollary 5, it is sufficient to prove that a *basis* of $pred(\uparrow R)$ can be computed from R. Let $g = \{q_t\}$, $f = ((t = j \wedge \vec{c}'(j) =$

$1) \vee (\vec{c'}(t) \geq 1 \wedge \vec{c'}(j) = 0))$. Consider the following set of states:

$$\begin{aligned}
\mathit{CBasis} = \{(q_A, \vec{c}) \in \Omega \mid \exists (q'_A, \vec{c'}) \in R :& \\
& [\ \exists (q_A, g, q'_A) \in \delta_A \wedge (q_A, \vec{c}) \models_{q_A} g \\
& \wedge (\ (\vec{c} = \vec{c'}) \vee (\vec{c'}(t) = 0 \wedge \vec{c} = \vec{c'} + \vec{u}_t)\ )\ ] \\
& \vee [\ \exists (q_i, g, q_j) \in \delta_B \wedge (q_A, \vec{c}) \models_{q_i} g \wedge q_A = q'_A \\
& \wedge \big(\ \ \vec{c'} = \vec{c} - \vec{u}_i + \vec{u}_j \\
& \vee (\ \vec{c'}(t) = 0 \wedge \vec{c'}(j) \geq 1 \wedge \vec{c'} + \vec{u}_t = \vec{c} - \vec{u}_i + \vec{u}_j) \\
& \vee (f \wedge \vec{c'} + \vec{u}_j = \vec{c} - \vec{u}_i + \vec{u}_j) \\
& \vee (\ \vec{c'}(t) = 0 \wedge \vec{c'}(j) = 0 \wedge \vec{c'} + \vec{u}_t + \vec{u}_j = \vec{c} - \vec{u}_i + \vec{u}_j)\ \ \big)\ ]\ \}
\end{aligned}$$

Clearly, $\mathit{CBasis} \subseteq \mathit{pred}(\uparrow R)$, and CBasis is finite. We claim that also $\mathit{CBasis} \supseteq \mathit{minBasis}(\mathit{pred}(\uparrow R))$. For the purpose of reaching a contradiction, assume $\mathit{CBasis} \not\supseteq \mathit{minBasis}(\mathit{pred}(\uparrow R))$, which implies that there exists a $(q_A, \vec{c}) \in (\mathit{minBasis}(\mathit{pred}(\uparrow R)) \cap \neg \mathit{CBasis})$. Since $(q_A, \vec{c}) \notin \mathit{CBasis}$, there exists $(q'_A, \vec{c'}) \notin R$ with $(q_A, \vec{c}) \to (q'_A, \vec{c'})$ and since $(q_A, \vec{c}) \in \mathit{minBasis}(\mathit{pred}(\uparrow R))$, there is a $(q'_A, \vec{d'}) \in R$ with $(q'_A, \vec{d'}) \lnsim (q'_A, \vec{c'})$. We differentiate between two cases:

- Case 1: Suppose $(q_A, \vec{c}) \xrightarrow{t_A} (q'_A, \vec{c'})$ with $t_A = (q_A, g, q'_A) \in \delta_A$ and $(q_A, \vec{c}) \models_{q_A} g$. Then $\vec{c} = \vec{c'}$, and by definition of CBasis there exists $(q_A, \vec{d}) \in \mathit{CBasis}$ with $[(q_A, \vec{d}) \to (q'_A, \vec{d'}) \wedge \vec{d} = \vec{d'} \wedge \vec{d'}(t) \geq 1]$ or $[(q_A, \vec{d}) \to (q'_A, \vec{d'} + u_t) \wedge \vec{d} = \vec{d'} + u_t \wedge \vec{d'}(t) = 0]$. Furthermore, we have $\vec{d'} \lesssim \vec{c'}$, which implies $(q_A, \vec{d}) \lnsim (q_A, \vec{c})$ with $(q'_A, \vec{d'}) \in R$. Contradiction.

- Case 2: Suppose $(q_A, \vec{c}) \xrightarrow{t_B} (q'_A, \vec{c'})$ with $t_B = (q_i, g, q_j) \in \delta_B$ and $(q_A, \vec{c}) \models_{q_i} g$. Then $q_A = q'_A \wedge \vec{c} = \vec{c'} + \vec{u}_i - \vec{u}_j$. By definition of CBasis there exists $(q_A, \vec{d}) \in \mathit{CBasis}$ such that one of the following holds:

  - $(q_A, \vec{d}) \to (q'_A, \vec{d'})\ \wedge\ \vec{d'} = \vec{d} - \vec{u}_i + \vec{u}_j$
  - $\vec{d'}(t) = 0 \wedge \vec{d'}(j) \geq 1 \wedge (q_A, \vec{d}) \to (q'_A, \vec{d'} + \vec{u}_t)\ \wedge\ \vec{d'} + \vec{u}_t = \vec{d} - \vec{u}_i + \vec{u}_j$
  - $f \wedge (q_A, \vec{d}) \to (q'_A, \vec{d'} + \vec{u}_j)\ \wedge\ \vec{d'} + \vec{u}_j = \vec{d} - \vec{u}_i + \vec{u}_j$
  - $\vec{d'}(t) = 0 \wedge \vec{d'}(j) = 0 \wedge (q_A, \vec{d}) \to (q'_A, \vec{d'} + \vec{u}_t + \vec{u}_j)\ \wedge\ \vec{d'} + \vec{u}_t + \vec{u}_j = \vec{d} - \vec{u}_i + \vec{u}_j$

  Furthermore, we have $\vec{d'} \lesssim \vec{c'}$, which implies that $(q_A, \vec{d}) \lnsim (q_A, \vec{c})$ with $(q_A, \vec{d}) \in \mathit{minBasis}(\mathit{pred}(\uparrow R))$. Contradiction.

□

5.3.2 Model Checking Algorithm

The model checking algorithm we present here is a variant of the known backwards reachability algorithm for WSTS [3]. We present it in detail to show how it stores intermediate results and returns not only a yes/no answer, but an *error sequence*. This is necessary for a model checker that is to be used in our repair algorithm, and understanding how error sequences are computed makes it easier to understand the correctness of that algorithm.

Detailed Description of Algorithm 1. Algorithm 1 takes as argument a counter system M and a finite basis ERR of the set of error states. In a loop, it computes the set of predecessors and checks whether it has reached a set that contains initial states, or a fixed point. If either of these is true, it terminates, otherwise it repeats the loop. The procedure returns either $True$, i.e. the system is safe, or an *error sequence* $E_0, \ldots, E_k$, where $E_0 = ERR$, $\forall 0 < i < k :$ $E_i = minBasis(pred(\uparrow E_{i-1}))$, and $E_k = minBasis(pred(\uparrow E_{k-1})) \cap \Omega_0$. That is, every E_i is the minimal basis of the set of states that can reach ERR in i steps. The steps of the algorithm are explained in more detail in the following.

ModelCheck: Line 2 initializes $tempSet$ and E_0 with ERR, i to 1, and $visitedSet$ (the set of visited states) with $\emptyset$. In Line 3 we enter a while loop that is left when a fixed point is reached, and Line 4 updates the set of visited states to include what we have computed in the last iteration of the loop, stored in $tempSet$. Line 5 computes the next error set E_i, based on the construction used in the proof of Lemma 23. Line 6 checks if the intersection with the initial states is non-empty, and the computed error sequence is returned if this is true. Line 8 adds the E_i that has been computed in this iteration to $visitedSet$ and updates $tempSet$ with the minimal basis of the result. If $visitedSet = tempSet$ at this point, we have reached a fixed point and Line 7 returns True.

Algorithm 1 Parameterized Model Checking

```
 1: procedure ModelCheck($Counter\ System\ M$,$ERR$)
 2:     $tempSet \leftarrow ERR$, $E_0 \leftarrow ERR$, $i \leftarrow 1$, $visitedSet \leftarrow \emptyset$
 3:     while $tempSet \neq visitedSet$ do
 4:         $visitedSet \leftarrow tempSet$
 5:         $E_i \leftarrow minBasis(pred(\uparrow E_{i-1}))$
 6:         if $E_i \cap \Omega_0 \neq \emptyset$ then
 7:             return $False, \{E_0, \ldots, E_i \cap \Omega_0\}$
 8:         $tempSet \leftarrow minBasis(visitedSet \cup E_i)$
 9:         $i \leftarrow i + 1$
10:     return $True, \emptyset$
```

Example. Consider again the reader-writer system in Figures 4.2 and 4.3. Suppose the error states are all states where the writer is in w while a reader is in r. In other words, the error set of the corresponding counter system M is $\uparrow\{(w,(0,1))\}$ where $(0,1)$ means zero reader-processes are in nr and one in r. Then if we run Algorithm 1 with the parameters $M, \{(w,(0,1))\}$, we get the following error sequence: $E_0 = \{(w,(0,1))\}$, $E_1 = \{(nw,(0,1))\}$, $E_2 = \{(nw,(1,0))\}$, with $E_2 \cap \Omega_0 \neq \emptyset$, i.e., the error is reachable.

Properties of Algorithm 1. Correctness of the algorithm follows from the correctness of the algorithm by Abdulla et al. [3], and from Lemma 23. Termination follows from the fact that a non-terminating run would produce an infinite minimal basis, which is impossible since a minimal basis is an antichain.

Moreover, note that due to the cutoff results for disjunctive systems [43, 65] and the fact that our algorithm finds shortest error paths, any state on the error paths will have a number of processes smaller than the cutoff. Let c be the cutoff for disjunctive systems of the form $A\|B^n$. Then if an error is reached in the cutoff system, $A\|B^c$, it will be reached in any system $A\|B^n$ with $n \geq c$ (the other direction holds too). In other words, we need at most c processes to reach the error, if the error is reachable. On the other hand, the backward model checking algorithm presented in this section guarantees to find errors with the least possible number of processes, due to the fact that it starts from the minimal basis ERR of the error set. That is, for a given state $\sigma \in ERR$, if a state $\sigma_0 \in \Omega_0$ could reach the upwards closure of σ, i.e. $\uparrow\sigma$, then σ_0 must include at least the same number of processes in σ. Furthermore, in each computation of the predecessor, at most one process is added to a state in the current set (check the proof of Lemma 23). That means, during the backward computation, if an initial state has not been reached yet, and if nb is the least number of processes in a state in the last computed predecessor set, then with $nb - 1$ processes, the set $\uparrow ERR$ is not reachable.

5.3.3 Deadlock Detection in Disjunctive Systems

The repair of concurrent systems is much harder than fixing monolithic systems. One of the sources of complexity is that a fix for a faulty concurrent system might introduce a deadlock, which is usually an unwanted behavior. In this section we show how to extend our parameterized model checking algorithm to the detection of deadlocks.

To this end, we need to refine the wqo $\lessapprox$ that is used in Section 5.3.1, since a set of deadlocked states is in general not upward-closed under $\lessapprox$: let $\sigma = (q_A, \vec{c}), r = (q_A, \vec{d})$ be two global states with $\sigma \lessapprox r$. If σ is deadlocked, then $\vec{c}(i) = 0$ for every q_i that appears in a guard of an outgoing local transion from σ. Now if $\vec{d}(i) > 0$ for one of these q_i, then the corresponding transition is enabled in r, which is therefore not deadlocked.

Refined wqo for Deadlock Detection. Consider $\lesssim_0 \subseteq \mathbb{N}_0^{|B|} \times \mathbb{N}_0^{|B|}$ where

$$\vec{c} \lesssim_0 \vec{d} \;\Leftrightarrow\; \left(\vec{c} \lesssim \vec{d} \wedge \forall i \leq |B| : \left(\vec{c}(i) = 0 \Leftrightarrow \vec{d}(i) = 0\right)\right),$$

and $\lessapprox_0 \subseteq \Omega \times \Omega$ where

$$(q_A, \vec{c}) \lessapprox_0 (q'_A, \vec{d}) \;\Leftrightarrow\; \left(q_A = q'_A \wedge \vec{c} \lesssim_0 \vec{d}\right).$$

Lemma 24. *Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system for guarded process templates A, B. Then $(M, \lessapprox_0)$ is a WSTS.*

Proof. The partial order $\lessapprox_0$ is obviously reflexive and transitive. Additionally $\lessapprox_0$ has no infinite antichains due to the fact that $\lessapprox$ is a wqo and the comparison of vector positions against zero only introduces a finite case distinction. Thus, $\lessapprox_0$ is a wqo. Moreover we show that $\lessapprox_0$ is compatible with Δ. Let $\sigma = (q_A, \vec{c}), \sigma' = (q'_A, \vec{c'}), r = (q_A, \vec{d}) \in \Omega$ such that $\sigma \xrightarrow{t_U} \sigma'$ and $\sigma \lessapprox_0 r$. We show that there exists $r' = (q'_A, \vec{d'}) \in \Omega$ and a sequence of transitions $r \xrightarrow{t_U}{}^{*} r'$ such that $\sigma' \lessapprox r'$. First, note that since transition t_U is enabled in σ, it is also enabled in r. If t_U is a transition of A then we have $\vec{c} = \vec{d}$ and $\vec{c'} = \vec{d'}$, and we are done. If t_U is a transition of B with $t_U = (q_i, g, q_j)$ and $(\vec{c}(i) > 1 \vee \vec{c}(i) = \vec{d}(i))$ then

$q_A = q'_A$ and $\vec{c'} = \vec{c} - \vec{u}_i + \vec{u}_j \lesssim_0 \vec{d} - \vec{u}_i + \vec{u}_j = \vec{d'}$. However if $\vec{c}(i) = 1 \wedge \vec{d}(i) = b > 1$, then r' is reachable from r by executing t_U b times. In this case, we get $\vec{c'} = \vec{c} - \vec{u}_i + \vec{u}_j \lesssim_0 \vec{d} - b \cdot \vec{u}_i + b \cdot \vec{u}_j = \vec{d'}$. □

As shown in the proof above, $\lessapprox_0$ is compatible with Δ, but not strongly compatible. By Corollary 5, this implies that for a set of states R that is upward-closed (wrt. $\lessapprox_0$), $\mathit{pred}(R)$ is not upward-closed, and we can therefore not use upward-closed sets to compute $\mathit{pred}^*(R)$ when checking reachability of R. Therefore, we now introduce an overapproximation of $\mathit{pred}(R)$ that is upward-closed wrt. $\lessapprox_0$ and is safe in the sense that every state in the overapproximation is backwards reachable (in a number of steps) from R.

O-Predecessor. Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system and let $R \subseteq \Omega$. Then the set of *O-predecessors* of R is

$$\mathit{opred}(R) = \{\sigma \in S \mid \exists r \in R : \sigma \xrightarrow{t_U}{}^{+} r\},$$

where $\sigma \xrightarrow{t_U}{}^{+} r$ denotes that r is reachable from σ by executing the local transition t_U one or more times. We write $\sigma \xrightarrow{t_U}{}^{c} r$ if r is reachable from σ by executing the local transition t_U exactly c times.

Lemma 25. *Let $R \subseteq \Omega$ be upward-closed with respect to $\lessapprox_0$. Then $\mathit{opred}(R)$ is upward-closed with respect to $\lessapprox_0$.*

Proof. Assume $\mathit{opred}(R)$ is not upward-closed. Let $(q_A, \vec{c}) \lessapprox_0 (q_A, \vec{d})$ such that $(q_A, \vec{c}) \in \mathit{opred}(R)$ and $(q_A, \vec{d}) \notin \mathit{opred}(R)$. Since $(q_A, \vec{c}) \in \mathit{opred}(R)$, there exist $(q'_A, \vec{c'}) \in R, c \in \mathbb{N}_0, t_U = (q_i, g, q_j)$ such that $(q_A, \vec{c}) \xrightarrow{t_U}{}^{c} (q'_A, \vec{c'})$ and $\vec{c}(i) \geq c$. Since $(q_A, \vec{c}) \lessapprox_0 (q_A, \vec{d})$ and $c \leq \vec{c}(i) \leq \vec{d}(i)$, t_U is also enabled in $(q_A, \vec{d})$ and can be executed c times. Let $(q'_A, \vec{d'})$ such that $(q_A, \vec{d}) \xrightarrow{t_U}{}^{c} (q'_A, \vec{d'})$. If $c < \vec{c}(i) \vee \vec{c}(i) = \vec{d}(i)$, we have $(q'_A, \vec{c'}) \lessapprox_0 (q'_A, \vec{d'})$. Otherwise, for $c' = \vec{d}(i)$, we have $(q_A, \vec{d}) \xrightarrow{t_U}{}^{c'} (q'_A, \vec{d''})$ and $(q'_A, \vec{d''}) \in R$ (since $(q'_A, \vec{c'}) \lessapprox_0 (q'_A, \vec{d''})$). Hence $(q_A, \vec{d}) \in \mathit{opred}(R)$, which contradicts our assumption. □

Effective opred-basis. Similar to what we had before, we need to have *effective opred-basis*, i.e., to be able to compute a basis of $\mathit{opred}(R)$ from a basis of R.

Lemma 26. *Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system for guarded process templates A, B. Then $(M, \lessapprox_0)$ is a WSTS with effective opred-basis.*

Proof. Let $R \subseteq \Omega$ be finite. Then it suffices to prove that a finite basis of $\mathit{opred}(\uparrow R)$ can be computed from R (where the upward-closure is wrt. $\lessapprox_0$). Consider the following set of states:

$$\begin{aligned}
\mathit{DB\text{-}Pred}(R) = \{(q_A, \vec{c}) \in S \mid \exists (q'_A, \vec{c'}) \in R :& \\
&[\,(q_A, g, q'_A) \in \delta_A \wedge (q_A, \vec{c}) \models_{q_A} g \wedge (\,\vec{c} = \vec{c'}\,)\,] \\
&\vee [\,(q_A = q'_A \wedge (q_i, g, q_j) \in \delta_B) \\
&\quad \wedge ((\vec{c'} = \vec{c} - \vec{u}_i + \vec{u}_j) \vee (\vec{c'}(j) = 1 \wedge \vec{c'} + \vec{u}_j = \vec{c} - \vec{u}_i + \vec{u}_j))\,]\,\}
\end{aligned}$$

That is, *DB-Pred* contains all states that map with a single transition into R, as well as states $(q_A, \vec{c})$ with $(q_A, \vec{c}) \rightarrow (q_A, \vec{c'} + \vec{u}_j)$ for $(q_A, \vec{c'}) \in R$ with $\vec{c'}(j) = 1$.

Obviously, $\textit{DB-Pred}(R) \subseteq \textit{opred}(\uparrow R)$. We claim that also $\textit{DB-Pred}(R) \supseteq \textit{minBasis}(\textit{opred}(\uparrow R))$. For the purpose of reaching a contradiction, assume $\textit{DB-Pred}(R) \not\supseteq \textit{minBasis}(\textit{opred}(\uparrow R))$, and let $(q_A, \vec{c}) \in (\textit{minBasis}(\textit{opred}(\uparrow R)) \cap \neg \textit{DB-Pred}(R))$. Since $(q_A, \vec{c}) \notin \textit{DB-Pred}(R)$, there exist $(q'_A, \vec{c'}) \notin R$ with $(q_A, \vec{c}) \rightarrow (q'_A, \vec{c'})$ and $(q'_A, \vec{d'}) \in R$ with $(q'_A, \vec{d'}) \lessapprox_0 (q'_A, \vec{c'})$. We differentiate between two cases:

- Case 1: $\vec{c} = \vec{c'}$, and $\exists g$ with $(q_A, g, q'_A) \in \delta_A$ and $(q_A, \vec{c}) \models_{q_A} g$. By definition of $\lessapprox_0$ we have $(q_A, \vec{d'}) \models_{q_A} g$, and therefore $(q_A, \vec{d'}) \rightarrow (q'_A, \vec{d'})$ with $(q_A, \vec{d'}) \lessapprox_0 (q_A, \vec{c})$. Contradiction.
- Case 2: $q_A = q'_A$ and $\exists g$ with $(q_i, g, q_j) \in \delta_B$, $(q_A, \vec{c}) \models_{q_i} g$, and $(\vec{c'} = \vec{c} - \vec{u}_i + \vec{u}_j)$. Then $(q_A, \vec{c}) \models_{q_i} g$ implies $(q'_A, (\vec{c'} + \vec{u}_i)) \models_{q_i} g$, and $(q'_A, (\vec{d'} + \vec{u}_i)) \models_{q_i} g$ since $(q'_A, \vec{d'}) \lessapprox_0 (q'_A, \vec{c'})$. We differentiate between two cases:
  - If $\vec{c}(j) = 0$ or $\vec{d'}(j) > 1$ then for $(\vec{d} = \vec{d'} + \vec{u}_i - \vec{u}_j)$ we have $(q_A, \vec{d}) \models_{q_i} g$, $(q_A, \vec{d}) \rightarrow (q'_A, \vec{d'})$, and $(q_A, \vec{d}) \lessapprox_0 (q_A, \vec{c})$. Contradiction.
  - If $\vec{c}(j) > 0 \wedge \vec{d'}(j) \leq 1$ then $\vec{c'}(j) > 1$. Note that $\vec{d'}(j) \neq 0$, as $(q'_A, \vec{d'}) \lessapprox_0 (q'_A, \vec{c'})$, so $\vec{d'}(j) = 1$. Then for $\vec{d} = \vec{d'} + \vec{u}_i - \vec{u}_j$ we have $(q_A, \vec{d})$ and $(q_A, \vec{c})$ are incomparable with respect to $\lessapprox_0$. However, in this case there is also a transition $(q'_A, \vec{d'} + \vec{u}_i) \rightarrow (q'_A, \vec{d'} + \vec{u}_j)$, and by definition of *DB-Pred*, then $(q'_A, \vec{d'} + \vec{u}_i) \in \textit{DB-Pred}(R)$, with $(\vec{d'} + \vec{u}_i)(j) = 1 \leq \vec{c}(j)$, $\vec{d'} + \vec{u}_i \lesssim_0 \vec{c}$, and $(q_A, \vec{d'} + \vec{u}_i) \lessapprox_0 (q_A, \vec{c})$. Contradiction.

□

Let *opred** and *DB-Pred** be defined similarly to *pred**. Then, note that $\textit{pred}(R) \subseteq \textit{opred}(R) \subseteq \textit{pred}^*(R)$ for any $R \subseteq S$, and therefore $\textit{opred}^*(R) = \textit{pred}^*(R)$. This leads to the following.

Corollary 6. *Let $M = (\Omega, \Omega_0, \Delta)$ be a counter system for guarded process templates A, B, and ERR a finite set of deadlocked states. Then $\uparrow(\textit{DB-Pred}^*(ERR)) = \textit{opred}^*(\uparrow ERR) = \textit{pred}^*(\uparrow ERR)$.*

An Algorithm for Deadlock Detection. Based on this result, we can now modify Algorithm 1 to detect a deadlock in a counter system M: instead of passing a basis of the set of errors in the parameter ERR, we pass a basis of the deadlocked states. Furthermore, in Line 5 we now compute *opred* instead of *pred*, which can be done with *DB-Pred* from the previous proof. Finally, the computation of a minimal basis in Lines 5 and 8 needs to be done wrt. the refined wqo $\lessapprox_0$.

5.4 Parameterized Repair

In the following, we present an algorithm to solve the parameterized repair problem. This algorithm is inspired by the lazy synthesis approach of Finkbeiner and Jacobs [53], but it works on a system with infinite state space, making use of the well-quasi-order on the states to perform precise computations.

The Parameterized Repair Problem. Let $M = (\Omega, \Omega_0, \Delta)$ the counter system for guarded process templates $A = (Q_A, \mathsf{init}_A, \mathcal{G}_A, \delta_A)$, $B = (Q_B, \mathsf{init}_B, \mathcal{G}_B, \delta_B)$, and $Err \subseteq Q_A \times \mathbb{N}_0^{|B|}$ a set of error states. The *parameterized repair problem* is to decide if there exist $A' = (Q_A, \mathsf{init}_A, \mathcal{G}_A, \delta'_A)$ with $\delta'_A \subseteq \delta_A$ and $B' = (Q_B, \mathsf{init}_B, \mathcal{G}_B, \delta'_B)$ with $\delta'_B \subseteq \delta_B$ such that the counter system M' for A' and B' does not reach any state in $\uparrow ERR$.

If they exist, we call $\delta'_A \cup \delta'_B$ a *repair* for A and B. We also call M' the *restriction* of M to $\delta'_A \cup \delta'_B$, denoted $Restrict(M, \delta'_A \cup \delta'_B)$.

Note that by our assumption that the local transition relations are total, a trivial repair that disables all transitions from the initial state (or any other state) is not allowed.

5.4.1 Parameterized Repair Algorithm

We introduce a parameterized repair algorithm that interleaves the backwards model checking algorithm (Algorithm 1) with a forward reachability analysis and the computation of candidate repairs.

Forward Reachability Analysis. In the following, for a set $R \subseteq \Omega$, let $Succ(R) = \{\sigma' \in \Omega \mid \exists \sigma \in R : \sigma \rightarrow \sigma'\}$. Furthermore, for $\sigma \in \Omega$, let $\Delta^{local}(\sigma, R) = \{t_U \in \delta \mid t_U \in \Delta^{local}(\sigma) \wedge \Delta(\sigma, t_U) \in R\}$.

Given an error sequence $E_0, \ldots, E_k$, let the *reachable error sequence* $\mathcal{RE} = RE_0, \ldots, RE_k$ be defined by $RE_k = E_k$ (which by definition only contains initial states), and $RE_{i-1} = Succ(RE_i) \cap \uparrow E_{i-1}$ for $1 \leq i \leq k$. That is, each RE_i contains a set of states that can reach $\uparrow ERR$ in i steps, and are reachable from Ω_0 in $k - i$ steps. Thus, it represents a set of concrete error paths of length k.

Constraint Solving for Candidate Repairs. The generation of candidate solutions is guided by a constraint solving approach: a SAT solver receives constraints over the local transitions δ as atomic propositions, and a satisfying assignment of the constraints corresponds to the candidate repair, where only transitions that are assigned true remain in $\delta'_A \cup \delta'_B$. During an execution of the algorithm, these constraints ensure that all concrete error paths discovered so far will be avoided, and additionally include a set of fixed constraints that can express additional desired properties of the system, as explained in the following.

Initial Constraints. To avoid the construction of repairs that violate the totality assumption on the transition relations of the process templates, every repair for disjunctive systems has to satisfy the following constraint:

$$TRConstr_{Disj} = \bigwedge_{q_A \in Q_A} \bigvee_{t_A \in \delta_A(q_A)} t_A \wedge \bigwedge_{q_B \in Q_B} \bigvee_{t_B \in \delta_B(q_B)} t_B$$

Informally, $TRConstr_{Disj}$ guarantees that a candidate repair returned by the SAT solver never removes all local transitions of a local state in $Q_A \cup Q_B$. Furthermore a designer can add any constraints that are needed to obtain a repair that conforms with their requirements (see Sect. 5.6.1 and 5.6.3 for examples of additional constraints).

Detailed description of Algorithm 2. Given a counter system M, a basis ERR of the error states, and initial Boolean constraints $initConstr$ on the transition relation (including at least $TRConstr_{Disj}$), the algorithm returns either a *repair* or the string *Unrealizable* to denote that no repair exists.

ParamRepair: In Line 2 we initialize the candidate solution M', the constraint system $accConstr$ which accumulates all constraints that have been computed so far, and the flag $isCorrect$ before we enter the loop in Line 3. Line 4 uses a model checker (see Algorithm 1) to check if M' is correct, and returns an error sequence $E_0, \ldots, E_k$ if it is not.

If the system is not correct, Line 7 computes the reachable error sequence $RE_0, \ldots, RE_k$. Then, in Line 8 we use $RE_k, \ldots, RE_0$ to construct the Boolean constraint $newConstr$ that ensures that all error paths represented by the reachable error sequence will be avoided in future iterations (see BuildConstr below). In Line 9 this $newConstr$ is added to $accConstr$.

Line 10 calls a SAT solver to check if there exists a solution for $initConstr$ and $accConstr$, which now includes constraints that exclude any error paths found so far. If the constraints are not satisfiable, Line 12 returns the string "Unrealizable" and the algorithm terminates. Otherwise, in Line 14 we use the satisfying assignment γ to compute our new candidate solution $Restrict(M, \gamma)$, and another iteration of the loop starts. Finally, if model checking in Line 4 showed that the current candidate M' is already correct, then Line 14 returns the repair γ and the algorithm terminates.

BuildConstr: This is a recursive procedure that receives the sequence $RE_{k-1}, \ldots, RE_0$ and a state $\sigma \in RE_k$ as input and returns a propositional formula over the set of local transitions that encodes all possible ways for σ to avoid reaching an error: Line 3 asserts that if a state σ is in $\mathcal{RE}[1]$ then all its transitions to $\mathcal{RE}[0]$ are to be deleted. Line 6 asserts that $\forall t_U \in \Delta^{local}(\sigma)$ if t_U leads to a state $\sigma' \in \mathcal{RE}[0]$ then t_U should be deleted, or we need to delete all transitions $t'_U \in \Delta^{local}(\sigma')$ if $\Delta(\sigma', t'_U) \in \mathcal{RE}[1]$.

5.4.2 Properties of Algorithm 2

Theorem 17 (Soundness)**.** *For every repair γ returned by Algorithm 2:*

- *$Restrict(M, \gamma)$ is safe, i.e., $\uparrow ERR$ is not reachable, and*
- *the transition relation of $Restrict(M, \gamma)$ is total in the first two arguments.*

Proof. The parameterized model checker guarantees that the transition relation is safe, i.e., $\uparrow ERR$ is not reachable. Moreover, the transition relation constraint $TRConstr$ is part of $initConstr$ and guarantees that, for any candidate repair returned by the SAT solver, the transition relation is total. □

Theorem 18 (Completeness)**.** *If Algorithm 2 returns "Unrealizable", then the parameterized system has no repair.*

Proof. Algorithm 2 returns "Unrealizable" if $accConstr \wedge initConstr$ has become unsatisfiable.

We consider an arbitrary $\gamma \subseteq \delta$ and show that it can not be a repair. First, note that for the given run of the algorithm, there is an iteration i of the loop such that γ, seen as an assignment of truth values to atomic propositions δ, was

Algorithm 2 Parameterized Repair

```
procedure ParamRepair($M$, $ERR$, $InitConstr$)
    $M' \leftarrow M$, $accConstr \leftarrow True$, $isCorrect \leftarrow False$
    while $isCorrect = False$ do
        $isCorrect, [E_0, \ldots, E_k] \leftarrow ModelCheck(M', ERR)$
        if $isCorrect = False$ then
            $RE_k \leftarrow E_k$   //$E_k$ contains only initial states
            $RE_{k-1} \leftarrow Succ(RE_k) \cap \uparrow E_{k-1}, \ldots, RE_0 \leftarrow Succ(RE_1) \cap \uparrow E_0$
            $newConstr \leftarrow \bigwedge_{\sigma \in RE_k}$ BuildConstr$(\sigma, [RE_{k-1}, \ldots, RE_0]\})$
            $accConstr \leftarrow newConstr \wedge accConstr$
            $\gamma, isSAT \leftarrow SAT(accConstr \wedge initConstr)$
            if $isSAT = False$ then
                return $Unrealizable$
            $M' = Restrict(M, \gamma)$
        else return $\gamma$
```

```
procedure BuildConstr(State $\sigma$, $\mathcal{RE}$)
    //$\mathcal{RE}[1:]$ is a list obtained by removing the first element from $\mathcal{RE}$
    if $\mathcal{RE}[1:]$ is empty then
        return $\bigwedge_{t_U \in \Delta^{local}(\sigma, \mathcal{RE}[0])} \neg t_U$
    else
        return $\bigwedge_{t_U \in \Delta^{local}(\sigma, \mathcal{RE}[0])} (\neg t_U \vee$ BuildConstr$(\Delta(\sigma, t_U), \mathcal{RE}[1:]))$
```

a satisfying assignment of $accConstr \wedge initConstr$ up to this point, and is not anymore after this iteration.

If $i = 0$, i.e., γ was never a satisfying assignment, then γ does not satisfy *initConstr* and can clearly not be a repair. If $i > 0$, then γ is a satisfying assignment for *initConstr* and all constraints added before round i, but not for the constraints $\bigwedge_{\sigma \in RE_k}$ BuildConstr$(\sigma, [RE_{k-1}, \ldots, RE_0]\})$ added in this iteration of the loop, based on a reachable error sequence $\mathcal{RE} = RE_k, \ldots, RE_0$. By construction of BuildConstr, this means we can construct out of γ and $\mathcal{RE}$ a concrete error path in $Restrict(M, \gamma)$, and γ can also not be a repair. □

Theorem 19 (Termination)**.** *Algorithm 2 always terminates.*

Proof. For a counter system based on A and B, the number of possible repairs is bounded by $2^{|\delta|}$. In every iteration of the algorithm, either the algorithm terminates and returns a solution, or it adds constraints that exclude at least the repair that is currently under consideration. Therefore, the algorithm will always terminate. □

Local Witnesses are Upward-closed. We show another property of our algorithm: even though for the reachable error sequence $\mathcal{RE}$ we do not consider the upward closure, the error paths we discover are in a sense upward-closed. This implies that an $\mathcal{RE}$ of length k represents *all possible error paths* of length k. We formalize this in the following.

Given a reachable error sequence $\mathcal{RE} = RE_k, \ldots, RE_0$, we denote by $\mathcal{UE}$ the sequence $\uparrow RE_k, \ldots, \uparrow RE_0$. Furthermore, let a *local witness* of $\mathcal{RE}$ be a sequence

$\mathcal{T}_{\mathcal{RE}} = t_{U_k} \dots t_{U_1}$ where for all $i \in \{1, \dots, k\}$ there exists $\sigma \in RE_i, \sigma' \in RE_{i-1}$ with $\sigma \xrightarrow{t_{U_i}} \sigma'$. We define similarly the local witness $\mathcal{T}_{\mathcal{UE}}$ of $\mathcal{UE}$.

Lemma 27. *Let $\mathcal{RE}$ be a reachable error sequence. Then every local witness $\mathcal{T}_{\mathcal{UE}}$ of $\mathcal{UE}$ is also a local witness of $\mathcal{RE}$.*

Proof. Let $\mathcal{T}_{\mathcal{UE}} = t_{U_k} \dots t_{U_1}$. Then there exist $\sigma_k \in \uparrow E_k = \uparrow RE_k$, $\sigma_{k-1} \in \uparrow RE_{k-1}, \dots, \sigma_0 \in \uparrow RE_0$ such that $\sigma_k \xrightarrow{t_{U_k}} \sigma_{k-1} \xrightarrow{t_{U_{k-1}}} \dots\dots \xrightarrow{t_{U_2}} \sigma_1 \xrightarrow{t_{U_1}} \sigma_0$. Let $\sigma_0 = (q_A^0, \vec{d^0})$, and let $t_{U_1} = (q_{U_{i_1}}, \{q_{t_1}\}, q_{U_{j_1}})$. Then, by construction of $\mathcal{E}$, there exists $(q_A^0, \vec{c^0}) \in E_0, (q_A^1, \vec{c^1}) \in E_1$ with $(q_A^0, \vec{c^0}) \lessapprox (q_A^0, \vec{d^0})$ and $(q_A^1, \vec{c^1}) \xrightarrow{t_{U_1}} (q_A^0, \vec{c^0})$ or $(q_A^1, \vec{c^1}) \xrightarrow{t_{U_1}} (q_A^0, \vec{c^0} + \vec{u}_{t_1})$, hence t_{U_1} is enabled in $(q_A^1, \vec{c^1})$. Using the same argument we can compute $(q_A^2, \vec{c^2}) \in E_2, (q_A^3, \vec{c^3}) \in E_3, \dots$ until we reach the state $(q_A^k, \vec{c^k}) \in E_k$ where t_{U_k} is enabled. Therefore we have the sequence $\sigma_k^R \xrightarrow{t_{U_k}} \sigma_{k-1}^R \xrightarrow{t_{U_{k-1}}} \dots\dots \xrightarrow{t_{U_2}} \sigma_1^R \xrightarrow{t_{U_1}} \sigma_0^R$ with $\sigma_k^R = (q_A^k, \vec{c^k}) \in RE_k = E_k$ and for all $i < k$ we have $\sigma_i^R \in RE_i$, as they are reachable from $\sigma_k^R \in RE_k$ and $(q_A^i, \vec{c^i}) \lessapprox \sigma_i^R$ which guarantees that t_{U_i} is enabled in σ_i^R. □

5.4.3 Parameterized Repair with Deadlock Detection

Note that Algorithm 1 does not include any measures that prevent it from producing a repair that has deadlocked runs. In the repair of concurrent systems, exclusion of deadlocks is particularly important: since letting the system run into a deadlock may be the easiest solution for an automatic repair algorithm to avoid error paths, we can expect the algorithm to produce many deadlocked solutions unless we specifically exclude such cases.

As already sketched in Fig. 6.4, in principle a parameterized deadlock detection mechanism can be added into the loop of refinement and model checking. The difficulty is to provide deadlock detection that decides the problem for parameterized systems, and that can produce meaningful constraints for refinement of our *accConstraint*.

We have already solved both problems in Sect. 5.3.3, where we have shown that deadlock detection for disjunctive systems can be based on the same principles as checking reachability of other errors. Using this approach, it is straightforward to extend Algorithm 1 with a subprocedure for deadlock detection that is called in some interleaving way with the model checker, for example as depicted in Fig. 6.4.

5.5 Beyond Reachability

Algorithm 2 can also be used for repair with respect to general safety properties, based on the automata-theoretic approach to model checking. A safety property asserts that the set of error states must be unreachable. The set of runs that do not violate a safety property can be encoded in a finite-state automaton.

A *finite-state automaton* is a tuple $\mathcal{A} = (Q^{\mathcal{A}}, q_0^{\mathcal{A}}, \Sigma, \delta, \mathcal{F})$ where:

- $Q^{\mathcal{A}}$ is a finite set of states,
- $q_0^{\mathcal{A}}$ is the initial state,

- Σ is an input alphabet,
- $\delta \subseteq Q^{\mathcal{A}} \times \Sigma \times Q^{\mathcal{A}}$ is a transition relation, and
- $\mathcal{F} \subseteq Q^{\mathcal{A}}$ is a set of accepting states.

A *run* of the automaton is a finite sequence $q_0^{\mathcal{A}} a_0 q_1^{\mathcal{A}} a_1 q_2^{\mathcal{A}} a_2 \ldots q_{n+1}^{\mathcal{A}}$ where $\forall i : (q_i^{\mathcal{A}}, a_i, q_{i+1}^{\mathcal{A}}) \in \delta$. A run of the automaton is *accepting* if it visits a state in $\mathcal{F}$.

5.5.1 Checking Safety Properties

Let $M = (Q_A \times \mathbb{N}_0^n, \Omega_0, \Delta)$ be a counter system of process templates A and B that violates $\varphi(A)$, a safety property over the states of A, and let $\mathcal{A} = (Q^{\mathcal{A}}, q_0^{\mathcal{A}}, Q_A, \delta, \mathcal{F})$ be the automaton that accepts all words over Q_A that violate φ. To repair M, the composition $M \times \mathcal{A}$ and the set of error states $ERR = \{((q_A, c), q_{\mathcal{F}}^{\mathcal{A}}) \mid (q_A, c) \in \Omega \wedge q_{\mathcal{F}}^{\mathcal{A}} \in \mathcal{F}\}$ can be given as inputs to the procedure $ParamRepair$. This technique is correct due to the following property that holds on the composition $M \times \mathcal{A}$.

Corollary 7. *Let $\lesssim_{\mathcal{A}} \subseteq (M \times \mathcal{A}) \times (M \times \mathcal{A})$ be a binary relation defined by:*

$$((q_A, \vec{c}), q^{\mathcal{A}}) \lesssim_{\mathcal{A}} ((q'_A, \vec{c}'), q'^{\mathcal{A}}) \Leftrightarrow \vec{c} \lesssim \vec{c}' \wedge q_A = q'_A \wedge q^{\mathcal{A}} = q'^{\mathcal{A}}$$

then $((M \times \mathcal{A}), \lesssim_{\mathcal{A}})$ is a WSTS with effective pred-basis.

Similarly, the algorithm can be used for any safety property $\varphi(A, B^{(k)})$ over the states of A, and of k B-processes $B_1, \ldots, B_k$. To this end, we consider the composition $M \times B^k \times \mathcal{A}$ with $M = (Q_A \times \mathbb{N}_0^n, \Omega_0, \mathcal{G}, \Delta)$, $B = (Q_B, init_B, \mathcal{G}_B, \delta_B)$, and $\mathcal{A} = (Q^{\mathcal{A}}, q_0^{\mathcal{A}}, Q_A \times Q_{B^k}, \delta, \mathcal{F})$ is the automaton that reads states of $A \times B^k$ as actions and accepts all words that violate the property. By symmetry, property $\varphi(A, B^{(k)})$ can be violated by these k explicitly modeled processes iff it can be violated by any combination of k processes in the system.

5.5.2 Example

Consider again the simple reader-writer system in Figures 4.2 and 4.3, and assume that instead of local transition $(nr, \{nw\}, r)$ we have an unguarded transition (nr, Q, r). We want to repair the system with respect to the safety property $\varphi = G[(w \wedge nr_1) \implies (nr_1 W nw)]$ where G, W are the temporal operators *always* and *weak until*, respectively. Figure 5.2 depicts the automaton equivalent to $\neg\varphi$. To repair the system we first need to split the guards as mentioned in Section 5.2, i.e., the transition (nr, Q, r) will become $\{(nr, \{nr\}, r), (nr, \{r\}, r), (nr, \exists nw, r), (nr, \{w\}, r)\}$. Then we consider the composition $\mathcal{C} = M \times B \times \mathcal{A}$ and we run Algorithm 2 on the parameters $\mathcal{C}$, $((-, -, (*, *), q_2^{\mathcal{A}}))$ where $(-, -)$ means any writer state and any reader state, and $*$ means 0 or 1. The model checker in Line 4 may return the following error sequences where we only consider states that didn't occur before:
$E_0 = \{((-, -, (*, *)), q_2^{\mathcal{A}})\}$,
$E_1 = \{((w, r_1, (0, 0)), q_1^{\mathcal{A}})\}$,
$E_2 = \{((w, nr_1, (0, 0)), q_0^{\mathcal{A}}), ((w, nr_1, (0, 1)), q_0^{\mathcal{A}}), ((w, nr_1, (1, 0)), q_0^{\mathcal{A}})\}$,
$E_3 = \{((nw, nr_1, (0, 0)), q_0^{\mathcal{A}}), ((nw, nr_1, (0, 1)), q_0^{\mathcal{A}}), ((w, r_1, (0, 0)), q_0^{\mathcal{A}}),$
$((w, r_1, (0, 1)), q_0^{\mathcal{A}}), ((w, r_1, (1, 0)), q_0^{\mathcal{A}})\}$.

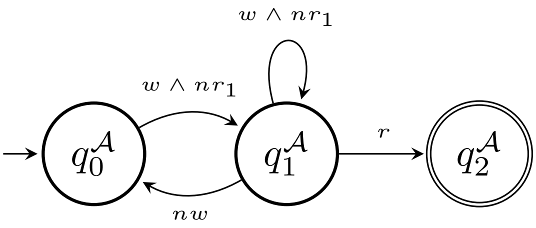


Figure 5.2: Automaton for $\neg\varphi$

In Line 10 we will find out that the error sequence can be avoided if we remove the transitions $\{(nr, \{nr\}, r), (nr, \{r\}, r), (nr, \{w\}, r)\}$. Another call to the model checker in Line 4 finally assures that the new system the new system M'' of $A' = (Q_A, \mathsf{init}_A, \delta_A)$ and $B'' = (Q_B, \mathsf{init}_B, \delta_B \setminus \{(nr, \{w\}, r), (nr, \{nr\}, r), (nr, \{r\}, r)\})$ is safe. Note that some states were omitted from error sequences in order to make presentation simple.

5.6 Beyond Disjunctive Systems

Algorithm 2 is not restricted to disjunctive systems. In principle, it can be used for any system that can be modeled as a WSTS with effective *pred*-basis, as long as we can construct the transition relation constraint ($TRConstr$) for the corresponding system. In this section we show two other classes of systems that can be modeled in this framework: pairwise rendezvous (PR) and broadcast (BC) systems. We introduce transition relation constraints for these systems, as well as a procedure BuildSyncConstr that must be used instead of BuildConstr when a transition relation comprises synchronous actions.

Since these two classes of systems require processes to synchronize on certain actions, we first introduce a different notion of process templates.

Synchronous Processes. A *synchronizing process template* is a transition system $U = (Q_U, \mathsf{init}_U, \Sigma, \delta_U)$ with

- $Q_U \subseteq Q$ is a finite set of states including the initial state init_U,
- $\Sigma = \Sigma_{sync} \times \{?, !, ??, !!\} \cup \tau$ where Σ_{sync} is a set of synchronizing actions, and τ is an internal action,
- $\delta_U : Q_U \times \Sigma \times Q_U$ is a transition relation.

Synchronizing actions like $(a, ?)$ or $(b, !)$ are shortened to $a?$ and $b!$. Intuitively actions of the form $a?$ and $a!$ are PR send and receive actions, respectively, and $a??, a!!$ are BC send and receive actions, respectively.

All processes mentioned in the following are based on a synchronizing process template. We will define global systems based on either PR or BC synchronization in the following subsections.

5.6.1 Pairwise Rendezvous Systems

A PR system [59] consists of a finite number of processes running concurrently. As before, we consider systems of the form $A \| B^n$. The semantics is interleaving, except for actions where two processes synchronize. That is, at every time step, either exactly one process makes an internal transition τ, or exactly two processes synchronize on a single action $a \in \Sigma_{sync}$. For a synchronizing action $a \in \Sigma_{sync}$, the initiator process locally executes the $a!$ action and the recipient process executes the $a?$ action.

Similar to what we defined for disjunctive systems, the configuration space of all systems $A \| B^n$, for fixed A, B but arbitrary $n \in \mathbb{N}$, is the counter system $M^{PR} = (\Omega, \Omega_0, \Delta)$, where:

- $\Omega \subseteq Q_A \times \mathbb{N}_0^{|B|}$ is the set of states,
- $\Omega_0 = \{(init_A, \vec{c}) \mid \forall q_B \in Q_B : \vec{c}(q_B) = 0 \text{ if } q_B \neq init_B)\}$ is the set of initial states,
- Δ is the set of transitions $((q_A, \vec{c}), (q'_A, \vec{c}'))$ such that one of the following holds:

  1. $(q_A, \tau, q'_A) \in \delta_A \wedge \vec{c} = \vec{c}'$ (internal transition A)
  2. $\exists q_i, q_j : (q_i, \tau, q_j) \in \delta_B \wedge c(i) \geq 1 \wedge \vec{c}' = \vec{c} - \vec{u}_i + \vec{u}_j \wedge q_A = q'_A$ (internal transition B)
  3. $a \in \Sigma_{sync} \wedge (q_A, a!, q'_A) \in \delta_A \wedge \exists q_i, q_j : (q_i, a?, q_j) \in \delta_B \wedge c(i) \geq 1, \vec{c}' = \vec{c} - \vec{u}_i + \vec{u}_j$ (synchronizing transition A, B)
  4. $a \in \Sigma_{sync} \wedge (q_A, a?, q'_A) \in \delta_A \wedge \exists q_i, q_j : (q_i, a!, q_j) \in \delta_B \wedge c(i) \geq 1, \vec{c}' = \vec{c} - \vec{u}_i + \vec{u}_j$ (synchronizing transition B, A)
  5. $\exists q_i, q_j : (q_i, a!, q_j) \in \delta_B \wedge \exists q_l, q_m : (q_l, a?, q_m) \in \delta_B \wedge c(i) \geq 1 \wedge c(l) \geq 1 \wedge \vec{c}' = \vec{c} - \vec{u}_i + \vec{u}_j - \vec{u}_l + \vec{u}_m$ (synchronizing transition B, B)

The following result can be considered folklore, a proof can be found in the survey by Bloem et al. [19].

Lemma 28. *Let $M^{PR} = (\Omega, \Omega_0, \Delta)$ be a counter system for process templates A, B with PR synchronization. Then $(M^{PR}, \precapprox)$ is a WSTS with effective pred-basis.*

Initial Constraints. The transition relation constraint for pairwise rendezvous systems, $TRConstr_{PR}$, is defined as follows:

$$TRConstr_{PR} = \bigwedge_{q_A \in Q_A} \bigvee_{t_A \in \delta_A(q_A)} t_A \vee \bigwedge_{q_B \in Q_B} \bigvee_{t_B \in \delta_B(q_B)} t_B$$

We denote by $t_{a?}, t_{a!}$ synchronous local transitions based on an action a, and let $\delta_{AB} = \delta_A \cup \delta_B$. Then the user can add a constraint that guarantees that for all $a \in \Sigma_{sync}$, $t_{a!}$ is deleted if and only if all $t_{a?}$ are deleted.

$$UserConstr_{PR} = \bigwedge_{a \in \Sigma_{sync}} [(t_{a!} \wedge (\bigvee_{t_{a?} \in \delta_{AB}} t_{a?})) \vee (\neg t_{a!} \wedge (\bigwedge_{t_{a?} \in \delta_{AB}} \neg t_{a?}))]$$

Furthermore, the user may want to ensure that in the returned repair, either (a) for all $a \in \Sigma_{sync}$, $t_{a!}$ is deleted if and only if all $t_{a?}$ are deleted, or (b) that synchronized actions are deterministic, i.e., for every state q_U and every synchronized action a, there is exactly one transition on $a?$ from q_U. We give *user constraints* that ensure such behavior.

Denote by $t_{a?}, t_{a!}$ synchronous local transitions based on an action a. Then, the constraint ensuring property (a) is

$$\bigwedge_{a\in\Sigma_{sync}} [(t_{a!} \wedge (\bigvee_{t_{a?}\in\delta} t_{a?})) \vee (\neg t_{a!} \wedge (\bigwedge_{t_{a?}\in\delta} \neg t_{a?}))]$$

To encode property (b), for $U \in \{A, B\}$ and $a \in \Sigma_{sync}$, let $\{t_{q_U}^{a_?^1}, \dots, t_{q_U}^{a_?^m}\}$ be the set of all $a?$ transitions from state $q_U \in Q_U$. Additionally, let $one(t_{q_U}^{a_?}) = \bigvee_{j\in\{1,\dots,m\}}[t_{q_U}^{a_?^j} \bigwedge_{l\neq j} \neg t_{q_U}^{a_?^l}]$. Then, (b) is ensured by

$$\bigwedge_{a\in\Sigma_{sync}} \bigwedge_{q_U\in Q} one(t_{q_U}^{a_?})$$

5.6.2 Deadlock Detection for PR Systems.

German and Sistla [59] have shown that deadlock detection in PR systems is decidable by checking a reachability property of the controller process in a modified system. Thus, at least a rudimentary version of repair including deadlock detection is possible, where the deadlock detection only excludes the current candidate repair, but may not be able to provide constraints on candidates that may be considered in the future.

We leave open the question whether an approach like in Sect. 5.3.3, which could return more meaningful constraints based on deadlock detection, is also possible for PR systems. We note however that deadlocked states in PR systems are also not upward-closed wrt. $\lessapprox$, so a solution would again need a refined wqo.

5.6.3 Broadcast Systems

In broadcast systems, the semantics is interleaving, except for actions where all processes synchronize, with one process "broadcasting" a message to all other processes. Via such a broadcast synchronization, a special process can be selected while the system is running, so we can restrict our model to systems that only contain an arbitrary number of user processes with identical template B.

Formally, at every time step either exactly one process makes an internal transition τ, or all processes synchronize on a single action $a \in \Sigma_{sync}$. For a synchronized action $a \in \Sigma_{sync}$, we say that the initiator process executes the $a!!$ action and all recipient processes execute the $a??$ action. For every action $a \in \Sigma_{sync}$ and every state $q_B \in Q_B$, there exists a state $q'_B \in Q_B$ such that $(q_B, a??, q'_B) \in \delta_B$. Like Esparza et al. [48], we assume w.l.o.g. that the transitions of recipients are deterministic for any given action, which implies that the effect of a broadcast message on the recipients can be modeled by multiplication of a *broadcast matrix*. We denote by M_a the broadcast matrix for action a.

Then, the configuration space of all broadcast systems B^n, for fixed B but arbitrary $n \in \mathbb{N}$, is the counter system $M^{BC} = (\Omega, \Omega_0, \Delta)$ where:

- $\Omega \subseteq \mathbb{N}_0^{|B|}$ is the set of states,
- $\Omega_0 = \{\vec{c} \mid \forall q_B \in Q_B : \vec{c}(q_B) = 0 \text{ iff } q_B \neq init_B)\}$ is the set of initial states,

- Δ is the set of transitions $(\vec{c}, \vec{c}')$ such that one of the following holds:
    1. $\exists q_i, q_j \in Q_B : (q_i, \tau, q_j) \in \delta_B \wedge \vec{c}' = \vec{c} - \vec{u}_i + \vec{u}_j$ (internal transition)
    2. $\exists a \in \Sigma_{sync} : \vec{c}' = M_a \cdot (\vec{c} - \vec{u}_i) + \vec{u}_j$ (broadcast)

Lemma 29. *[48] Let $M^{BC} = (\Omega, \Omega_0, \Delta)$ be a counter system for process template B with BC synchronization. Then $(M^{BC}, \lesssim)$ is a WSTS with effective pred-basis.*

Initial Constraints. The transition relation constraint for broadcast systems is defined as follows:

$$TRConstr_{\mathbf{BC}} = \bigwedge_{q_B \in Q_B} \bigvee_{t_B \in \delta_B(q_B)} t_B$$

We denote by $t_{a??}, t_{a!!}$ synchronous transitions based on an action a. To ensure that for all $a \in \Sigma_{sync}$, $t_{a!!}$ is deleted if and only if all $t_{a??}$ are deleted, the designer can use the following:

$$UserConstr_{BC} = \bigwedge_{a \in \Sigma_{sync}} [(t_{a!!} \wedge (\bigvee_{t_{a??} \in \delta_B} t_{a??})) \vee (\neg t_{a!!} \wedge (\bigwedge_{t_{a??} \in \delta_B} \neg t_{a??}))]$$

Below we show another possible user constraint formula for BC systems that guarantees that for all $a \in \Sigma_{sync}$, $t_{a!!}$ is deleted if and only if all $t_{a??}$ are deleted:

$$\bigwedge_{a \in \Sigma_{sync}} [(t_{a!!} \wedge (\bigwedge_{t_{a??} \in \delta_B} t_{a??})) \vee (\neg t_{a!!} \wedge (\bigwedge_{t_{a??} \in \delta_B} \neg t_{a??}))]$$

Note that the deletion of a transition $(q_B, a??, q'_B)$ is interpreted as if we replace it by the transition $(q_B, a??, q_B)$.

Furthermore, let $\{t_{q_B}^{a^1_{??}}, \ldots, t_{q_B}^{a^m_{??}}\}$ be the set of all $a??$ transitions of the state q_B with $a \in \Sigma_{sync}$, and let $one(t_{q_B}^{a_{??}}) = \bigvee_{j \in \{1,\ldots,m\}} [t_{q_B}^{a^j_{??}} \bigwedge_{l \neq j} \neg t_{q_B}^{a^l_{??}}]$. Then the below formula ensures determinism of synchronized actions:

$$\bigwedge_{a \in \Sigma_{sync}} \bigwedge_{q_B \in Q_B} one(t_{q_B}^{a_{??}})$$

Additionally if our aim is to only fix synchronizations then the following formula states that all non-synchronous actions must be untouchable:

$$\bigwedge_{t_B \in \delta_B^\tau} t_B$$

where $\delta_B^\tau \subseteq \delta_B$ represents the set of local transitions based on action τ.

5.6.4 Synchronous Systems Constraints

The procedure BuildConstr in Algorithm 2 does not take into consideration synchronous actions. Hence, we need a new procedure that offers special treatment for synchronization. To simplify presentation we assume w.l.o.g. that each $a+$, with $+ \in \{!, !!\}$, appears on exactly one local transition. We denote by $\Delta_{sync}(\sigma, a)$ the state obtained by executing action a in state σ. Additionally, let

Algorithm 3 Synchronous Constraint Computation

```
procedure BSC(State $\sigma$, $\mathcal{RE}$)
    if $\mathcal{RE}[1:]$ is empty then
        return $\bigwedge_{t_U \in \Delta^{local}(\sigma, \mathcal{RE}[0])} \neg t_U \bigwedge_{a \in \Sigma_{sync} \wedge \Delta(\sigma,a) \in \mathcal{RE}[0]} T(\sigma, a)$
    else
        return
        $\bigwedge_{t_U \in \Delta^{local}(\sigma, \mathcal{RE}[0])} (\neg t_U \vee BSC(\Delta(\sigma, t_U), \mathcal{RE}[1:]))$
        $\bigwedge_{a \in \Sigma_{sync} \wedge \Delta(\sigma,a) \in \mathcal{RE}[0]} [T(\sigma, a) \vee BSC(\Delta(\sigma, t_a), \mathcal{RE}[1:])]\}$
```

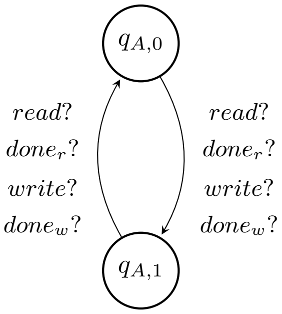


Figure 5.3: Scheduler

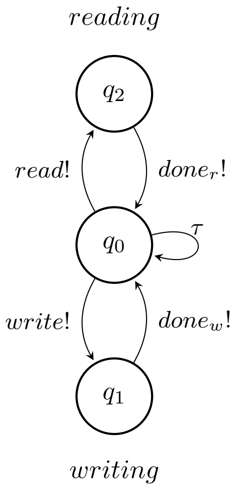


Figure 5.4: Reader-Writer

$\Delta^{local}_{sync}(\sigma, a) = \{(q_U, a_*, q'_U) \in \delta \mid * \in \{?, !, ??, !!\},$ and a is enabled in $\sigma\}$, and let $T(\sigma, a) = \bigvee_{t_a \in \Delta^{local}_{sync}(\sigma,a)} \neg t_a$. In a Broadcast system we say that an action a is enabled in a global state $\vec{c}$ if $\exists i, j < |B|$ s.t. $\vec{c}(i) > 0$ and $(q_{B_i}, a!!, q_{B_j}) \in \delta_B$. In a Pairwise rendezvous system we say that an action a is enabled in a global state $(\vec{c})$ if $\exists i, j, k, l < |B|$ s.t. $\vec{c}(i) > 0, \vec{c}(j) > 0)$ and $(q_{B_i}, a!, q_{B_k}), (q_{B_j}, a?, q_{B_l}), \in \delta_B$.

Given a synchronous system $M^X = (\Omega, \Omega_0, \Sigma, \Delta)$ with $X \in \{BR, PR\}$, a state σ, and a reachable error sequence $\mathcal{RE}$, Algorithm 3 computes a propositional formula over the set of local transitions that encodes all possible ways for a state σ to avoid reaching an error.

5.6.5 Example: Reader-Writer

Consider the parameterized pairwise system that consists of one scheduler (Figure 5.3) and a parameterized number of instances of the reader-writer process template (Figure 5.4). Every state in the scheduler process template has a receive action for every send action. In such a system, the scheduler can not guarantee that, at any moment, there is at most one process in the *writing* state q_1 (Figure 5.4).

Let $t_{U_1} = [q_0, (write!), q_1]$,
$t_{U_2} = [q_{A,0}, (write?), q_{A,1}]$,

$t_{U_3} = [q_{A,1}, (write?), q_{A,0}]$,

$t_{U_4} = [q_0, (read!), q_2]$,

$t_{U_5} = [q_{A,0}, (read?), q_{A,1}]$,

$t_{U_6} = [q_{A,1}, (read?), q_{A,0}]$,

$t_{U_7} = [q_1, (done_w!), q_0]$,

$t_{U_8} = [q_{A,1}, (done_w?), q_{A,0}]$,

$t_{U_9} = [q_{A,0}, (done_w?), q_{A,1}]$,

$t_{U_{10}} = [q_2, (done_r!), q_0]$,

$t_{U_{11}} = [q_{A,1}, (done_r?), q_{A,0}]$,

$t_{U_{12}} = [q_{A,0}, (done_r?), q_{A,1}]$.

Let $ERR = \uparrow\{(q_{A,0}, (0,2,0))(q_{A,1}, (0,2,0))\}$.

Let $UserConstr_{PR} = (t_{U_1} \wedge (t_{U_2} \vee t_{U_3})) \wedge (t_{U_4} \wedge (t_{U_5} \vee t_{U_6})) \wedge (t_{U_7} \wedge (t_{U_8} \vee t_{U_9})) \wedge (t_{U_{10}} \wedge (t_{U_{11}} \vee t_{U_{12}}))$.

Then running our repair algorithm will produce the following results:

First call to model checker returns:

$RE_0 = \{(q_{A,0}, (0,2,0))\}$,

$RE_1 = \{(q_{A,1}, (1,1,0))\}$,

$RE_2 = \{(q_{A,0}, (2,0,0))\}$.

Constraints for SAT:

$accConstr_1 = TRConstr_{PR} \wedge UserConstr_{PR} \wedge (\neg t_{U_1} \vee \neg t_{U_2} \vee \neg t_{U_3})$.

SAT solver solution 1:

$t_{U_2} = false$, $t_{U_6} = false$, $t_{U_9} = false$, and $t_{U_{12}} = false$.

Second call to model checker returns:

$RE_0 = \{(q_{A,0}, (0,2,0))\}$,

$RE_1 = \{(q_{A,1}, (1,1,0))\}$,

$RE_2 = \{(q_{A,0}, (2,1,0))\}$,

$RE_3 = \{(q_{A,1}, (3,0,0))\}$,

$RE_4 = \{(q_{A,0}, (4,0,0))\}$.

Constraints for SAT:

$accConstr_2 = accConstr_1 \wedge (\neg t_{U_1} \vee \neg t_{U_3} \vee \neg t_{U_4} \vee \neg t_{U_5})$.

SAT solver solution 2:

$t_{U_3} = false$, $t_{U_5} = false$, $t_{U_9} = false$, and $t_{U_{12}} = false$.

Third call to model checker returns:

$RE_0 = \{(q_{A,0}, (0,2,0))\}$,

$RE_1 = \{(q_{A,1}, (1,1,0))\}$,

$RE_2 = \{(q_{A,0}, (2,1,0))\}$,

$RE_3 = \{(q_{A,1}, (3,0,0))\}$,

$RE_4 = \{(q_{A,0}, (3,0,0))\}$.

Constraints for SAT:

$accConstr_3 = accConstr_2 \wedge (\neg t_{U_1} \vee \neg t_{U_2} \vee \neg t_{U_4} \vee \neg t_{U_6})$.

SAT solver solution 3:

$t_{U_3} = false$, $t_{U_6} = false$, $t_{U_9} = false$, and $t_{U_{12}} = false$.

The fourth call of the model checker returns true and we obtain the correct scheduler in Figure 5.5.

5.6.6 Example: MESI Protocol

Consider the cache coherence broadcast protocol MESI in Figure 5.6 where:

- M stands for modified and indicates that the cache has been changed.

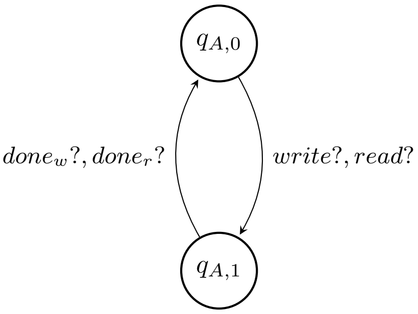


Figure 5.5: Safe Scheduler

- E stands for exclusive and indicates that no other process seizes this cache line.
- S stands for shared and indicates that more than one process hold this cache line.
- I stands for invalid and indicates that the cache's content is not guaranteed to be valid as it might have been changed by some process.

Initially all processes are in I and let a state vector be as follows: (M, E, S, I). An important property for MESI protocol is that a cache line should not be modified by one process (in state M) and in shared state for another process (in state S). In such case the set of error states is: $\uparrow\{(1,0,1,0)\}$. We can run Algorithm 2 on M, $\uparrow\{(1,0,1,0)\}$, $TRConstr_{BC} \wedge \bigwedge_{a\in\Sigma_{sync}} \bigwedge_{q_B\in Q_B} one(t^{a??}_{q_B})$. The model checker will return the following error sequence (non-essential states,e.g. unreachable states, are omitted):
$E_0 = \{(1,0,1,0)\}$,
$E_1 = \{(0,1,1,0)\}$,
$E_2 = \{(0,1,0,1)\}$,
$E_3 = \{(0,0,1,1)\}$,
$E_4 = \{(0,0,0,2)\}$.
Running the procedure BuildSyncConstr (Algorithm 3) in Line 8 will return the following Boolean formula $newConstr =$

$$\neg(I, read!!, S) \vee \neg(I, read??, I) \vee \neg(S, write\text{-}inv!!, E) \vee \neg(I, write\text{-}inv??, I)$$
$$\vee\neg(E, read??, E) \vee \neg(I, read!!, S) \vee \neg(E, write, S)$$

Running the SAT solve in Line 10 on

$$newConstr \wedge TRConstr'_{BC} \wedge \bigwedge_{a\in\Sigma_{sync}} \bigwedge_{q_B\in Q_B} one(t^{a??}_{q_B}) \bigwedge_{t_U\in\{\delta^\tau_B\}} t_U$$

will return the only solution $(E, read??, E) = false$ which clearly fixes the system.

5.7 Conclusion

We have investigated the parameterized repair problem for systems of the form $A\|B^n$ with an arbitrary $n \in \mathbb{N}$. We have introduced a general parameterized

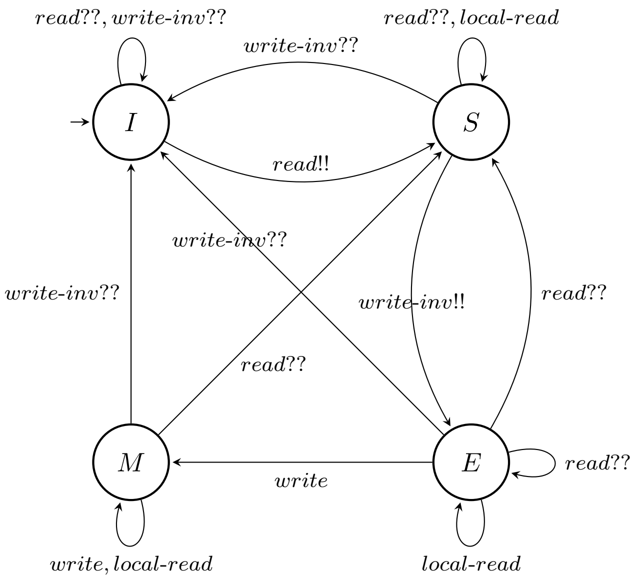


Figure 5.6: MESI protocol

repair algorithm, based on an interleaving of the generation of candidate repairs and parameterized model checking. We have instantiated this general approach to different classes of systems: disjunctive systems, pairwise rendezvous systems, and broadcast systems, and to reachability properties as well as general safety properties, specified by a finite automaton that reads the run of a fixed subset of the processes.

Chapter 6

A Symbolic Algorithm for Lazy Synthesis of Eager Strategies

Previous chapters dealt with the verification and repair of parameterized concurrent systems. In this chapter we study safety synthesis of monolithic systems. As aforementioned, such systems can simulate faithfully the behavior of multiple components or processes running in parallel, however in this case the number of components is fixed. Furthermore, we believe that the approaches for synthesis we present here can be adopted for parameterized systems.

6.1 Safety Synthesis

Automatic synthesis of digital circuits from logical specifications is one of the most challenging and ambitious problems in circuit design. Church [30] was first to identify the problem:

> Given a requirement φ on the input-output behavior of a Boolean circuit, compute a circuit C that satisfies φ.

The problem is usually seen as a game between two players where the "system" player tries to satisfy the specification and the "environment" player tries to violate it. If the system player can win the game we say that he has a winning strategy, and hence this strategy represents a circuit that is guaranteed to satisfy the specification. Many approaches have already been proposed to solve the problem [25, 86] and lately there has been much interest in approaches that use efficient data structures and automated reasoning methods to solve the problem in practice [20, 41, 52, 54, 78, 89].

We consider here only safety specifications. For safety synthesis, most of the efficient symbolic implementations use characteristic functions to manipulate sets of states where these functions are represented as Binary Decision Diagrams (BDDs) [63]. The "standard" approach (algorithm) for safety synthesis uses a backward state space traversal. It starts from the set of unsafe states and computes the set of states from which the environment can obligate the system to reach these undesired states. The negation of this computed set represents the winning region of the system player, and it defines the most permissive winning strategy, i.e., the strategy that allows only the moves (transitions) that do not leave the winning region. Therefore, to solve a safety game we can, as a first step, compute the winning region, as any winning strategy has to operate within this region and can never leave it. Furthermore the winning region can be used to compute a more specific strategy that fulfills desirable properties in a second computation step. Although in some cases this is a desirable approach, it is a very general one and may be sub-optimal. For instance the generality of the most permissive strategy might not be necessary because we know how to find directly and more efficiently a strategy with the desired properties.

In this chapter we present two lazy algorithms for solving two-player safety games that interleave symbolic model checking with the synthesis of candidate solutions. The first algorithm adopts a SAT-based synthesis approach and makes use of a constraint argument which guarantees that any synthesized strategy must respect these constraints. For instance, in case the monolithic system represents a set of processes running in parallel, the constraint can be used to ensure that the synthesized strategy respects the totality of all the system components. Our second algorithm for solving two-player safety games combines a mixed forward/backward search strategy with the symbolic model checker. The forward/backward state space exploration approach provides the algorithm with the capability to synthesize strategies that are *eager* in the sense that they seek to deny progress towards the unsafe states as soon as possible, in contrast to the most permissive strategy. These strategies are desirable in many applications, for instance, in systems that need to be tolerant to hardware faults or perturbations in the environment [36]. The standard backward algorithm may be used to find eager strategies, however it needs first to compute the winning region, and therefore it may spend a lot of time on the exploration of states that could easily be avoided by the system player. Then, in a second step, it can compute an eager strategy and discover that a lot of the explored states are not necessary for the eager solution. To avoid such issues and keep the explored state space small, some sort of forward search from the initial states is required. Unfortunately there is no efficient symbolic algorithm that employs a pure forward search, and most existing techniques that integrate forward search into backwards algorithms do so in a limited manner [63]. Particularly, Brenguier et al. [23] have integrated forward search into an abstraction-based synthesis algorithm, however their experiments revealed that their technique was faster than the standard backwards approach only in few benchmarks.

The two approaches we present here are for monolithic systems, however we have presented in Chapter 5 the parameterized version of the SAT-based approach. Furthermore, we believe that also the forward/backward algorithm we present here can be used for parameterized systems. For this sake, as we have seen in Chapter 5, we can use the backward computation of a parameterized model checker on the counter abstraction of our system. A forward state space exploration for counter systems was also used in Section 2 for computing the so-called successor function.

Outline of the Chapter. We introduce the synthesis problem in Section 6.2 and recapitulate a number of existing approaches to solve it in Section 6.3. In

Section 6.4 we present our SAT-based synthesis algorithm then in Section 6.5 we introduce our lazy synthesis algorithm, followed by a number of optimizations in Section 6.6. The experimental evaluation is presented in Section 6.7, and we discuss further experiences with implementing forward exploration in Section 6.8. In Section 6.9 we discuss connections of our approach to approaches for the synthesis of controllers that are resilient against certain faults, before we conclude in Section 6.10.

6.2 Preliminaries

Given a specification ϕ, the reactive synthesis problem consists in finding a system that satisfies ϕ in an adversarial environment. The problem can be viewed as a game between two players, Player 0 (the system) and Player 1 (the environment), where Player 0 chooses controllable inputs and Player 1 chooses uncontrollable inputs to a given transition function. In this chapter we consider synthesis problems for safety specifications: given a transition system that may raise a BAD flag when entering certain states, we check the existence of a function that reads the current state and the values of uncontrollable inputs, and provides valuations of the controllable inputs such that the BAD flag is not raised on any possible execution. We consider systems where the state space is defined by a set L of Boolean state variables, also called *latches*. We write $\mathbb{B}$ for the set $\{0, 1\}$. A state of the system is a valuation $q \in \mathbb{B}^L$ of the latches. We will represent sets of states by their characteristic functions of type $\mathbb{B}^L \to \mathbb{B}$, and similarly for sets of transitions etc.

A **controllable transition system** (or short: controllable system) TS is a 6-tuple $(L, X_u, X_c, \mathcal{R}, BAD, q_0)$, where:

- L is a set of state variables for the latches
- X_u is a set of uncontrollable input variables
- X_c is a set of controllable input variables
- $\mathcal{R} : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \times \mathbb{B}^{L'} \to \mathbb{B}$ is the transition relation, where $L' = \{l' \mid l \in L\}$ stands for the state variables after the transition
- $BAD : \mathbb{B}^L \to \mathbb{B}$ is the set of unsafe states
- q_0 is the initial state where all latches are initialized to 0.

We assume that the transition relation $\mathcal{R}$ of a controllable system is *deterministic* and *total* in its first three arguments, i.e., for every state $q \in \mathbb{B}^L$, uncontrollable input $u \in \mathbb{B}^{X_u}$ and controllable input $c \in \mathbb{B}^{X_c}$ there exists exactly one state $q' \in \mathbb{B}^{L'}$ such that $(q, u, c, q') \in \mathcal{R}$.

In our setting, characteristic functions are usually applied to a fixed vector of variables. Therefore, if $C : \mathbb{B}^L \to \mathbb{B}$ is a characteristic function, we write C as a short-hand for $C(L)$. Characteristic functions of sets of states can also be applied to next-state variables L', in that case we write C' for $C(L')$.

Let $X = \{x_1, \dots, x_n\}$ be a set of Boolean variables, and $Y \subseteq X \setminus \{x_i\}$ for some x_i. For Boolean functions $F : \mathbb{B}^X \to \mathbb{B}$ and $f_{x_i} : \mathbb{B}^Y \to \mathbb{B}$, we denote by $F[x_i \leftarrow f_{x_i}]$ the Boolean function that substitutes x_i by f_{x_i} in F.

Given a controllable system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$, the *synthesis problem* consists in finding for every $x \in X_c$ a solution function $f_x : \mathbb{B}^L \times \mathbb{B}^{X_u} \rightarrow \mathbb{B}$ such that if we replace $\mathcal{R}$ by $\mathcal{R}[x \leftarrow f_x]_{x \in X_c}$, we obtain a safe system, i.e., no state in BAD is reachable.

If such a solution does not exist, we say the system is *unrealizable.*

A set of solution functions for all $x \in X_c$ is also called a *strategy* (for Player 0). We call the states that are reachable under a given strategy the *care-set* of the strategy. Note that the behavior of the system does not change if the strategy is modified on states outside of the care-set. If BAD is unreachable under a given strategy, we call it a *winning strategy.*

To determine the possible behaviors of a controllable system, two forms of image computation can be used: i) the *image* of a set of states C is the set of states that are reachable from C in one step, and the *preimage* are those states from which C is reachable in one step—in both cases ignoring who controls the input variables; ii) the *uncontrollable preimage* of C is the set of states from which the environment can force the next transition to go into C, regardless of the choice of controllable variables. Formally, we define:

Given a controllable system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$ and a set of states C, we have:

- $image(C) = \{q' \in \mathbb{B}^{L'} \mid \exists (q, u, c) \in \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} : C(q) \wedge \mathcal{R}(q, u, c, q')\}$. We also write this set as $\exists L\ \exists X_u\ \exists X_c\ (C \wedge \mathcal{R})$.

- $preimage(C) = \{q \in \mathbb{B}^L \mid \exists (u, c, q') \in \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \times \mathbb{B}^{L'} : C(q') \wedge \mathcal{R}(q, u, c, q')\}$. We also write this set as $\exists X_u\ \exists X_c\ \exists L'\ (C' \wedge \mathcal{R})$.

- $UPRE(C) = \{q \in \mathbb{B}^L \mid \exists u \in \mathbb{B}^{X_u}\ \forall c \in \mathbb{B}^{X_c}\ \exists q' \in \mathbb{B}^L : C(q') \wedge \mathcal{R}(q, u, c, q')\}$. We also write this set as $\exists X_u\ \forall X_c\ \exists L'\ (C' \wedge \mathcal{R})$.

A direct correspondence of the uncontrollable preimage $UPRE$ for forward computation does not exist: if the environment can force the next transition out of a given set of states, in general the states that we reach are not uniquely determined and depend on the choice of Player 0.

6.2.1 Efficient symbolic computation

BDDs are a suitable data structure for the efficient representation and manipulation of Boolean functions, including all operations needed for the computation of *image*, *preimage*, and $UPRE$. Between these three, *preimage* can be computed most efficiently, while *image* and $UPRE$ are more expensive: there exist a number of optimizations for the computation of *preimage* that cannot be used when computing *image* (see Section 6.6); and $UPRE$ contains a quantifier alternation, which makes it much more expensive than the other two operations.

6.3 Existing Approaches

As mentioned before, the safety synthesis problem is usually seen as a game between Player 1, who chooses the uncontrollable inputs, and Player 0, who chooses the controllable inputs. The goal of Player 0 is to choose the inputs in a way that he never visits an unsafe state. The classical approach to solve such a game is to compute the so-called winning regions of the two players, where the winning region of Player 1 is the set of states from which he can force Player 0 into an unsafe state and the winning region for Player 0 is any state that is not winning for Player 1.

Before we introduce our new approach, we recapitulate three existing approaches and point out their benefits and drawbacks.

6.3.1 Backward fixed-point algorithm

Given a controllable transition system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$ with $BAD(q) \neq 0$ for some $q \in \mathbb{B}^L$, the standard backward BDD-based algorithm (see e.g. [63]) computes the winning region of Player 1, i.e., the set of states from which the environment can force the system into unsafe states, in a fixed-point computation that starts with the unsafe states. The winning region of Player 1 is the least fixed-point of $UPRE$ on $BAD : \mu C.\ UPRE(C') \cup BAD \cup C$.

Since safety games are determined, the complement of the computed set is the winning region for Player 0, i.e., the set of all states from which the system can win the game. Thus, this set also represents the most permissive winning strategy for Player 0. We note two things regarding this approach:

1. To obtain the winning region, it computes the set of all states that cannot avoid moving into an error state, using the rather expensive $UPRE$ operation.

2. The most permissive winning strategy will not avoid progress towards the error states unless we reach the border of the winning region.

6.3.2 A Forward Algorithm [28, 79]

A forward algorithm is presented by Cassez et al. [28] for the dual problem of solving reachability games, based on the work of Liu and Smolka [79]. The algorithm starts from the initial state and explores all states that are reachable in a forward manner. Whenever a state is visited, the algorithm checks whether it is losing; if it is, the algorithm revisits all reachable states that have a transition to this state and checks if they can avoid moving to a losing state. Although the algorithm is optimal in that it has linear time complexity in the state space, two issues should be taken into account:

1. The algorithm explicitly enumerates states and transitions, which is impractical even for moderate-size systems.

2. A fully symbolic implementation of the algorithm does not exist, and it would have to rely heavily on the expensive forward *image* computation.

We will discuss the difficulties of implementing a symbolic forward algorithm in more detail in Section 6.8.1.

6.3.3 Lazy Synthesis [55]

Lazy Synthesis interleaves a backwards model checking algorithm that identifies possible error paths with the synthesis of candidate solutions. To this end,

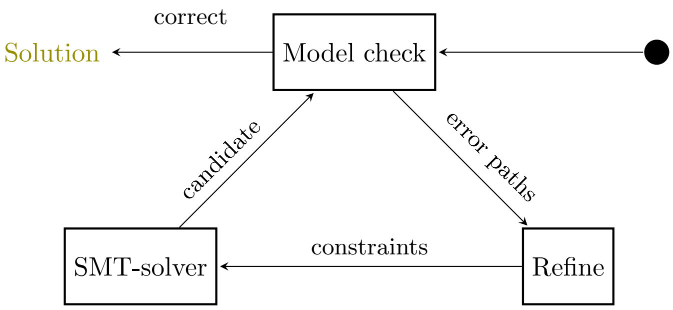


Figure 6.1: High-level description of the lazy synthesis algorithm

the error paths are encoded into a set of constraints, and an SMT solver produces a candidate solution that avoids all known errors. If new error paths are discovered, more constraints are added that exclude them. The procedure terminates once a correct candidate is found (see Figure 6.1). The approach works in a more general setting than ours, for systems with multiple components and partial information. When applied to our setting and challenging benchmark problems, the following issues arise:

1. Even though the error paths are encoded as constraints, the representation is such that it explicitly branches over valuations of all input variables, for each step of the error paths. This is clearly impractical for systems that have more than a dozen input variables (which is frequently the case in the classes of problems we target).

2. In each iteration of the main loop a single deterministic candidate is checked. Therefore, many iterations may be needed to discover all error paths.

6.4 SAT-Based Lazy Safety Synthesis

Before introducing our symbolic lazy algorithm we will show how we can adopt the general synthesis algorithm presented in [55] which is not restricted to safety synthesis. However our version of the algorithm makes use of a Boolean constraints formula that must be provided by a user. Algorithm 4 takes as input a controllable system $sys = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$, a transition relation constraint $TrConstr$, and it returns either *Unrealizable*, i.e., the system can not be fixed, or a new transition relation that is safe and total.

Transition relation constraint. $TrConstr$ is a Boolean formula over transitions in $\mathcal{R}$. It allows to add constraints to the solution requested from the SAT solver. An important scenario in which $TrConstr$ is fundamental for the correctness of the candidate solution is the synthesis of a concurrent system. Suppose that the system we seek to synthesize is a monolithic representation of a concurrent system, then in such a case, the candidate solution must ensure the totality of each component of the system. For that sake, we can encode the totality of each process inside $TrConstr$ and pass it as a parameter to the synthesizer.

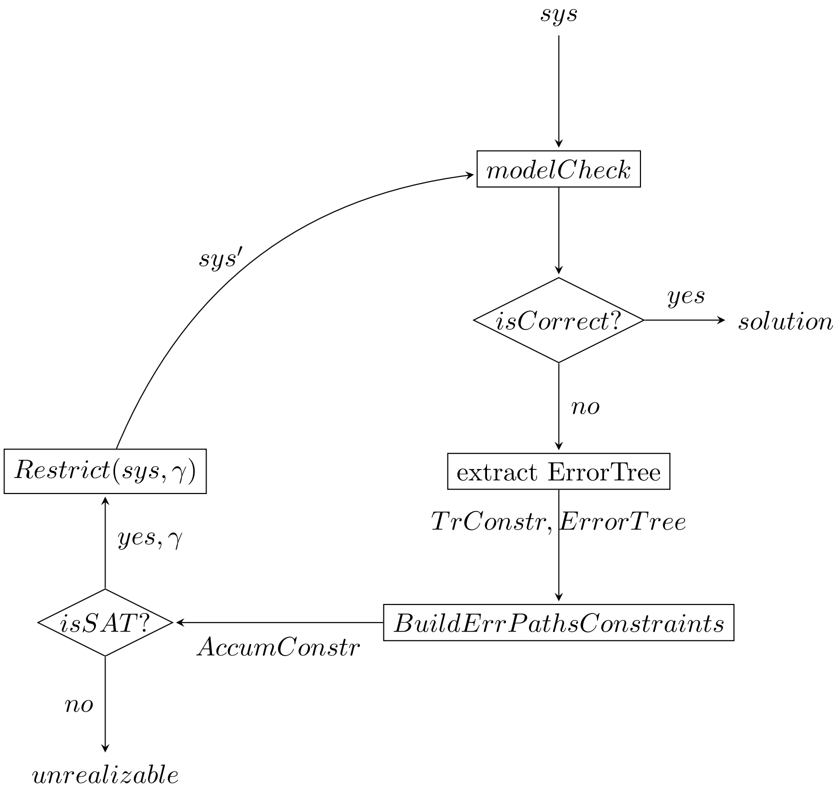


Figure 6.2: Control flow of the algorithm

Overview.

Figure 6.2 sketches the basic idea of Algorithm 4. It starts by model checking the controllable system sys, without any restriction on the transition relation wrt. the controllable inputs. If unsafe states are reachable, the model checker returns an error tree that contains the error paths found so far. Out of this error tree, the transition relation constraint $TrConstr$, and the constraints obtained in previous iterations, a new constraint formula $AccumConstr$ is constructed and sent to a SAT solver. If there is no solution for the computed constraints then the problem is unrealizable. Otherwise a new candidate solution sys' is produced by restricting sys with the SAT solver solution γ. Then the whole loop restarts with sys' as the input system.

6.4.1 Algorithm 4 Description

SAT-LazySynthesis. Lines 2,3 initialize the variables TR, and $AccumConstr$. $AccumConstr$ will be used to accumulate constraints obtained in each iteration of the algorithm. In Line 5 we call the model checker ModelCheckForSAT. We assume that the model checker returns either *true*, i.e., the system is safe or an error tree *errTree*. The error tree is a tuple $(Q_T, q_{T_0}, E, Leaves)$ where

- $Q_T \subseteq \mathbb{B}^L$ is the set of tree nodes.

Algorithm 4 Non-symbolic SAT Based Lazy Synthesis

```
1:  procedure SAT-LazySynthesis($ControllableSystem\ sys$,$TrConstr$)
2:      $TR \leftarrow sys.\mathcal{R}$
3:      $AccumConstr \leftarrow TrConstr$
4:      while true do
5:          $isCorrect, errTree \leftarrow$ ModelCheckforSAT($TR, sys.BAD$)
6:          if $isCorrect$ then
7:              return $TR$
8:          $ErrPathsConstr \leftarrow BuildErrPathsConstr(errTree, errTree.q_{T_0})$
9:          $TotalConstr \leftarrow \bigwedge_{q \in errTree.Q_T} \bigvee_{(q,u,c,q') \in sys.\mathcal{R}} (q, u, c, q')$
10:         $AccumConstr \leftarrow ErrPathsConstr \wedge AccumConstr$
11:         $isSAT, \gamma \leftarrow SAT(AccumConstr \wedge TotalConstr \wedge TrConstr)$
12:         if not $isSAT$ then
13:             return $Unrealizable$
14:         $TR \leftarrow Restrict(sys.\mathcal{R}, \gamma)$

1:  procedure BuildErrPathsConstr($errTree$,$q$)
2:      if $q \in errTree.Leaves$ then
3:          return $True$
4:      return $\bigwedge_{(q,u,c,q') \in errTree.E} (\neg(q, u, c, q') \vee$
    $BuildErrPathsConstr(errTree, q'))$
```

- $q_{T_0} \in Q_T$ is the initial node.

- $E \subseteq \mathcal{R}$ is the set of edges.

- $Leaves \subseteq Q_T$ is the set of leaf nodes, and $\forall q \in Leaves : BAD(q) = 1$. Informally all leaf nodes are unsafe states of the system.

Line 7 returns *true* if TR is safe. Otherwise, Line 8 constructs the Boolean constraint $ErrPathsConstr$ that ensures that all error paths found in the error tree will be avoided in future iterations. The Boolean constraint $TotalConstr$ in Line 9 ensures that the solution returned by the SAT solver has at least one transition for every uncontrollable input. In Line 10 $ErrPathsConstr$ is added to $AccumConstr$. Line 12 calls a SAT solver to check if there exists a solution for $AccumConstr$ that respects $TotalConstr$ and $TrConstr$. If the constraints are not satisfiable, Line 13 returns the string *Unrealizable* and the algorithm terminates. Otherwise, in Line 14 the transition relation $\mathcal{R}$ is updated according to the solution returned by the SAT solver, and then another iteration of the loop starts.
BuildErrPathsConstr. This recursive procedure, given an error tree and an initial node, returns a propositional formula over the set of transitions that encodes all possible ways to avoid reaching an unsafe state. Line 3 is the base case of the recursion when a leaf node is reached. Line 4 asserts that an edge of a node must be either deleted or we need to ensure that the path of the edge's target state to leaf nodes is broken.

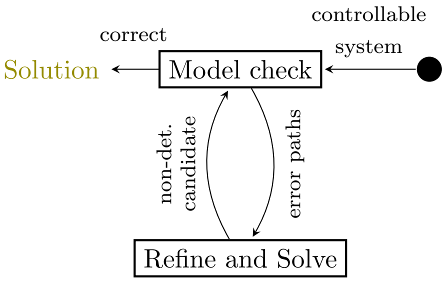


Figure 6.3: High-level description of the symbolic lazy synthesis algorithm

6.5 Symbolic Lazy Synthesis Algorithms

In the following, we present symbolic algorithms that are inspired by the lazy synthesis approach and overcome some of its weaknesses to make it suitable for challenging benchmark problems like those from the library of the reactive synthesis competition SYNTCOMP [62]. We show that in our setting, we can avoid the explicit enumeration of error paths. Furthermore, we can use non-deterministic candidate models that are restricted such that they avoid the known error paths. When choosing these restrictions, we prioritize the removal of transitions that are close to the initial state, which can help us avoid error paths that are not known yet. The high-level control flow of the algorithm is depicted in Figure 6.3.

6.5.1 The basic algorithm

To explain the algorithm, we need some additional definitions. Fix a controllable system $TS = (L, X_u, X_c, \mathcal{R}, BAD, q_0)$.

An *error level* E_i is a set of states that are on a path from q_0 to BAD, and all states in E_i are reachable from q_0 in i steps. Formally, E_i is a subset of

$$\left\{ q_i \in \mathbb{B}^L \;\middle|\; \begin{array}{l} \exists q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n \in \mathbb{B}^L : \\ q_n \in BAD \text{ and } \exists (q_j, u, c, q_{j+1}) \in \mathcal{R} \text{ for } 0 \leq j < n \end{array} \right\}.$$

We call $(E_0, ..., E_n)$ a *sequence of error levels* if i) each E_i is an error level, ii) each state in each E_i has a transition to a state in E_{i+1}, and iii) $E_n \subseteq BAD$. Note that the same state can appear in multiple error levels of a sequence, and E_0 contains only q_0.

Given a sequence of error levels $(E_0, ..., E_n)$, an *escape* for a transition (q, u, c, q') with $q \in E_i$ and $q' \in E_{i+1}$ is a transition (q, u, c', q'') such that $\forall m > i : q'' \notin E_m$. We say the transition (q, u, c, q') *matches* the escape (q, u, c', q'').

Given two error levels E_i and E_{i+1}, we denote by RT_i the following set of tuples, representing the "removable" transitions, i.e., all transitions from E_i to E_{i+1} that match an escape:

$$RT_i = \{(q, u, q') \mid q \in E_i, q' \in E_{i+1} \text{ and } \exists (q, u, c, q') \in \mathcal{R} \text{ that has an escape}\}.$$

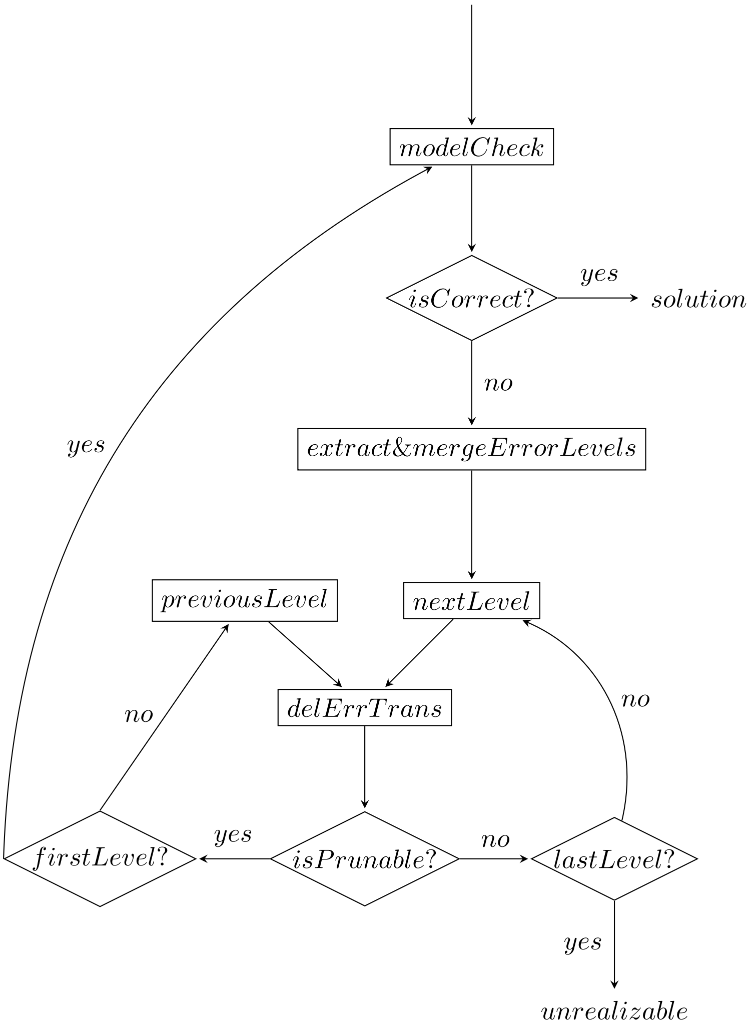


Figure 6.4: Control flow of the algorithm

Overview

Figure 6.4 sketches the control flow of the algorithm where all operations are performed symbolically on sets of states. It starts by model checking the controllable system, without any restriction on the transition relation wrt. the controllable inputs. If unsafe states are reachable, the model checker returns a sequence of error levels. Iterating over all levels, we identify the transitions from the current level for which there exists an escape, and temporarily remove them from the transition relation. Based on the new restrictions on the transition relation, the algorithm then prunes the current error level by removing states that do not have transitions to the next level anymore. Whenever we prune at least one state, we move to the previous level to propagate back this information. If this eventually allows us to prune the first level, i.e., remove the initial state, then this error sequence has been invalidated and the new transition system (with deleted transitions) is sent to the model checker. Otherwise the system is unrealizable. In any following iteration, we accumulate information by merging the new error sequence with the ones we found before, and reset the transition relation before we analyze the error sequence for escapes.

Detailed Description

In more detail, Algorithm 5 describes a symbolic lazy synthesis algorithm. The method takes as input a controllable system and checks if its transition relation can be fixed in a way that error states are avoided. Upon termination, the algorithm returns either *unrealizable*, i.e., the system can not be fixed, or a restricted transition relation that is safe and total. From such a transition relation, a (deterministic) solution for the synthesis problem can be extracted in the same way as for existing algorithms. Therefore, we restrict the description of our algorithm to the computation of the safe transition relation.

LazySynthesis: In Line 2, we initialize TR to the unrestricted transition relation $\mathcal{R}$ of the input system and E to the empty sequence, before we enter the main loop. Line 4 uses a model checker to check if the current TR is correct, and returns a sequence of error levels $mcLvls$ if it is not. In more detail, procedure ModelCheck(TR) starts from the set of error states and uses the *preimage* function (see Section 6.2) to iteratively compute a sequence of error levels.[1] It terminates if a level contains the initial state or if it reaches a fixed point. If the initial state was reached, the model checker uses the *image* function to remove from the error levels any state that is not reachable from the initial state.[2] Otherwise, in Line 6 we return the safe transition relation. If TR is not safe yet, Line 7 merges the new error levels with the error levels obtained in previous iterations by letting $E[i] \leftarrow E[i] \vee mcLvls[i]$ for every i. In Line 8 we call PruneLevels($sys.\mathcal{R}, E$), which searches for a transition relation that avoids all error paths represented in E, as explained below. If pruning is not successful, in Lines 9-10 we return "*Unrealizable*".

PruneLevels: In the first loop, ResolveLevel(E, i, TR) is called for increasing values of i (Line 4). Resolving a level is explained in detail below; roughly it means that we remove transitions that match an escape, and then remove states from this level that are not on an error path anymore. If ResolveLevel has removed states from the current level, indicated by the return value of *isPrunable*, we check whether we are at the topmost level — if this is the case, we have removed the initial state from the level, which means that we have shown that every path from the initial state along the error sequence can be avoided. If we are not at the topmost level, we decrement i before returning to the start of the loop, in order to propagate the information about removed states to the previous level(s). If *isPrunable* is false, we instead increment i and continue on the next level of the error sequence.

The first loop terminates either in Line 7, or if we reach the last level. In the latter case, we were not able to remove the initial state from $E[0]$ with the local propagation of information during the main loop (that stops if we reach a

[1] This part is the light-weight backward search: unlike $UPRE$ in the standard backward algorithm, *preimage* does not contain any quantifier alternation.

[2] This is the only place where our algorithm uses *image*, and it is only included to keep the definitions and correctness argument simple - the algorithm also works if the model checker omits this last *image* computation step, see Section 6.6.

Algorithm 5 Lazy Synthesis

```
procedure LazySynthesis(ControllableSystem sys)
    $TR \leftarrow sys.\mathcal{R}, \quad E \leftarrow ()$
    while true do
        $isCorrect, mcLvls \leftarrow$ ModelCheck($TR, sys.BAD$)
        if $isCorrect$ then
            return $TR$
        $E \leftarrow$ mergeLevels($E, mcLvls$)
        $isUnrealizable, TR \leftarrow$ PruneLevels($sys.\mathcal{R}, E$)
        if $isUnrealizable$ then
            return $Unrealizable$
procedure PruneLevels(TransitionRelation TR, ErrorSequence E)
    $i \leftarrow 0$
    while $i < length(E) - 1$ do
        $isPrunable, TR, E \leftarrow$ ResolveLevel($E, i, TR$)
        if $isPrunable$ then
            if $i == 0$ then   // we have removed the initial state from $E[0]$
                return false, TR
            $i \leftarrow i - 1$
        else
            $i \leftarrow i + 1$
    while $i \geq 1$ do   // $i == length(E) - 1$ when we enter the loop
        $i \leftarrow i - 1$
        $isPrunable, TR, E \leftarrow$ ResolveLevel($E, i, TR$)
    if $isPrunable$ then   // we have removed the initial state from $E[0]$
        return false, TR
    else   // we could not remove the initial state from $E[0]$
        return true, $\emptyset$
procedure ResolveLevel(ErrorSequence E, Int i, TransitionRelation
  TR)
    $RT \leftarrow (\exists L' \; (( \; \exists X_c \; TR \;) \wedge \neg E[i+1:n]' \;)) \; \wedge E[i] \wedge E[i+1]'$
    $TR \leftarrow TR \wedge \neg RT$
    $AVSet \leftarrow \forall X_u \; (E[i] \; \wedge \exists L'( \; \exists X_c \; TR \wedge \neg E[i+1:n]' \;) \;)$
    $E[i] \leftarrow E[i] \wedge \neg AVSet$
    return $AVSet \neq \emptyset, TR, E$
```

level that cannot be pruned). To make sure that all information is completely propagated, afterwards we start another loop were we resolve all levels bottom-up, propagating the information about removed states all the way to the top. If we arrive at $E[0]$ and still cannot remove the initial state, we conclude that the system is unrealizable. This last propagation is needed because, unlike previous propagations, it propagates all information up lo level $E[0]$ even if some error level is not prunable. To see why this is necessary, consider an error sequence obtained after merging error sequences from different iterations, where a state q can be in more than one error level at the same time, say in levels i and j with $i < j$. Now if some error level between i and j is not prunable, then level i will not be resolved again, and escapes for transitions from q will not be used to prune level i, even if they are used to prune level j.

ResolveLevel: Line 2 computes the set of transitions that have an escape: $\exists L'\ ((\ \exists X_c\ TR\ ) \wedge \neg E[i+1:n]'\ )$ is the set of all (q,u) for which there exists an escape (q,u,c,q'), and by conjoining this set with $E[i] \wedge E[i+1]'$ we compute all tuples (q,u,q') that represent transitions from $E[i]$ to $E[i+1]$ matching an escape. Line 3 removes the corresponding transitions from the transition relation TR. Line 4 computes $AVSet$ which represents the set of all states such that all their transitions within the error levels match an escape. $\forall X_u$ $(E[i]\ \wedge \exists L'(\ \exists X_c\ TR \wedge \neg E[i+1:n]'\ )\ )$ returns the set of states that have an escape for every uncontrollable input. After removing $AVSet$ from the current level, we return.

Illustration of the Algorithm

As an example, Figure 6.5 shows error levels that may be obtained from the model checker in a first iteration. The transitions are labeled with vectors of input bits, where the left bit is uncontrollable and the right bit controllable. The last level is a subset of BAD. After the first iteration of the algorithm, the transitions that are dashed in Figure 6.6 will be deleted. Note that another solution exists where instead we delete the two outgoing transitions from level E_1 to the error level Err. This solution can be obtained by a backward algorithm. However, our solution makes all states in E_1 unreachable and thus has a care-set that is much smaller than the winning region.

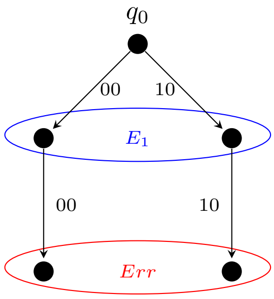


Figure 6.5: Error levels from iteration 1

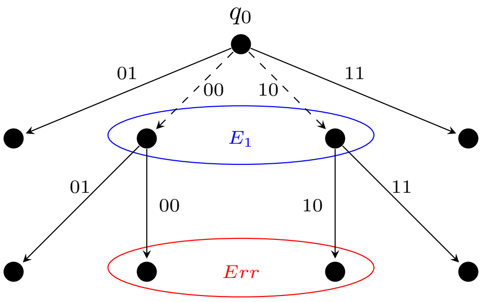


Figure 6.6: solution for iteration 1

Figure 6.7 depicts merged error levels obtained from iteration 1 and 2 where you can see that the initial state *init* cannot avoid the error level E_1 on uncontrollable input 0. Figure 6.8 shows that a state can be pruned from level E_1 as it state can avoid level E_2. Pruning E_1 allows *init* to find an escape for uncontrollable input 0 and as a consequence it can avoid E_1 completely.

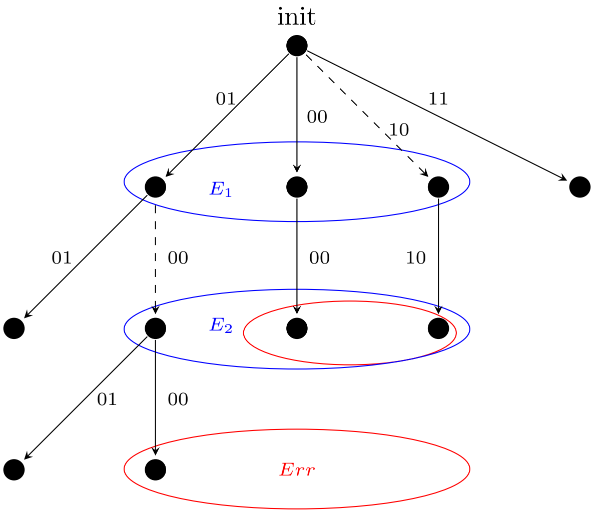


Figure 6.7: Error levels from iteration 2

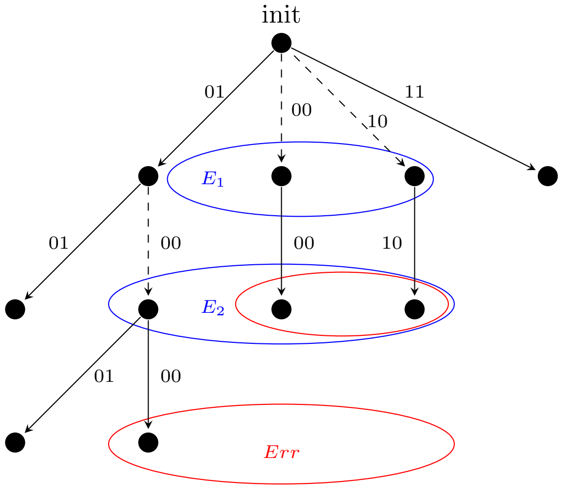


Figure 6.8: Solution for iteration 2

Comparison

Having defined our symbolic lazy synthesis algorithm formally, let us again compare it to the existing lazy synthesis algorithm, as well as to the standard backwards algorithm.

Lazy Synthesis: The approach depicted in Figure 6.1 uses model checking to obtain information on paths to the error states, just like our new approach. However, in contrast to our approach the error paths are encoded into SMT constraints, and based on these constraints the SMT solver chooses a deterministic strategy that avoids all known error paths. Thus, the two essential differences are:

1. The SMT encoding explicitly branches over all possible decisions in the error paths, making it impractical to encode long error paths due to the exponential growth of the encoding.

2. The candidate generated by the SMT solver is deterministic, in contrast to the *non-deterministic* strategy generated by the symbolic lazy algorithm, where the strategy is only determinized after being found correct by the model checker.

To evaluate the impact of the second point, we have implemented a version of the algorithm where the strategy is determinized before being sent to the model checker. As expected, it can only solve a few small instances from the challenging SYNTCOMP benchmark set, and the approach with non-deterministic strategies performs much better. We did not put any of these results in the experimental evaluation section as the obtained results were not interesting.

Standard Backwards Algorithm: Compared to the standard backward fixed-point approach (see Section 6.3.1), an important difference is that we explore the error paths in a forward analysis starting from the initial state, and avoid progress towards the error states as soon as possible. As a consequence, our algorithm can find strategies with a care-set that is much smaller than the winning region, and may solve the problem faster than the standard approach. We give a detailed comparison of the performance of our algorithm against the standard algorithm in Section 6.7.

6.5.2 Correctness of Algorithm 5

Theorem 20 (Soundness)**.** *Every transition relation returned by Algorithm 5 is safe, and total in the first two arguments.*

Proof. The model checker guarantees that the transition relation is safe, i.e., unsafe states are not reachable. To see that the returned transition relation is total in the first two arguments, i.e., $\forall q \in \mathbb{B}^L\ \forall u \in \mathbb{B}_u^X\ \exists c \in \mathbb{B}_c^X\ \exists q' \in \mathbb{B}^{L'} : (q, u, c, q') \in TR$, observe that this property holds for the initial TR, and is preserved by *ResolveLevels*: Lines 2 and 3 of the procedure ensure that a transition $(q, u, c, q') \in TR$ can only be deleted if $\exists c' \in \mathbb{B}_c^X\ \exists q'' \neq q' \in \mathbb{B}^{L'} : (q, u, c', q'') \in TR$, i.e., if there exists another transition with the same state q and uncontrollable input u. □

To prove completeness of the algorithm, we define formally what it means for an error level to be resolved.

Definition 1. *Given a sequence of error levels $E = (E_0, ..., E_n)$ and a transition relation TR, an error level E_i with $i < n$ is* resolved *with respect to TR if the following conditions hold:*

- $RT_i = \emptyset$
- $\forall q_i \in E_i \setminus BAD: \ \exists u \in \mathbb{B}^{X_u} \ \exists c \in \mathbb{B}^{X_c} \ \exists q_{i+1} \in E_{i+1}: \ (q_i, u, c, q_{i+1}) \in TR$

E_i is unresolved otherwise, and E_n is always resolved.

Informally, E_i is resolved if every state in E_i, on some uncontrollable input u, cannot avoid reaching lower levels (i.e. each controllable input of u leads to some E_j where $i < j \leq n$). We can conclude the following lemma.

Lemma 30. *A controllable system is unrealizable iff there exists an error sequence $E_0, E_1, ..., E_n$ where $E_0 = \{q_0\}$, and for all $i \leq n$, E_i is resolved and non-empty.*

Proof. Suppose the system is unrealizable, i.e., Player 1 has a strategy to always reach BAD. Then for some $n \in \mathbb{N}$ there exists a sequence of (non-empty) sets of states $E_0, E_1, \ldots, E_n$ such that $E_0 = \{q_0\}$, $E_n \subseteq BAD$, and for every E_i and every $q \in E_i$, Player 1 can force the game into E_{i+1} in one step, i.e., $\forall q \in E_i \ \forall c \in \mathbb{B}^{X_c} \ \exists u \in \mathbb{B}^{X_u} : (q, u, c, q') \in TR$ with $q' \in E_{i+1}$. In particular, $E_0, E_1, \ldots, E_n$ is an error sequence. To see that it is resolved, assume that it was not: then from some E_i, RT_i would have to be non-empty, i.e., for some $q \in E_i$and $u \in \mathbb{B}^{X_u}$ there would have to be a transition $(q, u, c, q') \in TR$ with $q' \notin E_{i+1}$, contradicting the properties of our error sequence.

In the other direction, suppose there exists an error sequence $E_0, E_1, ..., E_n)$ with $E_0 = \{q_0\}$ and $\forall i \leq n$, E_i is resolved and non-empty. Then we can construct a strategy for Player 1 to win the game: in each E_i, there must exist a state q and inputs u, c such that there is $(q, u, c, q') \in TR$ with $q' \in E_{i+1}$, for which there is no escape. A winning strategy for Player 1 is to always choose such an uncontrollable input u. □

Theorem 21 (Completeness)**.** *If Algorithm 5 returns "Unrealizable", then the controllable system is unrealizable.*

Proof. Observe that the algorithm returns unrealizable only when there exists an error sequence $E_0, E_1, ..., E_n$ where $E_0 = \{q_0\}$ and all levels are resolved and non-empty. Lines 2 and 3 of ResolveLevel guarantee that all transitions from E_i to E_{i+1} that match an escape will be deleted, so the only remaining transitions between E_i and E_{i+1} are those that have no escapes. Line 4 computes all states in E_i that no longer have transitions to lower levels (levels with greater index) and Line 5 removes these states. Thus, after calling ResolveLevel, the current level will be resolved.

However, since ResolveLevel may remove states from E_i, the levels E_j with $j < i$ could become unresolved. To see that this is not an issue note that before we output *Unrealizable*, we go through the second loop that resolves all levels from n to 0. After execution of this second loop all levels are resolved, and if E_0 still contains q_0, then from our sequence of error levels we can extract a subsequence[3] of resolved and non-empty error levels, which by Lemma 30 implies unrealizability. □

Theorem 22 (Termination)**.** *Algorithm 5 always terminates.*

[3] It may be a subsequence due to the merging of error levels from different iterations of the main loop.

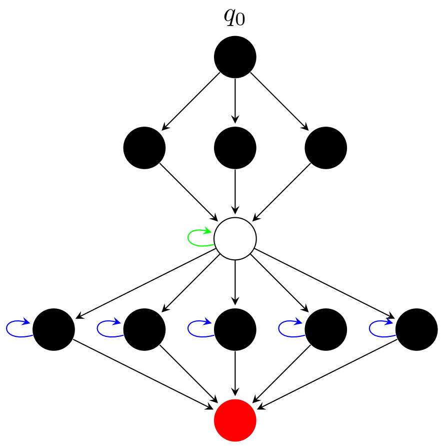


Figure 6.9: Example with small solution

Proof. Every call of the procedure PruneLevels returns a transition relation that is guaranteed to avoid all error paths returned by the model checker in all previous iterations (see Line 7 of procedure LazySynthesis). This is accomplished by making at least one state on every path from the initial state to an error state unreachable (see Lines 6-7,14-15 of PruneLevels). In particular, any transition relation returned by PruneLevels is different from all previous transition relations. Since for a fixed controllable system there is only a finite number of possible transition relations, the procedure will eventually terminate. □

6.5.3 Example Problems

We want to highlight the potential benefit of our algorithm on two families of examples.

First, consider a controllable system where all paths from the initial state to the error states have to go through a bottleneck, e.g., a single state, as depicted in Figure 6.9, and assume that Player 0 can force the system not to go beyond this bottleneck. In this case, the care-set of our solution only includes the states between the initial state and the bottleneck, whereas the winning region detected by the standard algorithm may be much bigger (in the example including all the states in the fourth row). Moreover, the strategy produced by our algorithm will be very simple: if we reach the bottleneck, we force the system to stay there. In contrast, the strategy produced by the standard algorithm will in general be much more complicated, as it has to define the behavior for a much larger number of states.

Second, consider a controllable system where the shortest path between error and initial state is short, but Player 1 can only *force* the system to move towards the error on a long path. Moreover, assume that Player 0 can avoid entering this long path, for example by entering a separate part of the state space like depicted in Figure 6.10. In this case, our algorithm will quickly find a simple solution: move to that separate part and stay there. In contrast, the standard algorithm will have to go through many iterations of the backwards fixed-point

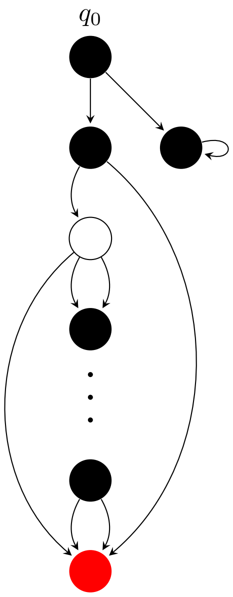


Figure 6.10: Example that is solved fast

computation, until finally finding the point where moving into the losing region can be avoided.

6.6 Optimization

As presented, Algorithm 5 requires the construction of a data structure that represents the full transition relation $\mathcal{R}$, which causes a significant memory consumption. In practice, the size of a BDD that represents the full transition relation can be prohibitive even for moderate-size models.

Since the transition relation is deterministic, it can alternatively be represented by a vector of functions, each of which updates one of the state variables. Such a partitioning of the transition relation is an additional computational effort, but it results in a more efficient representation that is necessary to handle large systems. In the following we describe optimizations based on such a representation.

Definition 2. *A* ***functional controllable system*** *is a 6-tuple* $TS_f = (L, X_u, X_c, \vec{F}, BAD, q_0)$, *where*

- L *is a set of state variables for the latches*
- X_u *is a set of uncontrollable input variables*
- X_c *is a set of controllable input variables*
- $\vec{F} = (f_1, ..., f_{|L|})$ *is a vector of update functions* $f_i : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \to \mathbb{B}$ *for* $i \in \{1, \ldots, |L|\}$
- $BAD : \mathbb{B}^L \to \mathbb{B}$ *is the set of unsafe states*

- q_0 *is the initial state where all latches are initialized to 0.*

In a functional controllable system with current state q and inputs u and c, the next-state value of the ith state variable l_i is computed as $f_i(q, u, c)$. Thus, we can compute image and preimage of a set of states C in the following way:

- $image_f(C) = \exists L \; \exists X_u \; \exists X_c \; (\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C)$
- $preimage_f(C) = \exists L' \; \exists X_u \; \exists X_c \; (\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C')$

However, computing $\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C'$ is still very expensive and might be as hard as computing the whole transition relation. To optimize the preimage computation, we instead directly substitute the state variables in the Boolean function that represents C by the function that computes their new value:

$$preimage_s(C) = \exists X_u \; \exists X_c \; C[l_i \leftarrow f_i]_{l_i \in L}$$

Since substitution cannot be used to compute $image(C)$, forward exploration of the state space is in practice much more expensive than backwards exploration. This even holds for alternative, more efficient ways to compute *image*, such as using the *range* function [75]. We consider a forward algorithm based on this alternative in Section 6.8.1.

6.6.1 The Optimized Algorithm

The optimized algorithm takes as input a functional controllable system, and uses the following modified procedures:

OPTIMIZEDLAZYSYNTHESIS: This procedure replaces LAZYSYNTHESIS, to which it is different in two aspects concerning the model checker:

1. the preimage is computed using $preimage_s$, and
2. unreachable states are not removed, in order to avoid image computation. Thus, the error levels are over-approximated.

OPTIMIZEDRESOLVELEVEL: This procedure replaces RESOLVELEVEL and computes RT and $AvSet$ more efficiently. Note that for a given set of states C, the set $pretrans(C) = \{(q, u, c) \in \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \mid \vec{F}(q, u, c) \in C\}$ can efficiently be computed as $C[l_i \leftarrow f_i]_{l_i \in L}$. Based on this, we get the following:

- RT: we compute the transitions that can be avoided as the conjunction of the transitions from E_i to E_{i+1}, given as $pretrans(E[i+1]') \wedge E[i]$, with those transitions that have an escape, $\exists X_c \; pretrans(\neg E[i+1:n]') \wedge E[i]$.
- $AvSet$: The states that can avoid all transitions to the lower levels can now be computed as $\forall X_u \; [\; \exists X_c \; pretrans(\neg E[i+1:n]') \wedge E[i] \;]$.

Generalized Deletion of Transitions

In addition, we consider a variant of our algorithm that uses the following heuristic to speed up computation: whenever we find an escape (q, u, c, q') with $q \in E_i$, then we not only remove all matching transitions that start in E_i, but matching transitions that start anywhere, and lead to a state $q'' \in E_j$ with $j > i$. Thus, we delete more transitions per iteration of the algorithm, all of which are known to lead to an error.

6.7 Experimental Evaluation

We implemented Algorithm 5 in Python, using the BDD package CUDD [91]. We evaluate our prototype on a family of parameterized benchmarks based on the examples in Section 6.5.3, on the benchmark set of SYNTCOMP 2017 [62], and on a set of random benchmarks.

We evaluate two versions of Algorithm 5: a version without generalized deletion (see Section 6.6.1), in the following called *Lazy*, and a version with generalized deletion, in the following called $Lazy_{GD}$. We compare them against our own implementation of the standard backward approach, in order to have a fair comparison between algorithms that use the same BDD library and programming language. For the SYNTCOMP benchmarks, we additionally compare against the results of the participants in SYNTCOMP 2017. Our implementations of all algorithms include the most important general optimizations for this kind of algorithms, including a functional transition relation and automatic reordering of BDDs (see Jacobs et al. [63]).

Note that we did not add here the experimental results of Algorithm 4. This is due to the fact that it includes the function *BuildErrPathsConstr* which explore the error paths in an explicit manner. therefore the algorithm was able only to solve toy benchmarks of SYNTCOMP 2017.

6.7.1 Parameterized Benchmarks

On the parameterized versions of the examples from Section 6.5.3, we observe the expected behaviour:

- for the first example, the care-set of the strategy found by our algorithm is always only about half as big as the winning region found by the standard algorithm. Even more notable is the size of the synthesized controller circuit: for example, our solution for an instance with 2^{18} states and 10 input variables has a size of just 9 AND-gates, whereas the solution obtained from the standard algorithm has 800 AND-gates.

- for the second example, we observe that for systems with 15 to 25 state variables, our algorithm solves the problem in constant time of 0.1s, whereas the solving time increases sharply for the standard algorithm: it uses 1.7s for a system with 15 latches, 92s for 20 latches, and 4194s for 25 latches.

6.7.2 SYNTCOMP Benchmarks

We compared our algorithms against the standard algorithm on the benchmark set that was used in the safety track of SYNTCOMP 2017, with a timeout of 5000s on an Intel Xeon processor (E3-1271 v3, 3.6 GHz) and 32 GB RAM.

First, we observe that our algorithms often detect care-sets that are much smaller than the full winning region: out of the 76 realizable benchmarks that the *Lazy* algorithm solved, we found a strictly smaller care-set in 28 cases. In 14 cases, the care-set is smaller by a factor of 10^3 or more, in 8 cases by a factor of 10^{20} or more, and in 4 cases by a factor of 10^{30} or more. The biggest difference in size is by a factor of 10^{68}. For the $Lazy_{GD}$ algorithm, the care-sets are somewhat bigger, but the tendency is the same. Table 6.1 gives detailed information for a selection of such examples.

Table 6.1: Comparison of care-set and winning region size for selected benchmarks
(number of states)

Instance	**Standard**	**Lazy**	**Lazy$_{\mathbf{GD}}$**	**Difference factor**
load_2c_comp_comp5	$1.08 * 10^{40}$	$\mathbf{5.67 * 10^{13}}$	$3.79 * 10^{22}$	$> 10^{26}$
load_3c_comp_comp4	$2.39 * 10^{52}$	$\mathbf{1.21 * 10^{18}}$	$8.5 * 10^{37}$	$> 10^{44}$
load_3c_comp_comp7	$4.97 * 10^{86}$	$\mathbf{1.21 * 10^{18}}$	$6.28 * 10^{57}$	$> 10^{68}$
load_4c_comp_comp4	$4.03 * 10^{63}$	*TO*	$\mathbf{4.79 * 10^{52}}$	$> 10^{10}$
load_4c_comp_comp6	$9.03 * 10^{92}$	*TO*	$\mathbf{2.2 * 10^{71}}$	$> 10^{21}$
load_full_2_comp5	$2.52 * 10^{80}$	*TO*	$\mathbf{4.21 * 10^{65}}$	$> 10^{15}$
load_full_2_comp7	$4.99 * 10^{108}$	*TO*	$\mathbf{3.11 * 10^{85}}$	$> 10^{23}$
ltl2dba_C2-6_comp3	$2.46 * 10^{35}$	$\mathbf{4.55 * 10^{25}}$	$\mathbf{4.55 * 10^{25}}$	$> 10^{9}$
ltl2dba_E4_comp3	$2.96 * 10^{79}$	$\mathbf{3.74 * 10^{50}}$	$1.05 * 10^{65}$	$> 10^{28}$
demo-v10_5	$1.93 * 10^{25}$	$\mathbf{1.31} * 10^{5}$	$2.25 * 10^{15}$	$> 10^{20}$
demo-v12_5	$2.81 * 10^{14}$	$\mathbf{1.64} * 10^{4}$	$6.98 * 10^{10}$	$> 10^{10}$
demo-v14_5	$1.23 * 10^{14}$	**356**	$2.69 * 10^{8}$	$> 10^{11}$
demo-v16_5	$9.03 * 10^{7}$	$\mathbf{1.36 * 10^{4}}$	$3.09 * 10^{5}$	$> 10^{3}$
demo-v18_5	$3.67 * 10^{27}$	*TO*	$\mathbf{6.97 * 10^{16}}$	$> 10^{10}$
demo-v19_5	$1.27 * 10^{11}$	**305**	$2.68 * 10^{8}$	$> 10^{8}$
demo-v20_5	$2.31 * 10^{41}$	$\mathbf{3.44 * 10^{10}}$	$1.22 * 10^{24}$	$> 10^{30}$
demo-v22_5	$3.4 * 10^{38}$	$\mathbf{1.71 * 10^{15}}$	$4.76 * 10^{21}$	$> 10^{23}$
demo-v23_5	$1.37 * 10^{12}$	$\mathbf{9.22 * 10^{3}}$	$1.09 * 10^{9}$	$> 10^{8}$
demo-v24_5	$3.27 * 10^{63}$	$\mathbf{1.17 * 10^{31}}$	$4.23 * 10^{28}$	$> 10^{32}$

However, note that our smaller sets do not necessarily correspond to smaller symbolic representations of these sets. Table 6.2 compares the sizes of BDDs instead of explicit number of states, showing that in some cases the BDD is smaller, but more often the symbolic representation for the smaller set of states is actually more complex. The results are also mixed when regarding the size of the synthesized circuits: in 11 cases the *Lazy* algorithm produces a smaller solution than the standard algorithm, in 21 cases it is the other way around. The *Lazy$_{GD}$* algorithm produced smaller circuits in 15 cases. Tables 6.3,6.3 contain a sample of these results, including also the size of the symbolic representation of the winning strategy. It is also important to note that the *Lazy* algorithm, for 10 out of the 11 benchmarks with smaller synthesized circuits, has produced smaller care-sets. Furthermore *Lazy$_{GD}$* has produced smaller care-sets for 11 out of the 15 benchmarks with smaller synthesized circuits.

Out of the 234 benchmarks, the *Lazy* algorithm solved 99 before the timeout, and the *Lazy$_{GD}$* algorithm solved 116. While the standard algorithm solves a higher number of instances overall (163), for a number of examples our algo-

Table 6.2: Comparison of care-set and winning region size for selected benchmarks
(number of BDD nodes in symbolic representation)

Instance	Standard	Lazy	Lazy$_{\text{GD}}$
load_2c_comp_comp5	**299**	986	518
load_3c_comp_comp4	**345**	9299	669
load_3c_comp_comp7	**442**	11253	1507
load_4c_comp_comp4	**2075**	*TO*	3263
load_4c_comp_comp6	**4413**	*TO*	7814
load_full_2_comp5	**1308**	*TO*	2182
load_full_2_comp7	**2068**	*TO*	5071
ltl2dba_C2-6_comp3	**199**	484	501
ltl2dba_E4_comp3	7361	**454**	508
demo-v10_5	**16**	83	49
demo-v12_5	**10**	44	34
demo-v14_5	**53**	83	87
demo-v16_5	262	233	**132**
demo-v18_5	125535	*TO*	**52687**
demo-v19_5	156	83	**76**
demo-v20_5	**87**	135	600
demo-v22_5	**226**	1373	1768
demo-v23_5	**46**	139	92
demo-v24_5	**195**	4075	561

rithms are faster. In particular, both versions each solve 7 benchmarks that are not solved by the standard algorithm, as shown in Table 6.5.

Moreover, we compare against the participants of SYNTCOMP 2017: with a timeout of 3600s, the best single-threaded solver in SYNTCOMP 2017 solved 155 problems, and the virtual best solver (VBS; i.e., a theoretical solver that on each benchmark performs as good as the best participating solver) would have solved 186 instances. If we include our two algorithms with a timeout of 3600s, the VBS can additionally solve 7 out of the 48 instances that could not be solved by any of the participants of SYNTCOMP before. As our algorithms also solve some instances much faster than the existing algorithms, they would be worthwhile additions to a portfolio solver for SYNTCOMP.

6.7.3 Random Benchmarks

The SYNTCOMP benchmark library consists of crafted benchmarks that were submitted by the participants. An inspection of these benchmarks shows that in many cases these benchmarks are such that progress towards the error states can *only* be avoided when we reach the border of the winning region. Obviously,

Table 6.3: Comparison of the size of solutions for selected benchmarks (number of AND-gates in synthesized circuit / number of BDD nodes in winning strategy)

Instance	**Standard**	**Lazy**
amba10c6n	28137 / 18612	28621 / 18711
driver_d10y	226581 / 140789	**156776** / **105244**
factory_assembly_7x5_2_0errors	31469 / 22841	**19853** / 18541
genbuf12c30n	3914 / 4808	**2153** / 3278
genbuf24c30y	23974 / 15528	18495 / 12365
genbuf56c40n	135025 / 59629	*TO*
genbuf8c30n	5767 / 5536	5753 / 5463
genbuf16c4y	**7137** / 10605	49373 / 55322
demo-v18_5_REAL	**50620** / **88189**	*TO*
driver_d8n	**171348** / **108099**	*TO*
factory_assembly_5x5_2_0errors	**11187** / **8578**	21078 / 13728
factory_assembly_5x5_2_1errors	**21300** / **17118**	46963 / 33821

benchmarks with such a structure benefit the standard backward approach and do not allow the lazy synthesis approach to show its strengths.

To obtain additional benchmarks that avoid the potential bias of handcrafted examples, we developed a tool called AIGEN for generating random benchmarks. Our tool implementation takes as input the number of controllable variables c, the number of uncontrollable variables u and the number of latches l, and generates an AIGER benchmark based on a uniformly random distribution over all controllable systems with these parameters. The implementation uses ROBDD as an intermediate representation of the benchmark to be generated and therefore the uniformity is guaranteed by the canonicity of ROBDDs. For a fixed u, c, and l there are $2^{2^{u+c+l}}$ different Boolean functions, and we need one such function to update each latch, and in addition we need a Boolean function only over l that determines the BAD output. Thus, overall we have a space of $(2^{2^{u+c+l}})^l \cdot (2^{2^l})$ possible benchmarks. Optionally, we can restrict the number o of latches that are used to define BAD. Using fewer latches for BAD decreases the expected size of the error states and increases the chance of obtaining a realizable benchmark. Chapter 7 describes the AIGEN tool in details. We have generated thousands of random benchmarks from different classes, where a class is defined by the number of controllable variables, uncontrollable variables, latches, and output function variables. A primary observation is that whenever a benchmark is easy to solve because there are many winning strategies (e.g., if parameters o and u are much smaller than l and c, respectively), then the standard algorithm is usually able to find a solution faster. However, when it is hard to find a winning strategy, then the results change. For instance, we compared the lazy algorithm with and without gen-

Table 6.4: Comparison of the size of solutions for selected benchmarks (number of AND-gates in synthesized circuit / number of BDD nodes in winning strategy)

Instance	**Standard**	**Lazy$_{\mathbf{GD}}$**
amba10c6n	28137 / 18612	**25816** / **17466**
driver_d10y	226581 / 140789	*TO*
factory_assembly_7x5_2_0errors	31469 / 22841	21453 / **17713**
genbuf12c30n	3914 / 4808	2278 / **3172**
genbuf24c30y	23974 / 15528	**9789** / **6529**
genbuf56c40n	135025 / 59629	**65284** / **32154**
genbuf8c30n	5767 / 5536	**5189** / **4890**
genbuf16c4y	**7137** / 10605	7375 / **10469**
demo-v18_5_REAL	**50620** / **88189**	140172 / 105279
driver_d8n	**171348** / **108099**	180917 / 116347
factory_assembly_5x5_2_0errors	**11187** / **8578**	22586 / 16680
factory_assembly_5x5_2_1errors	**21300** / **17118**	52730 / 33430

eral deletion against the standard algorithm on 100 random benchmarks with $c = 3, u = 1, l = 13$ (i.e., 17 variables overall), and $o = 12$. For 66 benchmarks, both of our algorithms synthesized circuits that were smaller or equal to the solutions of the standard algorithm (out of these 66, 30 where strictly smaller). Moreover, the *Lazy* algorithm solved 26 faster than the standard algorithm, and the $Lazy_{GD}$ algorithm was faster on 33 benchmarks.

Figures 6.11, 6.12, 6.13 compare solving time between the *Lazy* and the standard algorithm, for benchmarks with 17,18, and 19 variables respectively. For the benchmarks with 19 variables the *Lazy* algorithm solved 6 instances out of 100 that the standard algorithm could not solve, visible on the line on the right-hand side, marked with TO. The remaining benchmarks that we generated had 16 or fewer variables, and every single benchmark could be solved by all three algorithms, usually in a few seconds.

For the benchmarks we generated, we chose the parameters of our random generator in order to obtain interesting benchmarks, i.e.,

1. not too easy to solve (benchmarks with ≤ 16 variables can almost always be solved very quickly)

2. not too hard to solve (for 19 variables, both tools already run into a timeout on many examples), and

3. preferably realizable (by having more controllable than uncontrollable inputs).

We prefer realizable benchmarks since we also want to compare properties of the solutions, such as care-set/winning region or the size of the synthesized circuit.

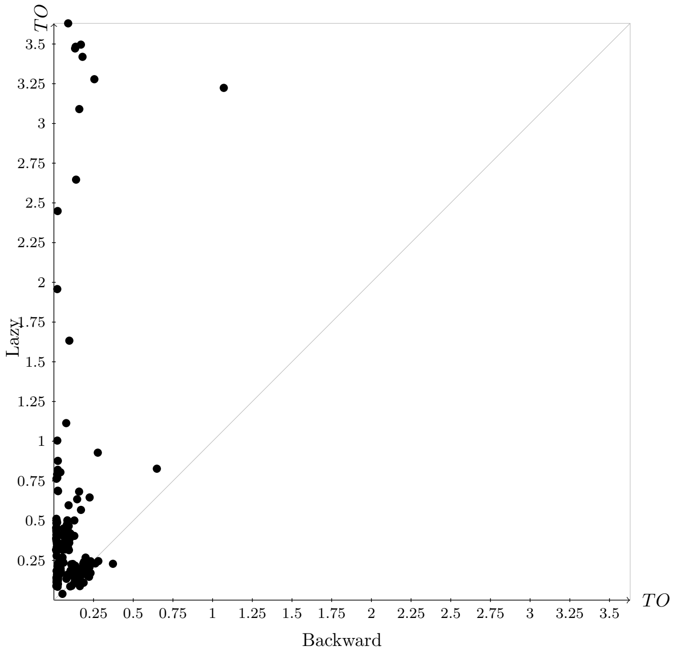


Figure 6.11: Time comparison between backward and lazy approaches for benchmarks with 17 variables. 100 random benchmarks with $c = 3, u = 1, l = 13, o = 12$, and 100 random benchmarks with $c = 3, u = 1, l = 13, o = 11$

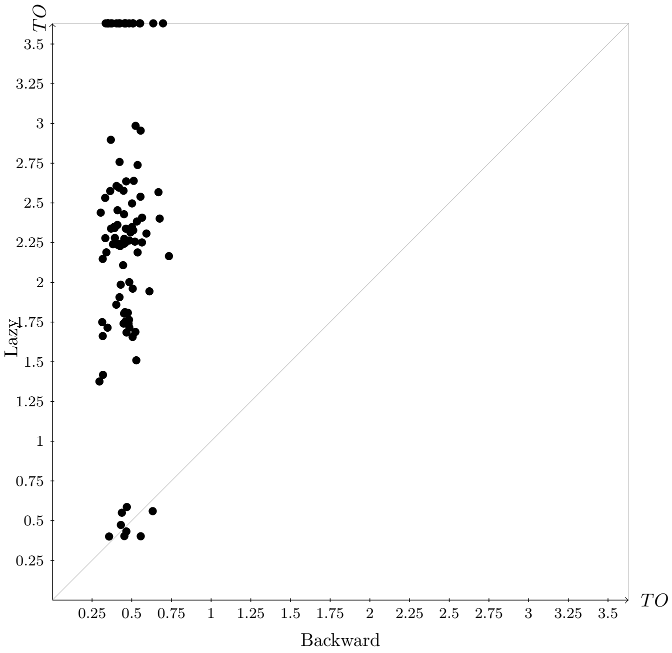


Figure 6.12: Time comparison between backward and lazy approachses for benchmarks with 18 latches. 100 random benchmarks with $c = 3, u = 1, l = 14, o = 12$

Table 6.5: Comparison of solving time for benchmarks solved by a lazy algorithm, but not by the standard algorithm (seconds)

Instance	**Standard**	**Lazy**	**Lazy$_{\text{GD}}$**	**SYNTCOMP 2017 Participants**
gb_s2_r3_comp1	*TO*	38	*TO*	solved by 1
genbuf48c6y	*TO*	*TO*	3839	solved by 4
ltl2dba_E6_comp4	*TO*	2435	*TO*	not solved
ltl2dba_Q4_comp5	*TO*	125	304	solved by 1
ltl2dba_U1-6_Comp3	*TO*	*TO*	4590	not solved
ltl2dpa_alpha5_Comp2	*TO*	*TO*	1880	not solved
ltl2dpa_alpha5_Comp3	*TO*	*TO*	2651	not solved
ltl2dpa_E4_comp2	*TO*	1081	*TO*	not solved
ltl2dpa_E4_comp4	*TO*	2122	*TO*	not solved
ltl2dpa_U14_comp2	*TO*	4019	615	not solved
ltl2dpa_U14_comp35	*TO*	2605	1681	not solved

6.8 Why Not a Purely Forward Exploration?

6.8.1 A Forward Algorithm

In Section 6.3.2 we mentioned a completely forward algorithm presented by Cassez et al. [28]. The algorithm starts from the initial state and explores all states that are reachable in a forward manner and checks if they can avoid moving to a losing state. The algorithm is not symbolic and it explicitly enumerates states and transitions. In this section, we propose a symbolic implementation and report on our experiences with integrating this form of forward exploration into our algorithms.

For a symbolic implementation, given a set of states, we need to compute all states that are reachable from this set in one transition. This can be accomplished by computing the image as defined in Section 6.6. However, image computation is very expensive in terms of memory and may be as hard as computing the whole transition relation. We explain below two methods that aim to reduce the overhead of image computation.

Early quantification [26]

When computing a BDD respresentation of an expression that contains existential quantification, it is desirable to evaluate terms under existential quantifiers as early as possible: existentially quantified variables can be completely eliminated from the ROBDD, which often results in a considerable reduction in the size of the BDD. Unfortunately, existential quantification is not distributive over conjunction and therefore we often have to first compute the result of the conjunction before we can remove the quantified variables.

However, if a term contains variables that are used only in this term, then existential quantification of these variables can always be performed locally.

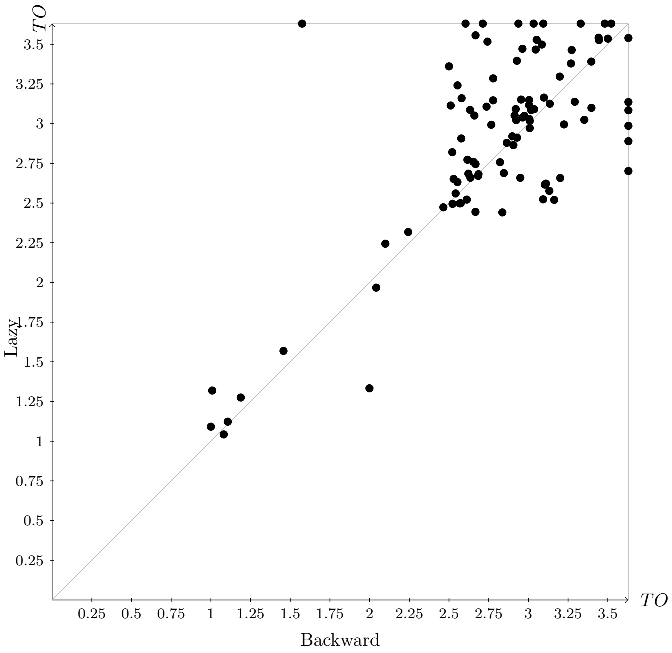


Figure 6.13: Time comparison between backward and lazy approaches for benchmarks with 19 latches. 100 random benchmarks with $c = 4, u = 2, l = 13, o = 12$

For synthesis algorithms, this would be useful for the update functions f_i of latches l_i. To take advantage of this property, one can use heuristics to search for a convenient ordering of update functions, represented as a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that as many existentially quantified subexpressions can be evaluated as early as possible. For a Boolean function f, we denote by $\mathit{support}(f)$ the set of variables which constitute f, i.e., the set of variables that f depends on. Let $E_i \leftarrow \mathit{support}(f_{\pi(i)}) \setminus \bigcup_{k=1}^{i-1} \mathit{support}(f_{\pi(k)})$ then the image can be computed as follows:

1. $S_1 \leftarrow C$
2. $S_{i+1} \leftarrow \exists x \in E_i \; (S_i \wedge (l'_{\pi(i)} \equiv f_{\pi(i)}))$
3. $\mathit{image}_f(C) = S_{n+1}$

We did not implement this optimization since an inspection of the benchmark set from SYNTCOMP has shown that the support of most of the update functions and the ROBDDs that represent them contain all the variables, which makes this optimization unlikely to be very useful. It remains open if this could be a worthwhile addition to synthesis tools, at least on certain kinds of benchmarks.

The range function

Given a function $f : \mathcal{A} \to \mathcal{B}$ and $C \subseteq \mathcal{A}$, define:

- *range*$(f) = \{y \in \mathcal{B} \mid \exists x \in \mathcal{A}$ with $y = f(x)\}$.
- *image*$(f, C) = \{y \in \mathcal{B} \mid \exists x \in C$ with $y = f(x)\}$.

Given $f : \mathcal{A} \to \mathcal{B}$ one can easily see that *range* can be used to compute the image of f on $\mathcal{A}$, since *range*$(f) =$ *image*$(f, \mathcal{A})$. However what we need for forward exploration of the state space is a way to compute *image*(f, C) for arbitrary $C \subseteq \mathcal{A}$.

Another way to view this is that instead of *range*(f), where f covers the whole set $\mathcal{A}$, we are interested in *range*(f') for some restriction f' of f that is only defined on C. To this end, Coudert et al. [35] introduced the *constraint operator*, a variant of the generalized cofactor [93].

Definition 3. *Let* $f_i : \mathbb{B}^m \to \mathbb{B}$, $\vec{F} : \mathbb{B}^m \to \mathbb{B}^n$ *with* $\vec{F} = (f_1, \dots f_n)$ *and* $C : \mathbb{B}^m \to \mathbb{B}$. *The* generalized cofactor $\vec{F}|_C = (f_1|_C, \dots f_n|_C)$ *is defined as*

$$\vec{F}|_C(x) = \begin{cases} \vec{X} & \textit{if } C(x) = \textit{False} \\ \vec{F}(x) & \textit{if } C(x) = \textit{True} \end{cases}$$

In the above definition $\vec{X}$ denotes a vector of "don't care" values, and these can be chosen in a way that makes the BDD that represents $\vec{F}|_C$ smaller than the BDD of $\vec{F}$.

Now, we can use the following theorem to implement the desired image computation:

Theorem 23. *[35, 93] range*$(\vec{F}|_C) =$ *image*$(\vec{F}, C)$

Although this method alleviates the memory requirements of $\bigwedge_{i=1}^{|L|} l'_i \equiv f_i \wedge C'$, the algorithm to compute *range*$(\vec{F}|_C)$ is a recursive algorithm [75] that requires a large number of recursive calls.

We have tried to integrate this form of forward search into our algorithms in order to detect unreachable states and prune the corresponding error paths. In our experiments, we experienced unacceptably large computation times, even on rather small examples, and with optimizations such as storing intermediate computation results. Therefore, we do not expect an algorithm based on *image* or *range* computation to be competitive on the class of benchmarks that we considered.

6.9 Synthesis of Resilient Controllers

As mentioned in Section 6.1, our algorithm produces strategies that avoid progress towards the error states as early as possible, which can be useful for generating controllers that allow for a margin of error, e.g. in the presence of sporadic faults or perturbations. In this section we review an algorithm that generates strategies with resilience to perturbations, compare it to the lazy synthesis algorithm and observe commonalities of their behavior on certain benchmarks.

6.9.1 Controllable Systems with Perturbations

Dallal et al. [36] have modeled systems with *perturbations*, which are defined as extraordinary transitions where values for the controllable inputs, or a subset thereof, are chosen non-deterministically. Thus, in a perturbation step Player 0 has only limited control over the behavior of the system, or none at all.

Formally, we modify controllable transition systems by fixing a subset $X_P \subseteq X_C$ and considering a transition relation of the form $\mathcal{R} : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_P} \times \mathbb{B}^{X_c \setminus X_P} \times \mathbb{B}^{L'} \rightarrow \mathbb{B}$. Given a set of solution functions for X_c and a bound k on the number of perturbations, the semantics of the composed system is the same as before, except that in a given run of the system, up to k times the values for variables in X_P are not chosen according to the solution function, but can be arbitrary.

Then, we are interested in an upper bound on the number of perturbations such that the synthesis problem can still be solved, and in strategies for the system with this number of perturbations, called *maximally resilient strategies.*

6.9.2 An Algorithm for Synthesis of Resilient Controllers

Dallal et al. [36] introduced an algorithm that produces maximally resilient strategies. It can be summarized as follows:

1. use the standard fixed-point algorithm to compute the winning region without perturbations,

2. use a mixed forward/backward analysis to find a strategy that makes as little progress towards the losing region as possible.

The second part can be seen as a variant of our lazy synthesis algorithm, except that it only has to handle a restricted setting: instead of the error states, the winning region can be used as a basis for the backwards analysis, and the forward analysis is simplified by the fact that from all states inside the winning region there is a winning strategy, so no backtracking is necessary to remove states from which winning is impossible.

6.9.3 Implementation, Experiments and Comparison to Lazy Synthesis

We have implemented this algorithm as a combination of the backward fixed-point algorithm and symbolic lazy synthesis, providing to our knowledge its first implementation. An evaluation on the SYNTCOMP 2017 benchmarks provides interesting insights: only on 6 out of the 234 benchmarks the algorithm can give a guarantee of resilience against one or more perturbations, as shown in Table 6.6.

When inspecting the behavior of our lazy algorithms on these benchmarks, we find that for 5 out of 6 benchmarks, our algorithms can give an additional *quantitative safety guarantee* by measuring the distance between the error states and any states that can be visited under the given strategy. Note that this information can be extracted without additional cost, simply by inspection of the final sequence of error levels. However, also note that this distance is not the same as the resilience property of Dallal et al., since (i) we compute the distance to the unsafe states, not to the losing region, and (ii) we do not take into account whether after a single perturbation there is still a winning strategy for Player 0. Thus, a distance of k to the unsafe states does not imply that the

Table 6.6: Benchmarks with Resilience Guarantees.
For Lazy Synthesis, we give the distance to error states, regardless of controllability after a perturbation. For Dallal et al., we give the distance to the losing region, taking controllability into account, i.e., the number of perturbations that the controller is resilient against.

Instance	**Lazy Synthesis**	**Dallal et al.** [36]
beembrdg2f1_c0to1	32	32
demo-v10_5	6	6
demo-v12_5	7	7
demo-v20_5	6	5
ltl2dba_C2-6_comp3	0	3
ltl2dba_E4_comp3	4	4

strategy is resilient to k perturbations—in fact, such a strategy does not always exist, as the results for benchmark demo-v20_5 in Table 6.6 show.

Furthermore, we observe that on all of these examples our algorithms detect a care-set that is much smaller than the full winning region. The results in Table 6.1 include 5 of the 6 benchmarks, and show that the care-sets provided by lazy synthesis are smaller by a factor of 10^9 or more. This leads us to the conjecture that lazy synthesis performs particularly well on synthesis problems that allow resilient controllers, together with the observation that not many of these appear in the SYNTCOMP 2017 benchmark set that we have tested against.

6.10 Conclusion

We have introduced lazy synthesis algorithms with a novel combination of forward and backward exploration. Our experimental results show that in many cases our algorithms detect solutions with care-sets that are much smaller than the full winning region. Moreover, they can solve a number of problems that are intractable for existing synthesis algorithms, both in crafted and random benchmarks. Finally, our algorithm produces eager solutions, and in some cases we can give quantitative safety guarantees, i.e., we can determine the minimum distance to any error state that a system running on our generated strategy will keep during execution.

Chapter 7

Random Generation of Symbolic Transition Systems

Verification and synthesis algorithms require a set of benchmark problems that can be used for testing and evaluation. Unfortunately a diverse set of benchmarks is very hard to obtain. This is a tedious problem not only for tool developers, but also for organizers of competitions [12, 14, 27, 63] that need to test and evaluate tools on a wide range of benchmarks, and to regularly search for new meaningful benchmarks.

If done properly, the generation of random benchmarks can be a solution to this problem by providing the best possible diversity and by generating new benchmarks whenever needed. On the other hand, random benchmarks come with a few caveats. First of all, totally random generation is usually not desired, since that may result in many benchmarks that, while drawn from a diverse set, are not interesting for different reasons, e.g. due to the fact that they may be too easy or too difficult to solve for existing tools. Second, the user may be interested in how his implementation handles benchmarks from a specific subset of the space of all possible benchmarks, for instance those that require long chains of computations to reach a conclusion. Finally, if the user knows how *realistic* benchmarks for a certain type of verification or synthesis problem usually look like, he may want to restrict the random generation to such benchmarks, e.g. by forcing them comply with certain conditions on their structure.

In this chapter we consider random generation of transition systems in a symbolic representation. Like in Chapter 6, the transition system we generate has a partitioned transition relation, i.e., it consists of a set of Boolean functions. To ensure diversity, we require a uniform random sampling over the space of all Boolean functions with a given number of variables.

While for some application areas there exist tools that generate random Boolean functions in a specific form (e.g. randomly generated propositional formulas in CNF [29, 82]), to the best of our knowledge none of these supports uniformly random distributions over sets of Boolean functions with a given number of inputs. The obvious benefit is that random samplings allow to make statements about the actual space of Boolean functions, instead of statements about a specific representation of the functions. These benefits extend to the random generation of transition systems.

To ensure uniform random sampling, we need a canonical representation for Boolean functions. To this end we can enumerate truth tables randomly and generate out of them, in a straight forward way, random Boolean functions in canonical disjunctive normal form (CDNF) or canonical conjunctive normal form (CCNF). As a more memory-efficient alternative and the main contribution of this chapter, we present an encoding that is inspired by data structures used for implementing reduced ordered binary decision diagrams (ROBDDs).

The tool AIGEN implements our ROBDD-based algorithm and a CDNF-based algorithm. Development of AIGEN was motivated by the evaluation of reactive synthesis tools [66]. It was used to generate benchmarks for the reactive synthesis competition (SYNTCOMP) [63, 64], and we used it to test the lazy synthesis algorithm that is presented in Chapter 6. Since the existing benchmark library of SYNTCOMP consists mostly of benchmarks that were hand-crafted by tool developers, the diversity of benchmarks is limited, and their choice may be skewed towards problems or encodings that are well-suited for the existing tools. Therefore, random benchmarks are a valuable source of insight into the performance of synthesis algorithms, when used in addition to the existing hand-crafted examples.

Outline. We introduce BDDs and ROBDDs in Section 7.1. In Section 7.2 we present our basic idea for the random generation of symbolic transition systems, based on enumerating Boolean functions. In Section 7.3, we present a detailed description of the ROBDD-based algorithm, and in Section 7.3.2 the algorithm based on CDNF. Section 7.4 explains how we can efficiently use AIGEN and produce diverse benchmarks. Finally, in Section 7.5 we present a comparison between the ROBDD and the CDNF approaches, and we give details about our implementation.

7.1 Canonical Representation of Boolean Functions

A *Binary Decision Diagram (BDD)* over a set of variables X is a directed acyclic graph $G = (V, E)$ with $V \subset \mathbb{N}$, exactly one root $v_r \in V$, and a labeling on nodes with the following properties:

- each terminal node $v \in V$ is labeled with a value $val(v) \in \{0, 1\}$
- each non-terminal node $v \in V$ is labeled with a variable $var(v) \in X$ and has exactly two outgoing edges, leading to nodes that are denoted by $high(v) \in V$ and $low(v) \in V$, respectively.

Note that if $v \in V$ is a non-terminal node, then the directed acyclic graph rooted in v is also a BDD. It is called the *sub-BDD of G with root v*.

A BDD G over a set of variables X is *ordered* if on every path from the root to a terminal node, variables in node labels occur in the same order and each variable occurs at most once.

A BDD $G(V, E)$ is *reduced* if it does not contain any of the following:

- non-terminal nodes $v \neq w \in V$ with $var(v) = var(w)$, $low(v) = low(w)$ and $high(v) = high(w)$,
- terminal nodes $v \neq w \in V$ with $val(v) = val(w)$,
- a non-terminal node $v \in V$ with $low(v) = high(v)$.

Any ordered BDD can be transformed into a reduced BDD by using the isomorphism and Shannon reductions (cp. [40]). A BDD that is reduced and ordered is called a *Reduced Ordered Binary Decision Diagram (ROBDD).*

Note that in an ROBDD, a triple $(x, high(v), low(v))$ of a non-terminal node v, where $x = var(v)$, uniquely defines a sub-ROBDD. This implies that ROBDDs are a canonical representation of Boolean functions [40], i.e., for a fixed variable order there is a unique ROBDD representation for every Boolean function.

7.2 Enumerating Boolean Functions

Based on a canonical representation of Boolean functions, we define an enumeration, i.e., a bijective mapping from natural numbers to Boolean functions (or ROBDDs), such that any procedure that produces uniformly random natural numbers (in some range) can be used to produce uniformly random Boolean functions (in some range, see below for details).

To define our mapping, we first describe the data structure for ROBDDs that is used by various BDD packages. Then we will illustrate the data structure we use for ROBDDs and how it guarantees canonicity and uniform random distribution. In the following, we assume that $X = \{x_1, \ldots, x_m\}$ is a set of variables with a fixed order.

Unique Table.

BDD packages use the so-called *unique table* as a base data structure for storing ROBDD nodes. The unique table of a BDD $G = (V, E)$ over a set of variables X is a hash table that establishes a bijection between nodes $v \in V$ and triples $(x, h, l) \in X \times V \times V$ that uniquely identify them, where $x = val(v)$ if v is a terminal node, and $x = var(v)$ otherwise, $h = high(v)$ and $l = low(v)$. A terminal node has no children nodes and therefore has no h and l.

7.2.1 Virtual ROBDD Table

We will use the ideas from the unique table that is used in BDD packages to define the virtual ROBDD table that enumerates all possible ROBDDs with respect to our variable order. This table can of course not be constructed explicitly, but the idea of this table can be used to define a (bijective) mapping from natural numbers to ROBDDs.

To generate a ROBDD that represents a random Boolean function with m inputs, the idea of our algorithm is to generate randomly a natural number $bddID \leq 2^{2^m}$ (since there are 2^{2^m} different Boolean functions of type $\mathbb{B}^m \rightarrow \mathbb{B}$), compute the unique triple that corresponds to $bddID$, and iteratively build the ROBDD by computing the unique triples for its children.

To illustrate how the algorithm computes these triples, assume that there exists a table, called *Virtual ROBDD Table (or short: VRT)*, that maps natural numbers to ROBDDs, identified by a triple of variable index and high and low children. In other words, every entry in the table maps uniquely a number $bddID \in \mathbb{N}$ to a tuple $(level, high, low)$ where **level** is a variable index. Like the unique table, none of the entries (i.e., ROBDDs) appears twice. However, in contrast to the unique table, the VRT is based on the fixed variable order, and uses the variable index in this order instead of the variable itself. Table 7.1 depicts a sketch of the VRT.

Table 7.1: VRT: Entries in the table are in ascending order over $bddID$. The first row in the table corresponds to level 0 which include the two terminal nodes 1 and 2 which points to the values 0 and 1 respectively. Each row (except the first row) is annotated with a *level* and a *sublevel*. L_i denotes the i^{th} level, containing all triples with variable index i. The *sublevel* sl_{i_j} denotes the j^{th} sublevel of L_i which contains all triples of L_i where either the *high* element or the *low* element is equal to j. Each cell in a row annotated with L_i and sl_{i_j} is of the form $(bddID)[high.low]$ where $bddID$ is the unique identifier of the triple $(i, high, low)$. Let $X = 2^{2^{i-1}}$ and $Y = \sum_{m=1}^{j-1} 2(2^{2^{i-1}} - m)$.

L_0		(1)[**0**]	(2)[**1**]				
L_1	sl_{1_1}	(3)[1.2]	(4)[2.1]				
L_2	sl_{2_1}	(5)[1.2]	(6)[2.1]	(7)[1.3]	(8)[3.1]	(9)[1.4]	(10)[4.1]
	sl_{2_2}	(11)[2.3]	(12)[3.2]	(13)[2.4]	(14)[4.2]		
	sl_{2_3}	(15)[3.4]	(16)[4.3]				
$\vdots$	$\vdots$		$\vdots$				
L_i	sl_{i_1}	$(X{+}1)[1.2]$	$(X{+}2)[2.1]$		$\dots$		$(X{+}2(X{-}1))[X.1]$
	$\vdots$		$\vdots$				
	sl_{i_j}	$(X{+}Y{+}1)[j.j{+}1]$		$\dots$			$(X{+}Y{+}2(X{-}j))[X.j]$
	$\vdots$		$\vdots$				
	$sl_{i_{X-1}}$						
$\vdots$	$\vdots$						

7.2.2 Computing ROBDDs from Random Numbers

In the VRT, a $bddID$ between 1 and 2^{2^m} corresponds to a Boolean function with at most m input variables, and a $bddID$ between $2^{2^{m-1}} + 1$ and 2^{2^m} corresponds to a function with exactly m input variables. Thus, to uniformly sample Boolean functions, we can use a random number generator that uniformly samples natural numbers in such a range.

It is important to remember that the VRT is not constructed explicitly. Instead, given a number of variables m, the algorithm generates first a random number $bddID \leq 2^{2^m}$, then computes the triple $(level, high, low)$ to which $bddID$ maps in the VRT. We note: $level$ (or i) is equal to $\lceil log_2(log_2(bddID))\rceil$. Let $X = 2^{2^{i-1}}$, then we solve the following system of equations to compute x which is equivalent to the *sublevel*:

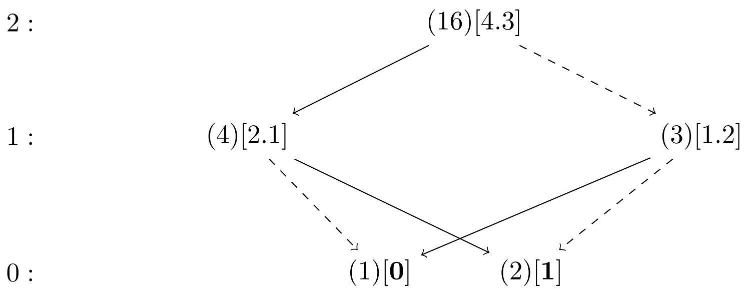


Figure 7.1: BDD generated for number 16. Equivalent to the Boolean function: $x_2x_1 + \bar{x_2}\bar{x_1}$. The numbers on the left of the BDD represent the *level* i.e. the corresponding variable indices.

$X + 2(X - 1) + \ldots + 2(X - x) < bddID$
$X + 2(X - 1) + \ldots + 2(X - (x + 1)) \geq bddID$

High and low are then computed according to what is given in the table, see Section 7.3 for more details. Figure 7.1 shows the BDD generated for $bddID = 16$ which is equivalent to: $x_2x_1 + \bar{x_2}\bar{x_1}$.

7.3 Random Generation of (Controllable) Transition Systems

In this section we present our algorithm for generating random transition systems, represented as AIGER circuits [15]. We use a generalization of the usual notion of transition systems that allows some of the input signals to be declared as controllable. This is useful to define synthesis problems, i.e., a synthesis procedure can define how these inputs should behave depending on the state and uncontrollable inputs of the system.

A *controllable transition system* (check Section 6.2 for more details) TS is a 6-tuple $(L, X_u, X_c, \vec{F}, BAD, q_0)$, where:

- L is a set of state variables (also called *latches*)
- X_u is a set of uncontrollable input variables
- X_c is a set of controllable input variables
- $\vec{F} = (f_1, ..., f_{|L|})$ with $f_i : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \to \mathbb{B}$ is a vector of update functions for the latches
- $BAD : \mathbb{B}^L \to \mathbb{B}$ is the set of unsafe states
- q_0 is the initial state where all latches are initialized to 0.

Then, the idea of our algorithm for random generation of transition systems can be summarized in the following way:

- as input, the procedure receives numbers $l, c, u, o \in \mathbb{N}$ and optionally a list of seeds (i.e., natural numbers used to initialize a pseudorandom number generator), where l is the desired number of latches, c, u are the numbers of controllable and uncontrollable inputs, respectively, $o \leq l$ is a parameter that bounds the size of the set of unsafe states to be less than 2^o, and the list of seeds can be used if the user wants to generate a replica of the transition system that used the same list of seeds to generate the random numbers.

- for each latch, we generate a random function $\mathbb{B}^{l+c+u} \to \mathbb{B}$ that determines how this latch is updated based on the current state and input of the system.
- we generate a random function $f_{BAD} : \mathbb{B}^o \to \mathbb{B}$ that determines the set of unsafe states of the system: BAD is defined as $f(x_{i_1}, \ldots, x_{i_o}) \wedge \bigwedge_{j \in \{1,\ldots,l\} \setminus \{i_1,\ldots,i_o\}} x_j$ where the indices $\{i_1, \ldots, i_o\}$ are also picked randomly.
- the system composed of these functions is then encoded into an AIGER circuit.

7.3.1 ROBDD based Algorithm

The procedure GenerateRandomAiger takes as input the number of latches l, uncontrollable inputs u, controllable inputs c, the bound o, optionally a list of seeds, and produces a file in AIGER format as output. Lines 3-6 generate for every latch a random ROBDD that represents the *update function* of the latch, i.e., a function that takes all current values of inputs and latches as input, and returns a new value for the given latch. Line 4 generates a random integer with 2^{var} random bits, i.e., a natural number between 1 and $2^{2^{var}}$. All the seeds used for generating the random integers will be written in the comment section at the end of the generated file. These seeds can be fed to the algorithm in order to regenerate the same instance. Line 5 constructs the ROBDD that corresponds to the generated number. Line 6 converts the constructed ROBDD into an AIG (And Inverter Graph) relying on the fact that a BDD can be seen as a network of multiplexers. Lines 8-10 construct the ROBDD of the BAD function which uses $o \leq l$ latch variables. Line 11 creates the AIGER file that corresponds to the total number of variables and to the update functions that were randomly generated. Line 12 uses the *ABC* [13] tool to reduce the size of the generated AIGER file.

ConstructBDD is a recursive procedure for constructing all the nodes of the ROBDD that corresponds to the unique ID *bddID*. It starts with the root node and recursively proceeds to the child nodes until it reaches the nodes 1 or 2 (which corresponds to values 0 and 1 respectively). Line 2 checks if the node was already created. If not, Line 3 computes the triple $(level, high, low)$ that uniquely represent the node and adds it to the table $ROBDDTable$. Lines 5-6 construct the child nodes. Note that the $ROBDDTable$ is initialized with the IDs 1 and 2 which correspond respectively to the values 0 and 1.

Given an ID, the *GetChildren* procedure computes the triple $(level, high, low)$. Line 2 computes the level. Lines 3-6 compute the sublevel. Note that, as depicted in Table 7.1, a sub-level s_{i_j} has size $2(2^{2^{i-1}} - j)$, where $2^{2^{i-1}}$ is the sum of the sizes of all levels that are smaller than i. To compute the sublevel, we have to compute the single solution of the system of inequations in Lines 4,5, to see that check the VRT table. Line 7 computes the ID of the first element in the

Algorithm 6 Generate Random Aiger

```
procedure GenerateRandomAiger($l, u, c, o$)
    $vars \leftarrow l + u + c, l' = l, ROBDDTable = [(1,0),(2,1)]$
    while $l' > 0$ do
        $rand\_fct\_ID = random.getrandbits(2^{vars}) + 1$
        $ConstructBDD(rand\_fct\_ID, ROBDDTable)$
        $AIGERTable[rand\_fct\_ID] = ConvertToAIG(rand\_fct\_ID)$
        $l' \leftarrow l' - 1$
    $BAD\_ID = random.getrandbits(2^{o}) + 1$
    $ConstructBDD(BAD\_ID, ROBDDTable)$
    $AIGERTable[BAD\_ID] = ConvertToAIG(BAD\_ID, ROBDDTable)$
    $aigerFilePath \leftarrow CreateAiger(AIGERTable)$
    $ABCMinimize(aigerFilePath)$

procedure ConstructBDD($bddID, ROBDDTable$)
    if $bddID \notin ROBDDTable$ then
        $level, high, low \leftarrow GetChildren(bddID)$
        $ROBDDTable[bddID] \leftarrow (level, high, low)$
        $ConstructBDD(high)$
        $ConstructBDD(low)$

procedure GetChildren($bddID$)
    $level = \lceil log_2(log_2(bddID)) \rceil$
    $n \leftarrow 2^{2^{level-1}}$
    $sli \leftarrow ComputeASol(n + 2(n-1) + \ldots + 2(n-x) < bddID,$
                        $n + 2(n-1) + \ldots + 2(n-(x+1)) \geq bddID)$
    $child_1 \leftarrow sli + 1$
    $sl\_1\_ID \leftarrow n + 2(n-1) + \ldots + 2(n - sli)$
    $sle \leftarrow bddID - sl\_1\_ID$
    $child_2 \leftarrow child_1 + \lceil sle/2 \rceil$
    if $sle \mod 2 \neq 0$ then
        return $level, child_1, child_2$
    return $level, child_2, child_1$
```

sub-level. Lines 8-9 compute the ID of the second child node, and Lines 10-12 check which node is the low edge and which node is the high edge.

7.3.2 CDNF Approach

An obvious alternative to our ROBDD approach is to make use of the canonical disjunctive or conjunctive normal forms to generate random Boolean functions. Algorithm 7 employs CDNF as it is easier to convert to And-Inverter graph. CDNF is usually constructed directly from a truth table (which has an exponential size) by taking the OR of all satisfying assignments. To convert a Boolean formula $f_i = cl_1 \lor cl_2 \lor \ldots \lor cl_n$ in CDNF to AIG, we consider its equivalent $f'_i = \neg(\neg cl_1 \land \neg cl_2 \land \ldots \land \neg cl_n)$.

7.3.3 CDNF based Algorithm

Algorithm 7 Random Aiger generation using DNF approach

```
 1: procedure DNFGenerateRandomAiger($l, u, c, o$)
 2:     $vars \leftarrow l + u + c, l' \leftarrow 0$
 3:     while $l' < l$ do
 4:         $truthTable = random.getrandbits(2^{vars})$
 5:         $dnfFormula = ConstructDNF(truthTable, vars)$
 6:         $AIGERTable[l'] = ConvertToAIG(dnfFormula)$
 7:         $l' \leftarrow l' + 1$
 8:     $BADtruthTable = random.getrandbits(2^{o})$
 9:     $BADdnfFormula = ConstructDNF(truthTable, o)$
10:     $AIGERTable[l'] = ConvertToAIG(BADdnfFormula)$
11:     $aigerFilePath \leftarrow CreateAiger(AIGERTable)$
12:     $ABCMinimize(aigerFilePath)$

 1: procedure ConstructDNF($bitVec, vars$)
 2:     $dnfFormula \leftarrow True, i \leftarrow 0$
 3:     while $i < bitVec.size()$ do
 4:         if $bitVec[i] = 1$ then
 5:             $clauseBitvec \leftarrow ToBinary(i, vars)$
 6:             $dnfClause \leftarrow ToClause(clauseBitvec)$
 7:             $negate(dnfClause)$
 8:             $dnfFormula \leftarrow dnfFormula \land dnfClause$
 9:     return $negate(dnfFormula)$
```

The procedure DNFGenerateRandomAiger takes as input the number of latches l, uncontrollable inputs u, controllable inputs c, the bound o, and produces a file in AIGER format as output. Lines 3-6 generate for every latch a random update function. Line 4 generates a random bit vector of size 2^{vars}. This bit vector represents the valuation of all the *minterms*[1] of the truth table that represents the random function f_i. For instance, if the first element of the bit vector is equal 1 then $x_{c_0} = 0, \ldots, x_{c_{|c|-1}} = 0, x_{u_0} = 0, \ldots, x_{u_{|u|-1}} =$

[1] A minterm of n variables is a product (logical AND) of the variables in which each variable appears exactly once in uncomplemented or complemented form.

$0, x_{l_0} = 0, \ldots, x_{l_{|l|-1}} = 0$ is a satisfying assignment of f_i. Similarly if the last element of the bit vector is equal 1 then $x_{c_0} = 1, \ldots, x_{c_{|c|-1}} = 1, x_{u_0} = 1, \ldots, x_{u_{|u|-1}} = 1, x_{l_0} = 1, \ldots, x_{l_{|l|-1}} = 1$ is a satisfying assignment of f_i. Line 5 builds the random function that corresponds to the generated bit vector, and Line 6 converts it to AIG. Lines 8-10 generate the output random function, and Lines 11, 12 creates the AIGER file and call ABC to minimize it.

The procedure ConstructDNF takes as input a bit vector and the number of variables and generates the corresponding Boolean function. Line 2 initializes the DNF function to be created. For every element in the bit vector if the ith element is equal to 1, Line 3, then, in order to obtain the corresponding minterm, Line 5 converts the positive integer i to binary. For instance if $i = 3$ and vars $= 3$, then the minterm $x_c \wedge \neg x_u \wedge x_l$ is created. Line 6 creates the corresponding minterm. Line 7 negates the created clause and Line 8 adds is to the DNF formula. Line 9 returns the negation of the constructed formula. As mentioned earlier, as the formula represented by the truth table is in DNF, we need to generate its equivalent that includes only AND and NOT logical gates. For instance giving a formula $f_i = cl_1 \vee cl_2 \vee \ldots \vee cl_n$ in CDNF, we construct its equivalent $f_i' = \neg(\neg cl_1 \wedge \neg cl_2 \wedge \ldots \wedge \neg cl_n)$.

7.4 How to effectively use the tool

AIGEN implements both algorithms (Algorithm 6 and 7). It takes as input a file name *FileName*, the number of uncontrollable inputs u, the number of controllable inputs c, the number of latches l, the bound o, and optionally a seed for each latch. Given these inputs, the tool randomly generates an AIGER circuit (uniformly distributed in the given settings) *FileName.aag*, and an equivalent circuit of reduced size *FileName_abc.aag*. *FileName.aag* contains the exact AIG generated by the algorithm, and as comments the seeds used by the random number generator which can be fed as parameters to the tool in order to replicate the same file. *FileName_abc.aag* is obtained by running an ABC command on *FileName.aag* in order to reduce its size. Note that we can further reduce the size of *FileName.aag* using a BDD tool: once the file is read by a BDD manager, the size of the corresponding BDD can be minimized automatically using a variable reordering function. The minimized BDD can be then re-encoded into an AIGER file.

As mentioned, the tool can be used to generate benchmarks to evaluate model checking and/or synthesis tools. Although the benchmarks are randomly generated, the user can choose the input parameters to obtain benchmarks with certain properties that correspond to their needs, for example:

- The degree of the generated graph (i.e., the transition system) is equal to 2^{u+c}, therefore increasing the ratio $(u + c)/l$ will make the graph more congested and consequently more complex.

- The parameter o gives the user the ability to determine the size of the set of unsafe states, and as a consequence it is one of the factors that allows the user to manipulate the complexity of the benchmark. For instance, suppose that a user chooses $l = 10$ and $o = 3$, this means that the number of unsafe states cannot exceed 2^3 as the BAD function will be of the following form: $f(x_3, x_2, x_1) \wedge x_{10}x_9x_8x_7x_6x_5x_4$.

- Increasing the ratio o/l will increase the probability that the error set is reachable, and decreasing this ratio will lower the probability.
- Increasing the ratio c/u will increase the probability that the benchmark is realizable, and decreasing it will serve the opposite goal. Moreover, if this ratio is close to 1 the realizability check will be harder, since the probability of realizability will be roughly equal to the probability of unrealizability.

To demonstrate the effect of these parameters, Table 7.2 shows the running time and results (realizable or unrealizable) on selected benchmarks for the tool *SimpleBDDSolver* in SYNTCOMP 2019. SimpleBDDSolver has won all previous iterations of the SYNTCOMP competition safety track based on AIGER circuits. A benchmark name contains the parameters that was used to generate the file. For instance random_n_19_1_3_15_14_1_abc.aag means that the benchmark has in total 19 variables with 1 controllable input, 3 uncontrollable inputs, 15 latches, and $o = 14$. The table shows that the benchmarks with ratio $c/u = 1/3$ or $c/u = 1/5$ were unrealizable, the benchmarks with ratio $c/u = 2$ were realizable, while benchmarks with ratio $c/u = 1/2$ were difficult to solve for the tool, which timed out while trying to solve them. Note that the results of Table 7.2 can be found at the following URL: https://www.starexec.org/starexec/secure/details/job.jsp?id=35621

Table 7.2: Results of the SimpleBDDSolver tool on selected random benchmarks generated by AIGEN in SYNTCOMP 2019

Benchmark	**Time(s)**	**Results**
random_n_19_1_3_15_14_1_abc.aag	3412.41	UNREALIZABLE
random_n_19_1_5_13_13_2_abc.aag	1361.39	UNREALIZABLE
random_n_19_1_2_16_14_8_abc.aag	Timeout	-
random_n_19_1_4_14_13_11_abc.aag	Timeout	-
random_n_19_4_2_13_12_11_abc.aag	43.68	REALIZABLE
random_n_19_4_2_13_12_12_abc.aag	35.71	REALIZABLE
random_n_19_4_2_13_12_3_abc.aag	240.61	REALIZABLE
random_n_19_4_2_13_12_62_abc.aag	299.5	REALIZABLE
random_n_19_4_2_13_12_95_abc.aag	258.92	REALIZABLE

7.5 Implementation & Evaluation

AIGEN is implemented in Python, and available for download at https://github.com/mhdsakr/AIGEN-Tool. The tool is open source software, allowing interested users to add functionality, e.g., in order to add further parameters to generate only Boolean functions or transition systems with certain properties.

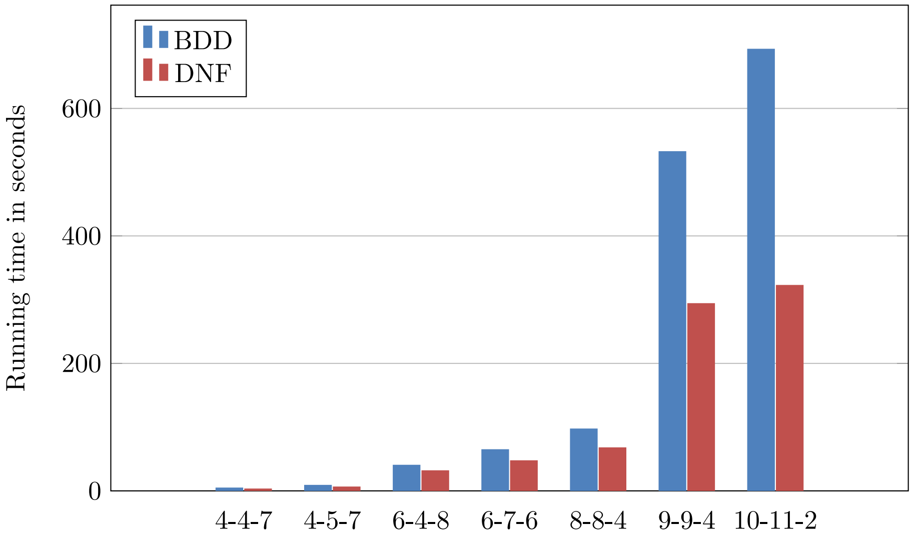


Figure 7.2: Average running times using ROBDD and DNF approaches

It uses the mpmath [70] library together with GMPY [1] to deal with large numbers. By default, mpmath uses Python integers, however if GMPY is also installed on the operating system, mpmath will automatically detect it and use gmpy integers intead. This makes mpmath perform much faster, particularly at high precision (approximately above 100 digits). Furthermore, AIGEN uses ABC [13], and the AIGER tool set [16] to post-process AIGER circuits.

AIGEN has been used to generate thousands of random transition systems. Figures 7.2, 7.3, and 7.4 show average times and sizes for the generated systems where, for example, 4-3-7 denotes systems with 4 controllable inputs, 3 uncontrollable inputs, and 7 latches ($o = l = 7$). These times were measured on a laptop with quad-core i7-6600U CPU at 2.6 GHz and 20 GB RAM.

Figure 7.2 shows average running time comparison between ROBDD and DNF approaches. These time results do not include the time needed for ABC tool to minimize the generated transition system (i.e. $ABCMinimize(FilePath)$ is skipped). The chart shows that the DNF approach was faster in all cases which was expected due to the fact that generating a random ROBDD is much more complex than generating a truth table. Figure 7.3 compares average number of AND Gates in generated transition systems. The depiction shows that the ROBDD approach is much better in all cases. Figure 7.4 shows average running time comparison between the ROBDD and DNF approaches including the time needed for ABC tool to minimize the generated transition system. Benchmarks 8-8-4, 9-9-4, and 10-11-2 timed out for the DNF approach(we used 10 hours as a time limit). Obviously the ABC tool needed a lot of time to process these benchmarks. After a thorough inspection, the reason was, in addition to the huge size of these circuits, the incredibly long chains of AND-gates for every generated Boolean function. This figure show that the total running time of the tool was way better when used with the ROBDD approach.

Transition systems generated by AIGEN have been used as benchmark problems in SYNTCOMP 2019. The difficulty of benchmarks can be scaled based

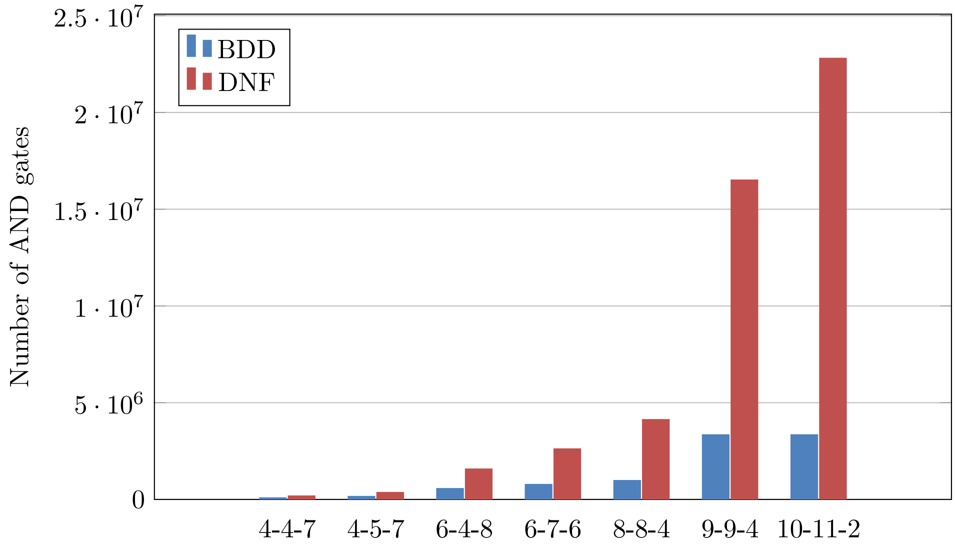


Figure 7.3: Average number of generated AND gates for ROBDD and DNF approaches

on several parameters, such as the number of inputs, number of latches, and ratio of controllable and uncontrollable inputs. In this way, we can generate benchmarks that are challenging for most or all existing synthesis tools, even though they may only be medium-sized (e.g., 15-20 variables overall, with a state space between 2^{10} and 2^{16} states).

7.6 Conclusion

Although the ROBDD based approach generates much smaller symbolic transition systems, the CDNF approach is faster when the ABC minimization procedure is disabled, however the resulting systems way smaller when the minimization procedure is used. In contrast to the ROBDD approach, to generate a random formula in CDNF, no heavy or complex computation is needed. However the huge size of these formulas becomes a problem for the ABC tool as it has to deal and inspect all the generated AND gates in order to carry out the minimization.

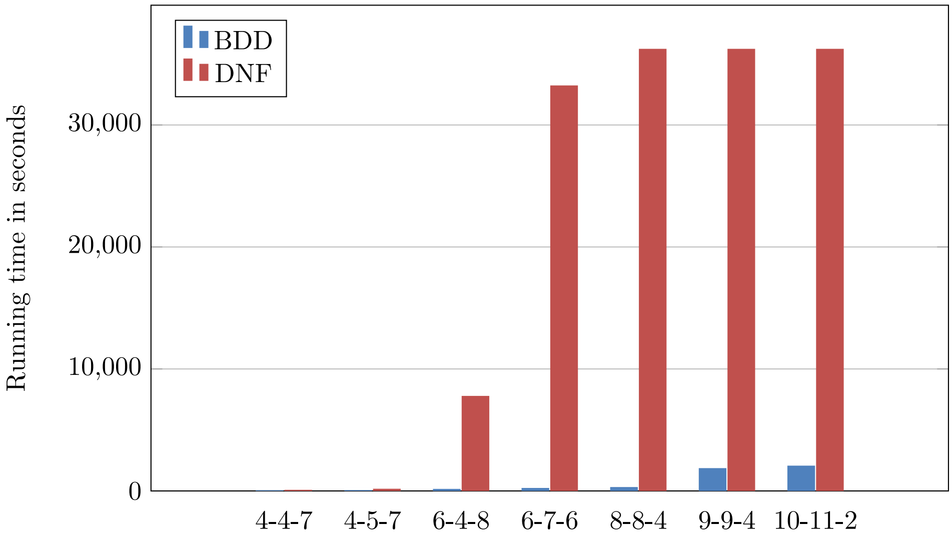


Figure 7.4: Average running time comparison including the time needed to minimize generated Aiger using ABC tool.

Chapter 8

Conclusion and Future Work

This last chapter of the thesis summarizes the main contributions and the chapters' content. Furthermore we suggest and discuss future works.

8.1 Summary

In this thesis, we presented novel techniques for model checking, repair and synthesis of parameterized concurrent systems. These new techniques allowed us to improve existing results for conjunctive, disjunctive, and token passing systems by extending the class of systems, the class of process templates, and the class of specifications for which the parameterized model checking problem is decidable. Furthermore, we presented, to our knowledge, the first repair algorithm for the synchronizations inside parameterized systems, and we showed how the algorithm may be applied on classes of systems that can be simulated by well structured transition systems (WSTS). Additionally we presented new approaches for the safety synthesis problem, and showed how one of these approaches solved many benchmarks that were not solved before.

8.1.1 Results for $\mathsf{LTL}\backslash\mathrm{X}$ Properties

In Chapter 3 we showed how we can profit from knowledge about the communication (specifically the number and the form of transition guards) and the structure of process templates in order to obtain better cutoff results on disjunctive and conjunctive systems for k-indexed $\mathsf{LTL}\backslash\mathbf{X}$ properties (see Table 8.1). Additionally we showed how our results enable us to verify systems that were intractable before (see Section 3.2). These results circumvent existing tightness results, which state that no smaller cutoffs can exist. Furthermore, we extended the existing cutoff results for a new class of disjunctive systems and a new class of specifications that have not been studied before (check Sections 3.3.4, and 3.3.5).

In Chapter 4 we investigated the parameterized model checking problem for guarded protocols and token passing systems under bounded fairness. Given a bound b, bounded fairness requires that in a global run of the system each process has to execute at least one transition in every b steps of the global system. To obtain cutoff results, we showed that if there exists a bounded-fair run that violates a specification in $\mathsf{LTL}\backslash\mathbf{X}$, then there exists an ultimately periodic run with the same property that can be extracted from the product of the automaton that represents the property and the system. Table 8.1 lists the obtained results.

8.1.2 Results for Prompt-LTL\X Properties

In Chapter 4 we investigated the parameterized model checking problem for specifications in Prompt-$\mathsf{LTL}\backslash\mathbf{X}$. Prompt-$\mathsf{LTL}$ is a linear time temporal logic that extends LTL with the prompt eventually operator $\mathbf{F_p}$. The formula $\mathbf{F_p}a$ is satisfied when the event a happens at some time in the future, and there exists a bound on the time that elapses before it happens. However, this new operator makes Prompt-LTL a stutter sensitive logic. For instance, given a run of a parameterized system that satisfies $\mathbf{GF_p}a$ for some bound b, we can construct a run that does not satisfy $\mathbf{GF_p}a$ for any bound b' by introducing an increasing (and unbounded) number of stuttering steps between every two appearances of a. Therefore, after establishing a connection between bounded fairness, bounded stuttering, and the satisfaction of Prompt-$\mathsf{LTL}\backslash\mathbf{X}$, we showed that we can modify existing approaches that solve parameterized model checking by the cutoff method to obtain correctness guarantees for specifications in Prompt-$\mathsf{LTL}\backslash\mathbf{X}$.

The types of systems that we covered in this chapter are guarded protocols and token-passing systems. Check Table 8.1 for more detailed results.

8.1.3 Deadlock Detection

A deadlock is an unwanted state of a parameterized concurrent system where one or more processes are unable to take any action. In this thesis we studied two types of deadlocks, global and local deadlocks. A run of a system is globally deadlocked if all processes are prevented from executing transitions. On the other hand, we say that a system run is locally deadlocked in a process p, if after some moment in the run, p can never execute any transition again.

Emerson and Kahlon [44] proved that, for conjunctive systems, a cutoff linear in the size of the process template is sufficient to detect global deadlocks. Außerlechneret al. [10] showed that, for local deadlock detection in conjunctive systems, a cutoff that is linear in the size of the process template is also adequate, but under a strong restriction where the results are only for systems in which each guard can only exclude a single state (1-conjunctive). Unfortunately, these results does not support even simple systems like the one in Figure 3.1. In Chapter 3, we expanded, for local deadlock detection in conjunctive systems, the class of process templates that have cutoff results. These results support classes of templates that are not 1-conjunctive, and handle systems like the one in Figure 3.1. Unfortunately we did not solve the general problem, however we proved that a cutoff for an arbitrary conjunctive system is at least quadratic in the size of the process template.

Furthermore, in Chapter 5, Section 5.3.3, we showed how we can modify a backward-based model checking algorithm (see Section 5.3.2) in order to detect global deadlocks in parameterized disjunctive systems. The modified version of the algorithm checks if a given counter system can reach a globally deadlocked state, where a counter system is an abstract model that can simulate parameterized disjunctive systems.

8.1.4 Parameterized Repair

In Chapter 5, we presented an algorithm for the repair of parameterized concurrent systems. Our repair algorithm interleaves the generation of candidate solutions (repairs) with parameterized model checking and parameterized deadlock detection approaches. The parameterized model checker and the parameterized deadlock detector provide the algorithm with the needed information in order to direct the search for candidate repairs. This information is encoded into propositional constraints in order to use a SAT solver to automatically find correct repairs. To achieve that, we modeled disjunctive systems as well-structured transition systems. Due to the well quasi order over its states, this system model facilitates parameterized model checking and parameterized deadlock detection.

The repair algorithm is not restricted to disjunctive systems, therefore we showed how it can be extended to other types of systems, like pairwise rendezvous, and broadcast protocols.

8.1.5 Safety Synthesis

In Chapter 6 we presented two safety synthesis algorithms that are based on the counter-example guided synthesis approach, and interleave symbolic model checking with the synthesis of candidate solutions. These two algorithms target monolithic systems, however, we believe that they can also be used for parameterized systems.

Algorithm 4, in Page 96, is SAT-based, and therefore it encodes error paths computed by the model checker into a set of constraints that guarantees that any synthesized candidate must respect these constraints. One of the advantages of this algorithm is that it can be used for concurrent systems. Algorithm 4, in addition to the non-deterministic system and the set of error states, takes as input the so-called *transition relation constraint* ($TrConstr$). Therefore, if the system we want to synthesize is a monolithic representation of a concurrent system, then we can encode the totality of each component inside $TrConstr$ and pass it as a parameter to the algorithm.

Unfortunately, Algorithm 4 does not scale due to the fact that it explores error paths in an explicit manner.

Algorithm 5, in Page 100, is BDD-based and uses a forward/backward procedure to search for candidate solutions. This technique allows us to detect small subsets of states that are sufficient to define a candidate solution. Therefore, it can solve certain classes of problems more efficiently than other approaches. Furthermore, due to the forward/backward exploration of error paths, Algorithm 5 avoids progress towards the error states as early as possible, and hence it can compute candidate solutions in which error states are ”far away” from the set of reachable states. Such solutions are desirable in systems that need to be tolerant to hardware faults or perturbations in the environment. We evaluated a prototype implementation of the BDD-based algorithm on the benchmark set of the Reactive Synthesis Competition (SYNTCOMP) 2017. The prototype implementation solved 8 benchmarks that have not been solved by any participant in SYNTCOMP 2017.

Additionally, we evaluated Algorithm 5 on a set of random benchmarks that were generated by our tool AIGEN. AIGEN is a tool for generating random symbolic transition systems. It takes as input the number of controllable variables, the number of uncontrollable variables, and the number of latches, and generates an AIGER benchmark based on a uniformly random distribution over all controllable systems with these parameters. As an intermediate representation of the symbolic transition system to be generated, AIGEN uses ROBDDs in order to guarantee canonicity (see Chapter 7).

8.2 Future Work

Parameterized Verification In Chapter 3, we have shown that for parameterized guarded protocols there exists a cutoff that is linear in the number of guards in the process template. These results make parameterized model checking easier for process templates in which the number of guards is smaller than the number of states. On the other hand, such results could be used to synthesize systems with a minimal number of guards, i.e. with minimal synchronization. Given a process template without guards, and a set of specifications, we can gradually increase the number of introduced guards, and check, in each step, if the new system satisfies the specification. Using our results, the complexity of the last step will increase gradually, and will reach the same complexity as previous results only when the number of guards will be equal to the number of states in the process template.

Furthermore, we improved existing results for local deadlock detection in conjunctive systems. However, although we showed that a cutoff for an arbitrary conjunctive system is at least quadratic in the size of the process template, we did not solve the general problem. We tend to believe that the problem is in general undecidable, however it remains open, and therefore, it will be considered in future works.

As shown in Table 8.1, we have no results for specifications in Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ under global bounded fairness. In addition to these missing results, we want to check in future work if we can compute a maximal bound on the eventuality operator for parameterized systems. Kupferman et al. [76] showed that if a Prompt-$\mathsf{LTL}\backslash\mathbf{X}$ formula is satisfied in a monolithic system S , then it is satisfied with bound b, where b is exponential in the size of the formula and polynomial in S. For parameterized systems, when a cutoff exists, we would like to check if there exists a bound in terms of the cutoff size instead of the system size.

Parameterized Repair and Synthesis The repair algorithm in Chapter 5 uses the counter-example guided approach. In a future work we would like to use instead the attractor approach where we start from the set of error states and compute states that cannot avoid it until we reach a fixed point.

Additionally, in Section 5.3.3, we presented an algorithm for global deadlock detection in disjunctive system. This algorithm is used in the repair algorithm in order to check if candidate solutions introduce global deadlocks. In a future work, we would like to develop similar deadlock detection algorithms for other type of systems, like pairwise-rendezvous and broadcast protocols.

Table 8.1: A summary for the cutoff results of Chapters 3 and 4.

		Disjunctive Systems	Conjunctive Systems
k-indexed LTL$\backslash$**X**	non-fair	$\lvert\mathcal{G}\rvert + k + 1$ & $\lvert B_{\mathcal{G}}\rvert + k + 1$	$k + 1$
k-indexed LTL$\backslash$**X**	fair	$\lvert B_{\mathcal{G}}\rvert + \lvert\mathcal{G}\rvert + k$ & $2\lvert B_{\mathcal{G}}\rvert + k$	$k + 1$
Local Deadlock	non-fair	$m + \lvert\mathcal{G}\rvert + 1$, $m < \lvert B\rvert$	$\lvert\mathcal{G}_{U,B}\rvert + 2^{*}$
Local Deadlock	fair	$\lvert B_{\mathcal{G}}\rvert + \lvert\mathcal{G}\rvert + 1$ & $2\lvert B_{\mathcal{G}}\rvert + 1$	$2\lvert\mathcal{G}_{U,B}\rvert^{**}$
Global Deadlock	-	$\lvert B\rvert + \lvert\mathcal{N}^{*}\rvert$, $\lvert\mathcal{N}^{*}\rvert < \lvert B\rvert$	$2\lvert B\rvert - k_{str}$
k-indexed LTL$\backslash$**X**	lb-fair	$\lvert B\rvert + min(\lvert\mathcal{G}\rvert, \lvert B\rvert) + k$	$k + 1$
k-indexed LTL$\backslash$**X**	gb-fair	$\lvert B\rvert + min(\lvert\mathcal{G}\rvert, \lvert B\rvert) + k$	$k + 1^{***}$
k-indexed Prompt-LTL$\backslash$**X**	lb-fair	$\lvert B\rvert + min(\lvert\mathcal{G}\rvert, \lvert B\rvert) + k$	$k + 1$
k-indexed Prompt-LTL$\backslash$**X**	gb-fair	-	$k + 1^{***}$
		Token Passing Systems	
k-indexed LTL$\backslash$**X**	lb-fair	$2k$	
k-indexed LTL$\backslash$**X**	gb-fair	$2k$	
k-indexed Prompt-LTL$\backslash$**X**	lb-fair	$2k$	
k-indexed Prompt-LTL$\backslash$**X**	gb-fair	$2k$	

* : systems need to have alternation-bounded local deadlocks (see Sect. 3.1.4)
** : systems need to be initializing and have alternation-bounded local deadlocks
*** : Restricted to bounded initializing processes (Section 4.3). $k_{str} = 2k_1 + 2k_2 + k_3$
where:
k_1: number of free states
k_2: number of non-blocking states that are not free
k_3: number of not self-blocking states that are not free or non-blocking
$\mathcal{N} = \{q \in Q_B \mid q \in \mathsf{Enable}_q\}$
$\mathcal{N}^{*}$ is the maximal subset of $\mathcal{N}$ such that $\forall q_i, q_j \in \mathcal{N}^{*} : q_i \notin \mathsf{Enable}_{q_j} \wedge q_j \notin \mathsf{Enable}_{q_i}$
lb-fair: Local Bounded Fairness
gb-fair: Global Bounded Fairness

Bibliography

[1] Gmpy. https://pypi.python.org/pypi/gmpy2/.

[2] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *International Journal on Software Tools for Technology Transfer*, 18(5):495–516, 2016.

[3] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 313–321. IEEE, 1996.

[4] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Monotonic abstraction in action. In *International Colloquium on Theoretical Aspects of Computing*, pages 50–65. Springer, 2008.

[5] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular model checking for ltl (mso). In *International Conference on Computer Aided Verification*, pages 348–360. Springer, 2004.

[6] Rajeev Alur, Kousha Etessami, Salvatore La Torre, and Doron A. Peled. Parametric temporal logic for "model measuring". *ACM Trans. Comput. Log.*, 2(3):388–407, 2001. doi:10.1145/377978.377990.

[7] B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR*, volume 8704 of *LNCS*, pages 109–124. Springer, 2014. doi:10.1007/978-3-662-44584-6_9.

[8] Paul C Attie, Kinan Dak Al Bab, and Mouhammad Sakr. Model and program repair via sat solving. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–25, 2017.

[9] Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. Tight cutoffs for guarded protocols with fairness. *CoRR*, abs/1505.03273, 2015. Extended version with full proofs. URL: http://arxiv.org/abs/1505.03273.

[10] Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. Tight cutoffs for guarded protocols with fairness. In *VMCAI*, volume 9583 of *LNCS*, pages 476–494. Springer, 2016. doi:10.1007/978-3-662-49122-5_23.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[12] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005). *J. Autom. Reasoning*, 35(4):373–390, 2005. doi:10.1007/s10817-006-9026-1.

[13] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 140221, http://www.eecs.berkeley.edu/~alanmi/abc/.

[14] Dirk Beyer. Competition on software verification - (SV-COMP). In *TACAS*, volume 7214 of *LNCS*, pages 504–524. Springer, 2012. doi:10.1007/978-3-642-28756-5_38.

[15] Armin Biere. *AIGER Format and Toolbox*. URL: http://fmv.jku.at/aiger/.

[16] Armin Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

[17] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Außerlechner, and Raphael Spörk. Synthesis of synchronization using uninterpreted functions. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 35–42. IEEE, 2014.

[18] Roderick Bloem, Swen Jacobs, and Ayrat Khalimov. Parameterized synthesis case study: AMBA AHB. In *SYNT*, volume 157 of *EPTCS*, pages 68–83, 2014. doi:10.4204/EPTCS.157.9.

[19] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015. doi:10.2200/S00658ED1V01Y201508DCT013.

[20] Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-based synthesis methods for safety specs. In *VMCAI*, volume 8318 of *LNCS*, pages 1–20. Springer, 2014.

[21] Michael Blondin and Mikhail Raskin. The complexity of reachability in affine vector addition systems with states. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 224–236, 2020.

[22] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000. doi:10.1007/10722167_31.

[23] Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin, and Ocan Sankur. AbsSynthe: abstract synthesis from succinct safety specifications. In *SYNT*, volume 157 of *EPTCS*, pages 100–116, 2014. doi:10.4204/EPTCS.157.11.

[24] Francesco Buccafurri, Thomas Eiter, Georg Gottlob, and Nicola Leone. Enhancing model checking in verification by ai techniques. *Artificial Intelligence*, 112(1-2):57–104, 1999.

[25] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969. doi:10.2307/1994916.

[26] Jerry R Burch, Edmund M Clarke, and David E Long. Representing circuits more efficiently in symbolic model checking.

[27] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, Keijo Heljanko, and Jason Baumgartner. Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:135–172, 2016.

[28] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, volume 3653 of *LNCS*, pages 66–80. Springer, 2005.

[29] Peter C. Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *IJCAI*, pages 331–340. Morgan Kaufmann, 1991.

[30] Alonzo Church. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic*, I:3–50, 1957.

[31] E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *CONCUR*, volume 3170 of *LNCS*, pages 276–291. Springer, 2004. doi:10.1007/978-3-540-28644-8_18.

[32] E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI*, volume 3855 of *LNCS*, pages 126–141. Springer, 2006. doi:10.1007/11609773_9.

[33] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[34] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*, volume 10. Springer, 2018.

[35] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, 1989. doi:10.1007/3-540-52148-8_30.

[36] Eric Dallal, Daniel Neider, and Paulo Tabuada. Synthesis of safety controllers robust to unmodeled intermittent disturbances. In *CDC*, pages 7425–7430. IEEE, 2016. doi:10.1109/CDC.2016.7799416.

[37] Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010. doi:10.1007/978-3-642-15375-4_22.

[38] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 78–95, 2003.

[39] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[40] Rolf Drechsler and Bernd Becker. *Binary decision diagrams: theory and implementation*. Springer Science & Business Media, 2013.

[41] Rüdiger Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012. doi:10.1007/s10703-011-0137-x.

[42] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982. doi:10.1016/0167-6423(83)90017-5.

[43] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000. doi:10.1007/10721959_19.

[44] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210076.

[45] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. Principles of Programming Languages*, pages 85–94, 1995.

[46] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *Foundations of Computer Science*, 14(4):527–549, 2003. doi:10.1142/S0129054103001881.

[47] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV*, volume 1102 of *LNCS*, pages 87–98. Springer, 1996. doi:10.1007/3-540-61474-5_60.

[48] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782630.

[49] Javier Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In *STACS*, volume 25 of *LIPIcs*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.1.

[50] Kousha Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *International Conference on Computer Aided Verification*, pages 236–248. Springer, 1999.

[51] Peter Faymonville and Martin Zimmermann. Parametric linear dynamic logic. *Inf. Comput.*, 253:237–256, 2017. doi:10.1016/j.ic.2016.07.009.

[52] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011. doi:10.1007/s10703-011-0115-3.

[53] B. Finkbeiner and S. Jacobs. Lazy synthesis. In *VMCAI*, volume 7148 of *LNCS*, pages 219–234. Springer, 2012.

[54] B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013. doi:10.1007/s10009-012-0228-z.

[55] Bernd Finkbeiner and Swen Jacobs. Lazy synthesis. In *VMCAI*, volume 7148 of *LNCS*, pages 219–234. Springer, 2012. doi:10.1007/978-3-642-27940-9_15.

[56] Alain Finkel and Ph Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[57] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference (GECCO'09)*, pages 947–954. ACM, 2009.

[58] Hadar Frenkel, Orna Grumberg, Corina Pasareanu, and Sarai Sheinvald. Assume, guarantee or repair. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 211–227. Springer, 2020.

[59] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. doi:10.1145/146637.146681.

[60] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In *18th Conference on Computer Aided Verification (CAV'06)*, pages 358–371, 2006. LNCS 4144.

[61] S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10:1–29, 2014. doi:10.2168/LMCS-10(1:12)2014.

[62] Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur, and Leander Tentrup. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In *SYNT@CAV*, volume 260 of *EPTCS*, pages 116–143, 2017. doi:10.4204/EPTCS.260.10.

[63] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The first reactive synthesis competition (SYNTCOMP 2014). *STTT*, 19(3):367–390, 2017. doi:10.1007/s10009-016-0416-3.

[64] Swen Jacobs, Roderick Bloem, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Philipp J. Meyer, Thibaud Michaud, Mouhammad Sakr, Salomon Sickert, Leander Tentrup, and Adam Walker. The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. *CoRR*, abs/1904.07736, 2019.

[65] Swen Jacobs and Mouhammad Sakr. Analyzing guarded protocols: Better cutoffs, more systems, more expressivity. In *VMCAI*, volume 10747 of *Lecture Notes in Computer Science*, pages 247–268. Springer, 2018. doi:10.1007/978-3-319-73721-8_12.

[66] Swen Jacobs and Mouhammad Sakr. A symbolic algorithm for lazy synthesis of eager strategies. In *ATVA*, volume 11138 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2018. doi:10.1007/978-3-030-01090-4_13.

[67] Swen Jacobs, Mouhammad Sakr, and Martin Zimmermann. Promptness and bounded fairness in concurrent and parameterized systems. *CoRR*, abs/1911.03122, 2019. URL: http://arxiv.org/abs/1911.03122.

[68] Swen Jacobs, Leander Tentrup, and Martin Zimmermann. Distributed synthesis for parameterized temporal logics. *Information and Computation*, 262:311–328, 2018. doi:10.1016/j.ic.2018.09.009.

[69] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer, 2005. LNCS 3576.

[70] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*, December 2013. http://mpmath.org/.

[71] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 220–235. Springer, 2000.

[72] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413. Springer, 2009.

[73] A. Khalimov, S. Jacobs, and R. Bloem. PARTY parameterized synthesis of token rings. In *CAV*, volume 8044 of *LNCS*, pages 928–933. Springer, 2013. doi:10.1007/978-3-642-39799-8_66.

[74] A. Khalimov, S. Jacobs, and R. Bloem. Towards efficient parameterized synthesis. In *VMCAI*, volume 7737 of *LNCS*, pages 108–127. Springer, 2013. doi:10.1007/978-3-642-35873-9_9.

[75] Thomas Kropf. *Introduction to formal hardware verification.* Springer Science & Business Media, 2013.

[76] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. From liveness to promptness. *Formal Methods in System Design*, 34(2):83–103, 2009.

[77] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[78] Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. A SAT-based counterexample guided method for unbounded synthesis. In *CAV (2)*, volume 9780 of *LNCS*, pages 364–382. Springer, 2016. doi:10.1007/978-3-319-41540-6_20.

[79] Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *ICALP*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998. doi:10.1007/BFb0055040.

[80] Ernst W Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on computing*, 13(3):441–460, 1984.

[81] Jedidiah McClurg, Hossein Hojjat, and Pavol Černỳ. Synchronization synthesis for network programs. In *International Conference on Computer Aided Verification*, pages 301–321. Springer, 2017.

[82] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of SAT problems. In *AAAI*, pages 459–465. AAAI Press / The MIT Press, 1992. URL: http://www.aaai.org/Library/AAAI/1992/aaai92-071.php.

[83] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

[84] A. Pnueli and R. Rosner. Distributed systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society, 1990.

[85] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[86] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190. ACM Press, 1989. doi:10.1145/75277.75293.

[87] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97. Springer, 2001.

[88] Sylvain Schmitz and Philippe Schnoebelen. The power of well-structured systems. In *International Conference on Concurrency Theory*, pages 5–24. Springer, 2013.

[89] Saqib Sohail and Fabio Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *STTT*, 15(5-6):433–454, 2013. doi:10.1007/s10009-012-0224-3.

[90] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 136–148, 2008.

[91] Fabio Somenzi. CUDD: CU decision diagram package, release 2.4.0. *University of Colorado at Boulder*, 2009.

[92] Ichiro Suzuki. Proving properties of a ring of finite-state machines. *Information Processing Letters*, 28(4):213–214, 1988.

[93] Herve J Touati, Hamid Savoj, Bill Lin, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 130–133. IEEE, 1990.

[94] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331. IEEE Computer Society, 1986.

[95] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 327–338, 2010.

Appendix A

Glossary of symbols

A.1 Chapters 2,3,4, and 5

Q	a finite set of states.
Q_U	a the finite set of states of process U.
$\mathcal{G}$	$\mathcal{G} \subseteq \{\exists, \forall\} \times 2^Q$ is a set of guards.
$\mathcal{G}_U$	is the set of guards of process U.
δ_U	$\delta_U : Q_U \times \mathcal{P}(Q) \times Q_U$ is a guarded transition relation of process U.
$\lvert U \rvert$	$\lvert U \rvert = \lvert Q_U \rvert$ is the size of the state space of process U.
$\mathcal{P}$	is the set of all processes in a parameterized system.
$s(p)$	is the local state of process p in the global state s.
$s(p_1, \ldots, p_k)$	is the projection of s onto the processes $p_1, \ldots, p_k \in \mathcal{P}$.
k-conjunctive	is a guard of the form $Q \setminus q_1, \ldots, q_k$ for some $q_1, ..., q_k \in Q$
u-fair(x)	x is an unconditionally fair run.
b-gfair(x)	x is a globally b-bounded fair run. $\forall p \in \mathcal{P}\ \forall m \in \mathbb{N}\ \exists j \in \mathbb{N} : m \leq j \leq m + b$ and p moves at moment j.
b-lfair(x, E)	x is a locally b-bounded fair run if x is an unconditionally fair run and $\forall p \in E\ \forall m \in \mathbb{N}\ \exists j \in \mathbb{N} : m \leq j \leq m + b$ and p moves at moment j.
$\mathsf{appears}^{B_i}(q)$	is the set of all moments in x where process B_i is in state q: $\mathsf{appears}^{B_i}(q) = \{m \in \mathbb{N} \mid s_m(B_i) = q\}$.
$\mathsf{appears}(q)$	is the set of all moments in x where at least one B-process is in state q: $\mathsf{appears}(q) = \{m \in \mathbb{N} \mid \exists i \in \{1, \ldots, n\} : s_m(B_i) = q\}$.
f_q	is the first moment in x where q appears: $f_q = min(\mathsf{appears}(q))$
first_q	$\mathsf{first}_q \in \{1, \ldots, n\}$ is the index of a B-process where q appears first
l_q	$l_q = max(\mathsf{appears}(q))$ is the last moment where q appears
last_q	is a process index with $s_{l_q}(B_{\mathsf{last}_q}) = q$
$\mathsf{Visited}(x)$	is the set of B-states that appeared in the run x. $\mathsf{Visited}(x) = \{q \in Q_B \mid \mathsf{appears}(q) \neq \emptyset\}$.
$\mathsf{Visited}_F(x)$	is the set of states in $F \subseteq Q_B$ that appeared in the run x. Formally, $\mathsf{Visited}_F(x) = \{q \in \mathsf{Visited}(x) \mid q \in F\}$.
$\mathsf{Visited}^{inf}(x)$	is the set of B-states that appeared infinitely often in the run x. $\mathsf{Visited}^{inf}(x) = \{q \in Q_B \mid \exists B_i \in \{B_2, \ldots, B_n\} : \mathsf{appears}^{B_i}(q)$ is infinite$\}$.
$\mathsf{Visited}^{fin}(x)$	is the set of B-states that appeared finitely often in the run x. $\mathsf{Visited}^{fin}(x) = \{q \in Q_B \mid \forall B_i \in \{B_2, \ldots, B_n\} : \mathsf{appears}^{B_i}(q)$ is finite$\}$.
$Set(s_i)$	is the set of all state that are visited by some process at moment i:

	$\mathsf{Set}(s_i) = \{q \mid q \in (Q_A \cup Q_B)$ and $\exists p \in \mathcal{P} : s_i(p) = q\}$.
$\varphi(A, B_{i_1}, \ldots, B_{i_k})$	is a temporal logic formula over atomic propositions from Q_A and indexed propositions from $Q_B \times \{i_1, \ldots, i_k\}$.
$\varphi(A, B^{(k)})$	$\varphi(A, B^{(k)}) = \varphi(A, B_1, \ldots, B_k)$
deadset	a deadset of a local state q is a minimal set $D \subseteq Q$ that blocks all outgoing transitions of any process that is currently in local state q.
$dead^{\wedge}_q$	is the set of all deadsets of q.
non-blocking	a state q is non-blocking if it does not appear in $dead^{\wedge}_{q'}$ for any $q' \in Q$.
not self-blocking	a state q is not self-blocking if it does not appear in $dead^{\wedge}_q$.
maxD_U	is be the maximal number of states from B that appear in any deadset of a state in U. $\mathsf{maxD}_U = max\{\lvert D \cap Q_B \rvert \mid D \in dead^{\wedge}_q$ for some $q \in Q_U\}$.
$\mathcal{G}_{U,B}$	is the set of guards of U that exclude one of the states of B.
C_q	is a connected sequence of states $q, q_1, \ldots, q_n, q$, i.e. a cycle such that $\forall q_i, q_j \in \{q_1, \ldots, q_n\} : q_i \neq q_j$.
$\mathcal{G}_{U,B*}$	is the set of guards of U that exclude at least one state of B.
$\mathcal{G}_q$	is the set of non-trivial guards in transitions from q.
$\mathcal{G}_{lo}$	is the set of guards of the transitions on lo.
$B_{\mathcal{G}}$	is the set of B-states that appear on a guard.
Enable_q	is the set of states of A and B that enable a transition from q.
$\mathcal{N}$	$\mathcal{N} = \{q \in Q_B \mid q \in \mathsf{Enable}_q\}$
$\mathcal{N}^*$	is the maximal subset of $\mathcal{N}$ such that: $\forall q_i, q_j \in \mathcal{N}^* : q_i \notin \mathsf{Enable}_{q_j} \wedge q_j \notin \mathsf{Enable}_{q_i}$.
AP	is the set of atomic propositions.
block	is a finite word $w \in (2^{AP})^+$ where $\exists \alpha \subseteq AP$ such that $w = \alpha^{\lvert w \rvert}$.
$N_i^{\hat{w}}$	given an infinite sequence of blocks $\hat{w} = w_0, w_1, w_2 \ldots$, $N_i^{\hat{w}} = \{\sum_{l=0}^{i-1} \lvert w_l \rvert, \ldots, \sum_{l=0}^{i-1} \lvert w_l \rvert + \lvert w_i \rvert - 1\}$ be the set of positions of the ith block.
$\models_{gb}$	is the satisfies relation under global bounded fairness.
$\models_{lb}$	is the satisfies relation under local bounded fairness.
$A\|_{b}B^n$	is a bounded-fair system.
$\mathcal{A}_\varphi$	is the Büchi automaton that accepts exactly all words that satisfy φ.
$\mathcal{A}_{\overline{\varphi}}$	is the Büchi automaton that accepts exactly all words that satisfy $\neg\varphi$.
$\mathcal{G}_b^n(\varphi)$	is a run graph of a Büchi automaton $\mathcal{A}_\varphi$ on a system $A\|_{b}B^n$
B_q^{inf}	is a process that visits q infinitely often in a run x.
$\equiv_d$	is the d-stutter equivalence relation.
w^c	$w^c = w_0^c w_1^c \ldots \in (2^{AP \cup \{r\}})^\omega$ is an *r-coloring* of the word w.
alt_r	$alt_r = \mathbf{GF}r \wedge \mathbf{GF}\neg r$.
$rel_r(\varphi)$	is a formula obtained from φ by replacing every instance of $\mathbf{F_p}\psi$ by $(r \rightarrow (r\mathbf{U}(\neg r\mathbf{U}\psi))) \wedge (\neg r \rightarrow (\neg r\mathbf{U}(r\mathbf{U}\psi)))$.
$c(\varphi)$	$c(\varphi) = alt_r \wedge rel_r(\varphi)$ is an LTL formula.
$\bar{c}(\varphi)$	$\bar{c}(\varphi) = alt_r \wedge \neg rel_r(\varphi)$ is an LTL formula.
$\mathcal{P}_b^n(\varphi)$	is the colored Büchi graph that represents the product $A\|_{b}B^n \times \mathcal{A}_{\bar{c}(\varphi)}$.
$\overline{Q_T}$	is a finite non-empty set.
Q_T	$Q_T = \overline{Q_T} \times \{0, 1\}$ is a finite set of states where the Boolean component $\{0, 1\}$ indicates the possession of the token.
T_G^n	is a token passing system with an arbitrary number of T processes and a topology G.
ac	is an action in a token passing system.
t_U	is a transition of process U.

$\delta_U(q_U)$	is the set of all outgoing transitions of $q_U \in Q_U$.
$\vec{c}(i)$ or $\vec{c}(q_i)$	indicates how many processes are in state q_i.
$\vec{c}$	is the vector $(\vec{c}(q_0), \ldots, \vec{c}(q_{\lvert B \rvert -1})) \in \mathbb{N}_0^{\lvert B \rvert}$.
$\Delta^{local}(\sigma)$	is the set of all enabled outgoing local transitions from global counter system state σ.
$\Delta(\sigma, t_U)$	$\Delta(\sigma, t_U) = \sigma'$ if $\sigma \xrightarrow{t_U} \sigma'$.
$\uparrow R$	is the upward closure of the set R.
$minBasis(R)$	is a minimal basis of the upward closed set R.
$\lesssim$	is the component-wise ordering of vectors.
$\lessapprox$	$\lessapprox \subseteq \Omega \times \Omega$ is the binary relation defined by: $(q_A, \vec{c}) \lessapprox (q'_A, \vec{d}) \;\Leftrightarrow\; \Big(q_A = q'_A \wedge \vec{c} \lesssim \vec{d}\Big)$.
$pred(R)$	is the set of immediate predecessors of R $pred(R) = \{\sigma \in \Omega \mid \exists r \in R : \sigma \rightarrow r\}$.
$\lesssim_0$	$\lesssim_0 \subseteq \mathbb{N}_0^{\lvert B \rvert} \times \mathbb{N}_0^{\lvert B \rvert}$ where $\vec{c} \lesssim_0 \vec{d} \;\Leftrightarrow\; \Big(\vec{c} \lesssim \vec{d} \wedge \forall i \leq \lvert B \rvert : \big(\vec{c}(i) = 0 \Leftrightarrow \vec{d}(i) = 0\big)\Big)$.
$\lessapprox_0$	is the relation $\lessapprox_0 \subseteq \Omega \times \Omega$ where $(q_A, \vec{c}) \lessapprox_0 (q'_A, \vec{d}) \;\Leftrightarrow\; \Big(q_A = q'_A \wedge \vec{c} \lesssim_0 \vec{d}\Big)$.
$opred(R)$	is the O-predecessors of R, i.e. $opred(R) = \{\sigma \in S \mid \exists r \in R : \sigma \xrightarrow{t_U}{}^{+} r\}$.
$Succ(R)$	$Succ(R) = \{\sigma' \in \Omega \mid \exists \sigma \in R : \sigma \rightarrow \sigma'\}$.
$\Delta^{local}(\sigma, R)$	$\Delta^{local}(\sigma, R) = \{t_U \in \delta \mid t_U \in \Delta^{local}(\sigma) \wedge \Delta(\sigma, t_U) \in R\}$.
RE_i	is a reachable error level. $RE_i = Succ(RE_{i-1}) \cap \uparrow E_i$.
$\mathcal{RE}$	is the reachable error sequence. $\mathcal{RE} = RE_0, \ldots, RE_k$.
$\mathcal{UE}$	is the sequence $\uparrow RE_k, \ldots, \uparrow RE_0$.
$\mathcal{T}_{\mathcal{RE}}$	is a local witness of $\mathcal{RE}$, i.e., $\mathcal{T}_{\mathcal{RE}} = t_{U_k} \ldots t_{U_1}$ where for all $i \in \{1, \ldots, k\}$ there exists $\sigma \in RE_i, \sigma' \in RE_{i-1}$ with $\sigma \xrightarrow{t_{U_i}} \sigma'$.
$\mathcal{T}_{\mathcal{UE}}$	is a local witness of $\mathcal{UE}$.
$a!$	is a send action in a pairwise rendezvous system.
$a?$	is a send action in a pairwise rendezvous system.
$a!!$	is a send action in a broadcast system.
$a??$	is a send action in a broadcast system.
M^{PR}	is a pairwise rendezvous system.
M^{BC}	is a broadcast system.

A.2 Chapters 6, and 7

$\mathbb{B}$	is the set $\{0, 1\}$.
L	is a set of state variables for the latches.
X_u	is a set of uncontrollable input variables.
X_c	is a set of controllable input variables.
L'	is the set of state variables after the transition, $L' = \{l' \mid l \in L\}$.
$\mathcal{R}$	is the transition relation, $\mathcal{R} : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \times \mathbb{B}^{L'} \rightarrow \mathbb{B}$.
BAD	is the set of unsafe states, $BAD : \mathbb{B}^L \rightarrow \mathbb{B}$.
$F[x_i \leftarrow f_{x_i}]$	is the Boolean function that substitutes x_i by f_{x_i} in F.
$image(C)$	$image(C) = \{q' \in \mathbb{B}^{L'} \mid \exists (q, u, c) \in \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} : C(q) \wedge \mathcal{R}(q, u, c, q')\}$.
$preimage(C)$	$preimage(C) = \{q \in \mathbb{B}^L \mid \exists (u, c, q') \in \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \times \mathbb{B}^{L'} : C(q') \wedge \mathcal{R}(q, u, c, q')\}$.
$UPRE(C)$	$UPRE(C) = \{q \in \mathbb{B}^L \mid \exists u \in \mathbb{B}^{X_u}\ \forall c \in \mathbb{B}^{X_c}\ \exists q' \in \mathbb{B}^L : C(q') \wedge \mathcal{R}(q, u, c, q')\}$.
E_i	is an error level, i.e. a set of states that are on a path from q_0 to BAD, and all states in E_i are reachable from q_0 in i steps.
RT_i	for two error levels E_i and E_{i+1}, RT_i is the set of tuples, representing the "removable" transitions, i.e., all transitions from E_i to E_{i+1} that match an escape.
$image_f(C)$	$image_f(C) = \exists L\ \exists X_u\ \exists X_c\ (\bigwedge_{i=1}^{\lvert L \rvert} l'_i \equiv f_i \wedge C)$.

$preimage_f(C)$	$preimage_f(C) = \exists L' \; \exists X_u \; \exists X_c \; (\bigwedge_{i=1}^{\|L\|} l'_i \equiv f_i \wedge C')$.
$preimage_s(C)$	$preimage_s(C) = \exists X_u \; \exists X_c \; C[l_i \leftarrow f_i]_{l_i \in L}$.