Saarland University
Department of Computer Science

Master's Thesis

Parameterized Repair of Guarded Protocols for Liveness Properties

submitted by
Tom Simon Baumeister

submitted on
October 13th, 2022

Supervisor
PD Dr.-Ing. Swen Jacobs

Reviewers
PD Dr.-Ing. Swen Jacobs
Prof. Bernd Finkbeiner, Ph.D.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

_____ _____

(Ort / Place, Datum / Date) (Unterschrift / Signature)

Abstract

Concurrent systems that are composed of an arbitrary number n of processes, are hard to get correct. For these systems, parameterized model checking can provide correctness guarantees that hold regardless of n. However, model checking gives the designer no information about a possible repair when detecting an incorrect behaviour. The parameterized repair problem is, for a given implementation, to find a deadlock-free refinement such that a given property is satisfied by the resulting parameterized system. Recently, a parameterized repair approach was introduced, providing correctness guarantees for the repaired system that hold regardless of the number of processes. For safety properties, this approach can be applied on classes of systems, including disjunctive systems, pairwise rendezvous systems and broadcast protocols. However, it cannot guarantee liveness properties, e.g., termination or the absence of undesired loops.

This master's thesis presents a repair approach that provides parameterized correctness guarantees for liveness properties by modifying the existing repair algorithm. The approach generates minimal repairs such that only behavior of the system is restricted which violates the specification. Since it cannot repair parameterized systems if the specification requires additional nondeterminism or communication between processes, we introduce a paramaterized repair approach that generates minimal repair transformations. A minimal repair transformation changes the system by applying a set of operations that can introduce additional behavior including more communication if needed. We show that a repair transformation can be generated if and only if there exists a system satisfying the specification while preserving the given system structure. Furthermore, both algorithms are evaluated on a range of benchmarks.

Acknowledgment

I am very grateful to Dr.-Ing. Swen Jacobs for giving me the opportunity to work on such an interesting topic, fulfilling my research interests. Furthermore, I want to thank him for the guidance and help to complete this thesis. Another grateful thank to Dr.-Ing. Swen Jacobs and Prof. Bernd Finkbeiner for reviewing this thesis. Last but not least, I want to thank my family and friends, especially my parents Ute and Jürgen, and my brother Jan, for their ongoing support.

Contents

Chapter 1

Introduction

Concurrent systems are systems composed of independent components that perform operations concurrently and may communicate with each other. Since they are hard to get correct, they are a promising application area for formal methods. For paramaterized systems, i.e., systems that are composed of an arbitrary number of processes, parameterized model checking is able to provide correctness guarantees that hold regardless of the number of processes. If the paramaterized model checker detects a fault in the system, it returns a system execution that violates the given specification. However, it does not provide any information how to repair the system such that is satisfies the specification. Instead, the designer has to find out which behavior of the system causes the error. Then, the designer has to repair the faulty system for the found incorrect behavior. Since both tasks may be nontrivial, there is a need for a repair method that automatically returns a corrected parameterized system.

A parameterized repair approach was recently introduced by Jacobs et al. where parameterized systems are represented as guarded protocols [36]. For a given nondeterministic system, the repair approach restricts nondeterminism to eliminate faults in the internal behavior of a process. For a system that is known to be incorrect, this nondeterminism may have been added by the designer to initiate possible repairs. To repair the communication between processes, the repair approach selects the right options out of a set of possible interactions. Furthermore, the approach includes a deadlock detection, to avoid repairs that introduce deadlocks. This is essential, since often the easiest way to "repair" incorrect behavior is to let the system run into a deadlock as soon as possible. While the repair approach provides parameterized correctness guarantees for safety properties, i.e., the repaired implementation is correct for any number of processes, the approach cannot guarantee any liveness properties such as termination or the absence of undesired loops. Furthermore, the repaired system is not guaranteed to only restrict behavior of the faulty system that causes incorrect executions. However, the designer may be interested in repaired systems that preserve as much communication between the processes as possible. Moreover, if a given

paramaterized system cannot be repaired, the approach offers no additional feedback about additional required behavior of the system. Instead, the designer has to add more nondeterminism for more communication, and run the repair algorithm again. Since this may be a non-trivial and exhausting task, there is a need for a parameterized repair approach that can automatically introduce more communication between processes.

This master's thesis introduces a minimal repair approach that provides parameterized correctness guarantees for liveness properties. A minimal repair ensures to only restrict behavior of the system that causes an incorrect execution. Our approach modifies the existing algorithm to generate refinements that only restricts incorrect behavior of the system. Moreover, we introduce a minimal repair approach that can automatically add new behavior including more communication while preserving correctness guarantees. This approach is inspired by techniques used for the explainable synthesis approach [10] and is based on repair transformations that can apply different operations to add more communication if needed. Thereby, the approach guarantees to repair any implementation if and only if there exists a system preserving the structure of the implementation that satisfies the specification. Both approaches are implemented as an extension to the synthesis tool **BoSy** [27] and are evaluated on a range of benchmarks.

This thesis is structured as follows. Chapter 2 gives the necessary background information and definitions for the upcoming constructions and repair algorithms. The parameterized repair approach for safety properties, introduced by Jacobs et al. [36], is presented in Chapter 3. The modified repair approach that minimally repairs parameterized systems for liveness properties is introduced in Chapter 4. Chapter 5 presents the minimal repair approach that can automatically add more communication. The prototype implementation is shown and evaluated in Chapter 6. Related Work is discussed in Chapter 7 and we conclude this thesis in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we introduce the system model, consisting of process templates and guarded protocols. Furthermore, we recap the specification language LTL used to formulate safety and liveness properties, and some automata that are essential for upcoming repair constructions. Last, we present a model-checking approach for finite state system and properties expresses as LTL-formulas.

2.1 System Model

In this thesis, we represent parameterized systems as guarded protocols, consisting of multiple process templates. The definitions are taken from Jacobs and Sakr [35].

Let Q be finite set of states.

Definition 2.1.1. *(Process Template)*
A process template *is a transition system* $U = (Q_U, \mathsf{init}_U, \delta_U)$ *with*

- Q_U *is a finite set of states, including the initial state* init_U
- $\delta_U : Q_U \times \mathcal{P}(Q) \times Q_U$ *is a guarded transition relation.*

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

Definition 2.1.2. *(Guarded Protocol)*
Fix process templates A and B. A guarded protocol *is a system $A||B^n$, consisting of one A-process and n B-processes in an interleaving parallel composition.*

We assume that $Q = Q_A \dot{\cup} Q_B$, i.e., process templates A and B have disjoints set of states. Different B-processes 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 A is denoted by q_A. We denote the set of $\{A, B_1, \ldots, B_n\}$ as $\mathcal{P}$, and write p for a process in

(a) Writer (b) Reader

Figure 2.1: Two process templates

$\mathcal{P}$. For $U = \{A, B\}$, we write G_U for the set of non-trivial guards, i.e., guards different from Q and $\emptyset$. Then, let $G = G_A \cup G_B$.

Definition 2.1.3. *(Disjunctive and Conjunctive Guards)*
In a guarded protocol $A||B^n$, a local transition $(q_p, g, q'_p) \in \delta_U$ of p is enabled in s if its guard g is satisfied for p in s, written $(s, p) \vDash g$. There are two types of guarded protocols, depending on their interpretation of guards:

$$\textit{In } \text{disjunctive } \textit{systems: } (s,p) \vDash g \quad \textit{iff} \quad \exists p' \in \mathcal{P}\backslash\{p\} : g'_p \in G.$$
$$\textit{In } \text{conjunctive } \textit{systems: } (s,p) \vDash g \quad \textit{iff} \quad \forall p' \in \mathcal{P}\backslash\{p\} : g'_p \in G.$$

Let $\mathsf{set}(s) = \{q_A, q_{B_1}, \ldots, q_{B_n}\}$, and for a set of processes $P = \{p_1, \ldots, p_k\}$, $\mathsf{set}_P(s) = \{q_{p_1}, \ldots, q_{p_k}\}$. Then for disjunctive systems, we can more succinctly state that $(s, p) \vDash g$ iff $\mathsf{set}_{\mathcal{P}\backslash p}(s) \cap g \neq \emptyset$, and for conjunctive systems $(s, p) \vDash g$ iff $\mathsf{set}_{\mathcal{P}\backslash p}(s) \subseteq g$. A process is *enabled* in s if at least one of its transitions is enabled in s, otherwise it is *disabled*. Intuitively, for disjunctive systems a transition with a guard g is enabled if there *exists* another process that is currently in one of the states of g. In contrast, for conjunctive systems a transition with a guard g is enabled if *every* other process is currently in one of the states of g.

Example 2.1.1. *Consider the process templates depicted in Figure* 2.1*, taken from* [36]*. Throughout this thesis, the examples consider guarded protocols $A||B^n$, where A is the writer and B is the reader. The guards of the transitions determine which transitions can be taken by a process, depending on its own state and the state of other processes. Transitions with the trivial guard $g = Q$ are displayed without a guard since they are always enabled.*

In this thesis, 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 $Q\backslash\{q_1, \ldots, q_k\}$ for some $q_1, \ldots, q_k \in Q$. A state q is *k-conjunctive* if all non-trivial guards of transitions of q are k'-conjunctive with $k' \leq k$. A conjunctive system is *k-conjunctive* if every state is k-conjunctive.

Then, $A||B^n$ is defined as the transition system $(S, \mathsf{init}_S, \Delta)$ with

- set of global states $S = (Q_A) \times (Q_B)^n$,

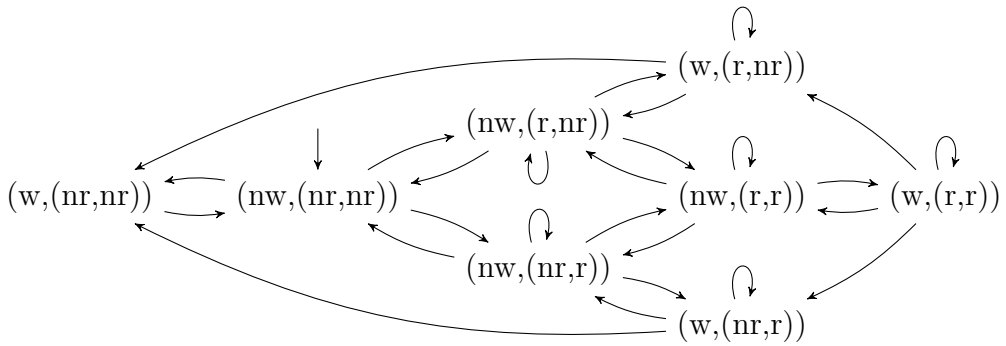


Figure 2.2: State space of the disjunctive system consisting of one writer process and two reader processes

- global initial state $\mathsf{init}_S = (\mathsf{init}_A, \mathsf{init}_B, \ldots, \mathsf{init}_B)$,
- and global transition relation $\Delta \subseteq S \times S$ with $(s, s') \in \Delta$ iff s' is obtained from $s = (q_A, q_{B_1}, \ldots, q_{B_n})$ by replacing one local state q_p with a new local state q'_p, where p is a U-process with local transition $(q_p, g, q'_p) \in \delta_U$ and $(s, p) \vDash g$.

A *path* of a system is a sequence of states $x = s_1, s_2, \ldots$ 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_m. We say that process p_m *moves* at *moment* m. A path can be finite or infinite, and a *maximal* path is a path that cannot be extended, i.e., it is either infinite or ends in a state where no transition is enabled.

Definition 2.1.4. *(Run)*
A system run *of a guarded protocol $A||B^n$ is a maximal path starting in the initial state* init_S.

We say that a run is *initializing* if every process p that moves infinitely often, visits its local initial state init_p infinitely often. The set of runs of a guarded protocol $A||B^n$ is called the language of $A||B^n$, denoted by $\mathcal{L}(A||B^n)$.

Given a system path $x = s_1, s_2, \ldots$ and a process p, the *local path* of p in x is the projection $x(p) = s_1(p), s_2(p), \ldots$ of x onto local states of p. A local path $x(p)$ is a *local run*, if x is a run.

A run is *globally deadlocked* if it is finite. An infinite run is *locally deadlocked* for process p if there exists m such that p is disabled for all $s_{m'}$ with $m' \geq m$. A run is *deadlocked* if it is locally or globally deadlocked. A system has a *(local/global) deadlock* if it has a (locally/globally) deadlocked run. Note that absence of local deadlocks for all p implies absence of global deadlocks, but not the other way around.

Example 2.1.2. *Figure* 2.2 *depicts the state space for the disjunctive system $A||B^2$, where A is the writer and B is the reader process. Every global state stores*

the current position for each process. Initially, every local state starts in its initial state, i.e., $s_0 = (\mathrm{nw}, (\mathrm{nr}, \mathrm{nr}))$*. For* s_0 *the writer could access the writing state and reach the global state* $(\mathrm{w}, (\mathrm{nr}, \mathrm{nr}))$*. Alternatively, one of the reader processes can move in* s_0 *and enter the reading state, since initially the writer is in state* nw*. Thus, the global states* $(\mathrm{nw}, (\mathrm{r}, \mathrm{nr}))$ *and* $(\mathrm{nw}, (\mathrm{nr}, \mathrm{r}))$ *are possible successors for* s_0*. Furthermore, the global states* $(\mathrm{w}, (\mathrm{r}, \mathrm{nr})), (\mathrm{nw}, (\mathrm{r}, \mathrm{r})), (\mathrm{w}, (\mathrm{nr}, \mathrm{r}))$ *and* $(\mathrm{w}, (\mathrm{r}, \mathrm{r}))$ *are also reachable. All possible transitions of* $A||B^2$ *are depicted in Figure* 2.2*. The system is globally deadlock-free since there is no finite run. However, it has a local deadlock. For the run* $(\mathrm{nw}, (\mathrm{nr}, \mathrm{nr})), (\mathrm{nw}, (\mathrm{r}, \mathrm{nr})), ((\mathrm{w}, (\mathrm{r}, \mathrm{nr})))^\omega$*, the second reader process is disabled in its local state* nr *when reaching the global state* $(\mathrm{w}, (\mathrm{r}, \mathrm{nr}))$*. Since the run stays in this global state forever, the run is locally deadlocked.*

2.2 LTL

In this thesis, the specifications are formulized in linear-time temporal logic (LTL). In this section, we define the syntax and semantics of the specification language, with definitions taken from Baier and Katoen [8].

2.2.1 Syntax

Definition 2.2.1. *(Syntax of LTL)*
LTL-formulas over a set of atomic propositions AP are built according to the following grammar:

$$\varphi ::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2,$$

where $a \in AP$*.*

The operators are defined as follows:

- $\neg$ is a unary operator, called *negation*
- $\wedge$ is a binary operator, called *conjunction*
- $\bigcirc$ is a unary operator, called *next*
- $\mathcal{U}$ is a binary operator, called *until*

LTL is defined over two different types of operators, boolean connectives and temporal operators. Boolean connectives are also known from other logics, e.g. predicate logics.

Using the boolean connectives $\neg$ and $\wedge$, we can derive some other basic boolean connectives:

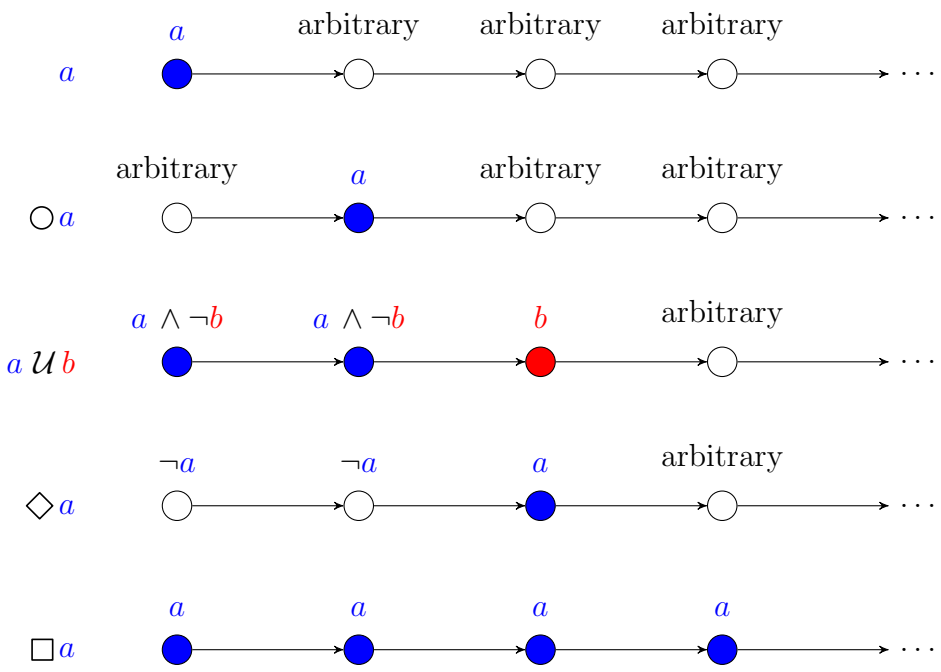


Figure 2.3: An intuitive semantics of temporal modalitites

- $\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2)$ is a binary operator, called *disjunction*
- $\varphi_1 \rightarrow \varphi_2 := \neg\varphi_1 \vee \varphi_2$ is a binary operator, called *implication*
- $\varphi_1 \leftrightarrow \varphi_2 := (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$ is a binary operator, called *equivalence*

The next and until operator are the basic temporal operators. They are extended by the following abbreviations:

- $\Diamond\varphi := true\,\mathcal{U}\,\varphi$ is a unary operator, called *eventually*
- $\Box\varphi := \neg(\Diamond\neg\varphi)$ is a unary operator, called *globally*
- $\varphi_1\,\mathcal{W}\,\varphi_2 := (\varphi_1\,\mathcal{U}\,\varphi_2) \vee \Box\varphi_1$ is a binary operator, called *weak until*

2.2.2 Semantics

The semantics of LTL is defined over infinite words, i.e., over infinite sequences. In this section, we introduce the semantics and formulate when an infinite words satisfies an LTL-formula. An infinite word is defined as follows:

Definition 2.2.2. *(Infinite Word)*
An infinite word *or* ω*-word on an alphabet* Σ *is a sequence* $\sigma : \mathbb{N} \rightarrow \Sigma$.

The LTL-syntax allows an LTL-formula to contain boolean connectives and temporal operators. In Figure 2.3 an intuition for the semantics of the temporal operators is depicted. The first column consists of an LTL-formula and the following graph represents an infinite word that satisfies the corresponding formula. Therefore, each state signals the variable assignment at this time step. If the given LTL-formula consists only of one atomic proposition a, obviously every infinite sequence σ, where a holds initially, satisfies the LTL-formula. For the LTL-formula $\bigcirc a$, every infinite sequence, where a holds at the second position, is accepted. $a \mathcal{U} b$ accepts an infinite word if there is a position j such that a holds for every position i with $0 \leq i < j$ and b holds at position j. The formula $\diamondsuit a$ accepts an infinite sequence σ if there is a position i where a eventually holds. The ω-word where a holds in every time step, is the only one that satisfies the LTL-formula $\square a$. The formal definition of the LTL-semantics is defined as follows.

Definition 2.2.3. *(Semantics of LTL)*
Let φ be an LTL-formula over AP. The LT-property induced by φ is

$$\mathcal{L}(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \vDash \varphi\},$$

where the satisfaction relation $\vDash \subseteq (2^{AP})^\omega \times LTL$ is the smallest relation satisfying:

$\sigma \vDash true$	
$\sigma \vDash a$	*iff* $a \in \sigma(0)$ *(i.e.* $\sigma(0) \vDash a$*)*
$\sigma \vDash \neg\varphi$	*iff* $\sigma \nvDash \varphi$
$\sigma \vDash \varphi_1 \wedge \varphi_2$	*iff* $\sigma \vDash \varphi_1$ *and* $\sigma \vDash \varphi_2$
$\sigma \vDash \bigcirc \varphi$	*iff* $\sigma[1, \infty] = \sigma(1)\sigma(2)\sigma(3) \cdots \vDash \varphi$
$\sigma \vDash \varphi_1 \mathcal{U} \varphi_2$	*iff* $\exists j \geq 0. \sigma[j, \infty] \vDash \varphi_2$ *and* $\sigma[i, \infty] \vDash \varphi_1$ *for all* $0 \leq i < j$

We say that a sequence $\sigma \in (2^{AP})^\omega$ is a *model* of an LTL-formula φ if $\sigma \vDash \varphi$. By $\mathcal{L}(\varphi)$ we denote the language of φ, i.e., $\mathcal{L}(\varphi)$ is the set of sequences $\sigma \in (2^{AP})^\omega$ that model φ. We distinguish between *safety* and *liveness* languages. Intuitively, a safety property requires that "something bad will never happen", whereas a liveness property states that "something good will eventually occur". Thus, a safety property is violated if there exists a finite sequence violating the specification.

Definition 2.2.4. *(Bad-prefix)*
A finite word $w \in \{1, \ldots, i\} \to \Sigma$ over some finite alphabet Σ is called a bad-prefix *for a language $L \subseteq \Sigma^\omega$, if every infinite word $\sigma \in (\mathbb{N} \to \Sigma)$ with prefix w is not in the language L.*

Definition 2.2.5. *(Safety Language)*
A language $L \subseteq (\mathbb{N} \to \Sigma)$ over some finite alphabet Σ is called a safety language, *if every $\sigma \notin L$ has a bad-prefix.*

We denote the set of bad-prefixes for a language L by $BP(L)$. In contrast, for liveness languages there exists no bad-prefix.

Definition 2.2.6. *(Liveness Language)*
A language $L \subseteq (\mathbb{N} \to \Sigma)$ over some finite alphabet Σ is called a liveness language, *if for every finite word $w \in \{1, \ldots, n\} \to \Sigma$, there exists an infinite word $\sigma \in (\mathbb{N} \to \Sigma)$ with prefix w such that $\sigma \in L$.*

The following theorem states that every linear time property, can be decomposed into a safety and liveness property.

Theorem 2.2.1. [8] *(Decomposition)*
For any LT-property φ over some finite alphabet Σ, there exists a safety property φ_{safe} over Σ and a liveness property $\varphi_{liveness}$ over Σ such that:

$$\mathcal{L}(\varphi) = \mathcal{L}(\varphi_{safe}) \cap \mathcal{L}(\varphi_{liveness})$$

In this thesis, we consider liveness properties, expressed as formulas in LTL\X, i.e., LTL formulas without the next operator $\bigcirc$. For a guarded protocol, defined over process templates A and B, we consider an LTL\X formula φ over atomic propositions from Q_A and indexed propositions from $Q_B \times \{1, \ldots, k\}$. We call φ a *paramaterized specification*. The satisfaction relation for a given parameterized specification and a guarded protocol is defined as follows.

Definition 2.2.7. *(Satisfaction Relation for Guarded Protocols)*
Given a guarded protocol $A||B^n$ and a parameterized specification φ over atomic propositions from Q_A and indexed propositions from $Q_B \times \{1, \ldots, k\}$ with $k \leq n$, we say that $A||B^n$ is a model of φ, denoted by $A||B^n \vDash \varphi$, iff $\mathcal{L}(A||B^n) \subseteq \mathcal{L}(\varphi)$.

Thus, a guarded protocol $A||B^n$ satisfies a parameterized specification φ if every trace of $A||B^n$ satisfies φ. A parameterized specification φ defined over k B-processes for process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$ is *realizable* if there exist $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_A, \delta'_B)$ such that $A'||B'^k \vDash \varphi$.

Example 2.2.1. *Consider the following specifications over atomic propositions from $Q_A = \{\mathrm{nw}, \mathrm{w}\}$ and $Q_B \times \{1, 2\} = \{\mathrm{nr}, \mathrm{r}\} \times \{1, 2\}$, i.e., states from the process templates in Figure 2.1. The specification $\varphi_{\mathrm{mutex}} = \square(\mathrm{w} \to (\mathrm{nr}_1 \wedge \mathrm{nr}_2))$ ensures that none of the reader processes is in the reading state, while the writer is currently writing. Thus, φ_{mutex} is a safety property since a word violates the specification iff a writer is writing and at least one reader is reading at the same time. The specification $\varphi_{\mathrm{fair}} = \square(\mathrm{nr}_1 \to \diamondsuit \mathrm{r}_1) \wedge \square(\mathrm{nr}_2 \to \diamondsuit \mathrm{r}_2)$ guarantees that both reader processes will eventually enter the reading state whenever they are not reading. φ_{fair} is a liveness property since every finite word could be fixed by entering the reading state for both processes and remaining there forever.*

2.3 Automata

In this section, we recap the automata from Baier and Katoen [8] that are essential for upcoming algorithms in this thesis. We start by introducing automata on finite words. Afterwards, we present automata on infinite words, i.e., nondeterministic Büchi automata and universal co-Büchi automata.

2.3.1 Automata on Finite Words

Definition 2.3.1. *(Nondeterministic Finite Automaton)*
A nondeterministic finite automaton $\mathcal{A}$ *over a finite alphabet* Σ *is a tuple* $\mathcal{A} = (Q, q_0, \delta, F)$ *where*

- Q *is a finite set of states, including the initial state* q_0
- $\delta : Q \times 2^{\Sigma} \times Q$ *is a transition relation*
- $F \subseteq Q$ *is a set of accepting states*

The language of an automaton $\mathcal{A}$ on finite words, denoted by $\mathcal{L}(\mathcal{A})$, is the set of finite words that are accepted by the automaton. A *finite word* w is defined as

$$w = w_1, w_2, \ldots, w_n \in (2^{\Sigma})^*,$$

where w_i is the letter of w at position i.

Definition 2.3.2. *(Run)*
A run *of a finite word* $\sigma \in (2^{\Sigma})^*$ *on a nondeterministic finite automaton* $\mathcal{A} = (Q, q_0, \delta, F)$ *is a finite path* $q_0 q_1 \ldots q_n \in Q^*$ *where* q_0 *is the initial state and for all* $0 \leq i < n$ *it holds that* $(q_i, \sigma_i, q_{i+1}) \in \delta$.

Definition 2.3.3. *(Accepting Run)*
A run $q_0 q_1 \ldots q_n \in Q^*$ *on a nondeterministic finite automaton* $\mathcal{A} = (Q, q_0, \delta, F)$ *is* accepting *iff* $q_n \in F$.

Intuitively, a run is accepting if it ends in an accepting state. A finite word w is accepted by a finite nondeterministic automaton $\mathcal{A}$ iff there exists an accepting run of w on $\mathcal{A}$. We also say that w is contained in the language of $\mathcal{A}$, denoted by $w \in \mathcal{A}$.

The language accepted by a nondeterministic finite automaton constitutes a regular language [8]. Vice versa, for any regular language L, there exists a nondeterministic finite automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = L$. Hence, the class of regular languages agrees with the class of languages accepted by a nondeterministic automaton. For a safety property φ, the *bad-prefix automaton* $\mathcal{A}_{BP}$ for φ is a finite automaton where for every finite word w it holds that $w \in \mathcal{L}(\mathcal{A}_{BP})$ iff w is a bad-prefix for φ.

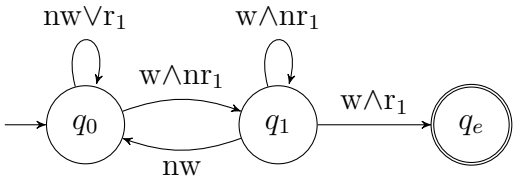


Figure 2.4: Bad-Prefix Automaton for $\varphi = \square((\mathrm{w} \wedge \mathrm{nr}_1) \rightarrow (\mathrm{nr}_1 \,\mathcal{W}\, \mathrm{nw}))$

Example 2.3.1. *Consider the finite automaton in Figure* 2.4 *and the specification* $\varphi = \square((w \wedge nr_1) \rightarrow (nr_1 \,\mathcal{W}\, nw))$. *Intuitively* φ *requires that whenever the writer is currently writing, the reader process is only allowed to start reading as soon as the writer process is done writing. The automaton accepts a finite word* w *if it reaches the accepting state* q_e. *An accepting run has to eventually reach* q_1 *which is only possible if the writer is writing while the reader process is not reading. Then,* q_e *is reached if the reader is entering the reading state while the writer is still writing. Thus, the automaton accepts a finite word if eventually the reader starts reading while the writer has not finished writing, i.e., the automaton accepts a finite word* w *iff* w *is a bad-prefix for* φ. *Hence, the the automaton is a bad-prefix automaton for* φ.

2.3.2 Automata on Infinite Words

Nondeterministic Büchi Automata

Definition 2.3.4. *(Nondetereministic Büchi Autamaton)*
A nondeterministic Büchi automaton $\mathcal{A}$ *over a finite alphabet* Σ *is a tuple* (Q, q_0, δ, F) *where*

- Q *is a finite set of states, including the initial state* q_0
- $\delta : Q \times 2^{\Sigma} \times Q$ *is a transition relation*
- $F \subseteq Q$ *is a set of accepting states.*

The language of an automaton $\mathcal{A}$ on infinite words, denoted by $\mathcal{L}(\mathcal{A})$, is the set of infinite words that are accepted by the automaton. An infinite word σ is defined as

$$\sigma = \sigma_0\sigma_1 \ldots \sigma_i \cdots \in (2^{\Sigma})^{\omega},$$

where σ_i is the letter of σ at position i.

Definition 2.3.5. *(Run)*
A run *of an infinite word* $\sigma \in (2^{\Sigma})^{\omega}$ *on a nondeterministic Büchi automaton* $\mathcal{A} = (Q, q_0, \delta, F)$ *is an infinite path* $q_0q_1q_2 \cdots \in Q^{\omega}$ *where* q_0 *is the initial state and for all* $i \geq 0$ *it holds that* $(q_i, \sigma_i, q_{i+1}) \in \delta$.

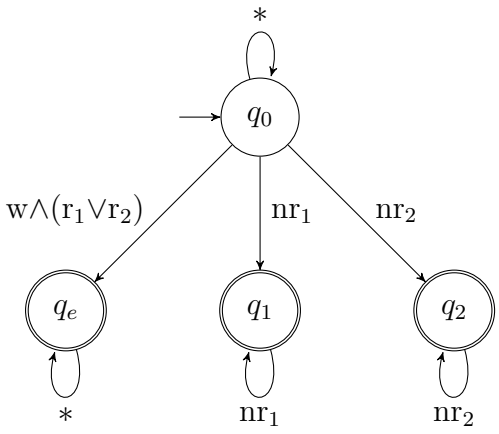


Figure 2.5: Representation of the nondeterministic Büchi Automaton used in Example 2.3.2 and the universal co-Büchi Automaton in Example 2.3.3

Definition 2.3.6. *(Accepting Run)*
A run $q_0q_1q_2\cdots \in Q^\omega$ on a nondeterministic Büchi automaton $\mathcal{A} = (Q, q_0, \delta, F)$ is accepting*, iff $q_i \in F$ holds for infinitely many i.*

Intuitively, a run is accepting if it visits an accepting state infinitely often. An infinite word σ is accepted by a nondeterministic Büchi automaton $\mathcal{A}$ iff there exists an accepting run of σ on $\mathcal{A}$. We also say that σ is contained in the language of $\mathcal{A}$, denoted by $\sigma \in \mathcal{L}(\mathcal{A})$.

Example 2.3.2. *Consider the nondeterministic Büchi automaton over the input alphabet $\Sigma = \{\mathrm{nw}, \mathrm{w}, \mathrm{nr}_1, \mathrm{nr}_2, \mathrm{r}_1, \mathrm{r}_2\}$ depicted in Figure* 2.5*. An infinite word σ is accepted if there exists a run for σ that remains in one of the accepting states forever, i.e., in q_e, q_1 or q_2. A run visits q_e infinitely often if* w *eventually holds while* r_1 *or* r_2 *hold at the same time. Thus, a word is accepted if it does not satisfy the formula φ_{mutex} from Example* 2.2.1*. Further, an accepting run visits q_1 or q_2 infinitely often if eventually* nr_1 *or* nr_2 *holds forever. Hence, a word is accepted if it does not satisfy φ_{fair} from Example* 2.2.1*. Thus, the nondeterministic Büchi automaton accepts an infinite word σ iff $\sigma \vDash \neg(\varphi_{\text{mutex}} \wedge \varphi_{\text{fair}})$.*

Universal co-Büchi Automata

Definition 2.3.7. *(Universal co-Büchi Automaton)*
A universal co-Büchi automaton *$\mathcal{A}$ over a finite alphabet Σ is a tuple (Q, q_0, δ, F), where*

- *Q is a finite set of states, including the initial state q_0*
- *$\delta : Q \times 2^\Sigma \times Q$ is a transition relation*

- $F \subseteq Q$ *is a set of rejecting states.*

Definition 2.3.8. *(Run)*
A run *of an infinite word* $\sigma \in (2^{\Sigma})^{\omega}$ *on a universal co-Büchi automaton* $\mathcal{A} = (Q, q_0, \delta, F)$ *is an infinite path* $q_0 q_1 q_2 \cdots \in Q^{\omega}$ *where* q_0 *is the initial state and for all* $i \geq 0$ *it holds that* $(q_i, \sigma_i, q_{i+1}) \in \delta$.

Definition 2.3.9. *(Accepting Run)*
A run $q_0 q_1 q_2 \in Q^{\omega}$ *on a universal co-Büchi automaton* $\mathcal{A} = (Q, q_0, \delta, F)$ *is* accepting, *iff* $q_i \in F$ *holds for finitely many* i.

Intuitively, a run is accepting if it visits every rejecting state only finitely often. An infinite word σ is accepted by a universal co-Büchi automaton iff every run of σ on $\mathcal{A}$ is accepting. We also say that σ is contained in the language of $\mathcal{A}$, denoted by $\sigma \in \mathcal{L}(\mathcal{A})$.

Example 2.3.3. *The automaton in Figure* 2.5 *can also be interpreted as a universal co-Büchi automaton. The automaton accepts a word* σ *iff every run of* σ *visits every rejecting state, i.e.,* q_e, q_1 *and* q_2*, finitely often. As mentioned in Example* 2.3.2*, a run for* σ *visits one of these states infinitely often if* σ *does not satisfy* φ_{mutex} *or* φ_{fair}*. Thus, a word* σ *is accepted by the universal co-Büchi automaton iff* $\sigma \vDash \varphi_{\text{mutex}} \wedge \varphi_{\text{fair}}$.

Büchi and co-Büchi are dual, i.e., a nondeterministic Büchi automaton can be complemented into a universal co-Büchi automaton and vice versa. Therefore, we switch the acceptance condition and dualize the transition function, i.e., nondeterministic transitions are interpreted universally and vice versa. For example, the nondeterministic Büchi automaton $\mathcal{A}$ in Example 2.3.2 accepts a word σ iff $\sigma \vDash \neg(\varphi_{\text{mutex}} \wedge \varphi_{\text{fair}})$. The automaton can be transformed into the universal co-Büchi automaton $\mathcal{A}'$ in Example 2.3.3 by switching the acceptance condition and dualizing the transition function. After complementing, $\mathcal{A}'$ recognizes the complement language $\mathcal{L}(\mathcal{A}') = \Sigma^{\omega} \backslash \mathcal{L}(\mathcal{A})$, i.e., $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\varphi_{\text{mutex}} \wedge \varphi_{\text{fair}})$.

The following two theorems are essential for upcoming algorithms. Theorem 2.3.1 states that every LTL-formula φ can be transformed into a nondeterministic Büchi automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$. Theorem 2.3.2 states that every nondeterministic Büchi automaton $\mathcal{A}$ can be transformed into a universal co-Büchi automaton $\mathcal{A}'$ with $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$, and vice versa.

Theorem 2.3.1. [38] *For a given LTL-formula* φ*, there is a nondeterministic Büchi automaton* $\mathcal{A}$ *with* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$*. The size of* $\mathcal{A}$ *is exponential in the length of* φ.

Theorem 2.3.2. [42] *Nondeterministic Büchi automata and universal co-Büchi automata are equivalent in expressive power, i.e., they recognize the same* ω*-languages.*

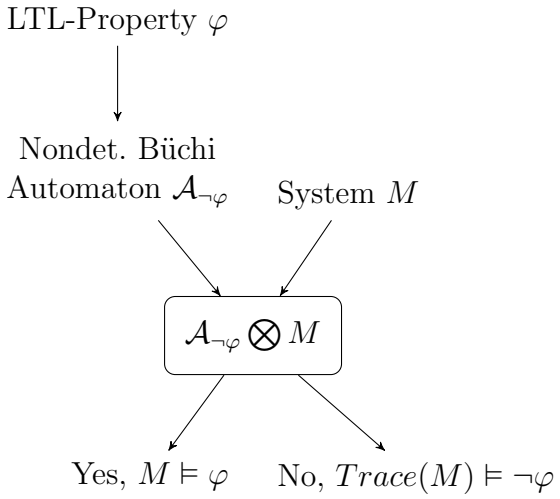


Figure 2.6: LTL-Model-Checking

2.4 Model-Checking

In this section, we explain the principles of model-checking. After presenting a method for model-checking general LTL-properties, we show a simplified approach to model-check safety properties from Baier and Kathoen [8]. This is essential for upcoming repair algorithms.

In Sections 2.1 and 2.2, we have described how guarded protocols can be represented as transition systems and how we can express general properties of a system as specifications, formulated in LTL. A given system model M models an LTL-formula φ if each trace of M satisfies φ, i.e. $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$. A model-checking algorithm checks whether a system models a given specification or not. One possibility of LTL-model-checking is depicted in Figure 2.6. The algorithm consists of the following steps:

1. Preprocessing: Formulate the specification as an LTL-formula φ and implement a system, represented by the transition system M.

2. Build a nondeterministic Büchi automaton $\mathcal{A}_{\neg\varphi}$ for the negated formula of φ, such that $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$, obtained by the construction in Theorem 2.3.1.

3. Construct the product $M \otimes \mathcal{A}_{\neg\varphi}$ of the system M and automaton $\mathcal{A}_{\neg\varphi}$.

4. Check if there is a trace of $M \otimes \mathcal{A}_{\neg\varphi}$ that simulates an accepting run in $\mathcal{A}_{\neg\varphi}$. If such a trace exists then $M \nvDash \varphi$, otherwise $M \vDash \varphi$.

The idea is that the model checking algorithm checks if there exists a trace of the system M that does not satisfy φ rather than checking if every trace of M

satisfies φ. If such a trace exists then $M \not\models \varphi$, otherwise $M \models \varphi$. The product of an automaton $\mathcal{A}$ and a system M is a transition system M' that simulates both $\mathcal{A}$ and M. So each state tells the position in $\mathcal{A}$ and M for a given sequence of inputs. Checking if there exists a trace that simulates an accepting run in $\mathcal{A}$ is done by performing a nested depth-first search on the product. Such a trace contains an infinite cycle that visits an accepting state. For safety properties, a violating trace only has to reach a "bad" state. Thus, a finite bad-prefix automaton is sufficient and instead of checking for an accepting cycle in the product, the algorithm checks for a finite execution that reaches an accepting state. The complexities of model-checking safety properties and LTL-model-checking are shown in the following theorems:

Theorem 2.4.1. [8] *The time and space complexity of checking a regular safety Property φ against a transition System M is in $\mathcal{O}(|M| \cdot |\mathcal{A}_{BP}|)$, where $\mathcal{A}_{BP}$ is a bad-prefix automaton for φ.*

Theorem 2.4.2. [8] *I The LTL-model-checking problem is* PSPACE*-complete.*

Chapter 3

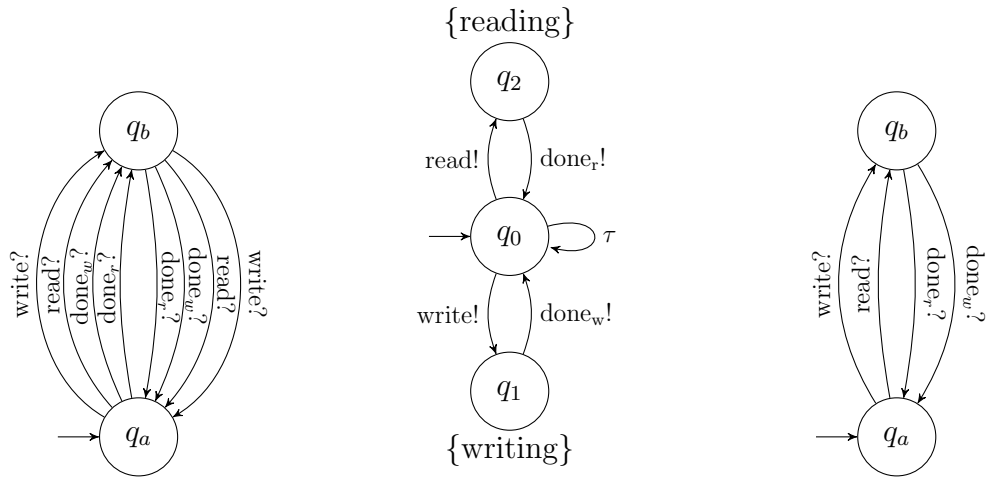
Parameterized Repair of Guarded Protocols for Safety Properties

In this chapter, we introduce the parameterized repair approach, introduced by Jacobs, Sakr and Völp [36]. Further, we present their parameterized model checking and deadlock detection algorithm. After showing their parameterized repair algorithm, we discuss the limitations of their approach. We start by giving a short motivation with examples from [36].

3.1 Motivation

Parameterized systems that are composed of an arbitrary number of processes, are hard to get correct. The parameterized model checking method is able to provide security guarantees that hold regardless of the number of processes. If the parameterized model checker detects a fault in the system, it returns a possible execution that violates a given specification. However, it does not give any information how the designer can repair the system or which behavior of the system causes the error. Since these tasks may be nontrivial, we are interested in a repair approach that automatically returns a correct implementation. Starting with a nondeterministic system, the repair algorithm restricts nondeterminism to eliminate faults in the internal behavior of a process. To repair the communication between processes, the approach selects the right options out of a set of possible interactions. Since the easiest way to avoid incorrect behavior is to let the system run into a deadlock as soon as possible, only repairs that do not introduce deadlocks, are generated. The resulting repaired implementation is a refinement such that all parameterized correctness guarantees hold regardless of the number of processes.

Consider the parameterized system in Figure 3.1, consisting of the unsafe



(a) Unsafe Scheduler (b) Reader-Writer (c) Safe Scheduler

Figure 3.1: Motivating Example

scheduler in Figure 3.1a and an arbitrary number of reader-writer processes, depicted in Figure 3.1b. The processes communicate via pairwise synchronisation such that a sending action (e.g. write!) can only proceed if another process executes a corresponding receive action (e.g. write?). For this system, global error states are reachable where multiple processes are in the writing-state at the same time. In the approach, we are interested in repairing the system by restricting the communication such that global error states are unreachable. One possibility to avoid error states where multiple processes are writing at the same time, is to remove all receiving actions for the scheduler in state q_a. However, then the system is globally deadlocked initially. Therefore, we are interested in repairs that are deadlock-free. Figure 3.1c shows a repair that is able to avoid global error states and does not introduce deadlocks.

3.2 Problem Statement

In this section, we formally define the problem statement. First, we show how parameterized systems can be represented as counter systems. Then, we define the parameterized repair problem and give a high-level explanation of the repair algorithm.

3.2.1 Counter System

In Section 2.1, process templates and an explicit representation of the global disjunctive system are represented. For fixed process templates A and B, a guarded protocol $A||B^n$ consists of one process A and n copies of process B. A

global state $s \in (Q_A) \times (Q_B)^n$ stores the current position of each local process. In the following, we show an alternative system representation that only counts the number of processes currently in q_B, for each local state q_B of process template B. Such a global state is called a *configuration*. In the following, we formalize this representation for disjunctive systems. The definitions are taken from [36].

Definition 3.2.1. *(Configuration)*
Fix process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$. A configuration *of a system $A||B^n$ is a tuple $(q_A, \mathbf{c})$, where $q_A \in Q_A$ and $\mathbf{c} : Q_B \to \mathbb{N}_0$.*

We identify $\mathbf{c}$ with the vector $(\mathbf{c}(q_0), \ldots, \mathbf{c}(q_{|B|-1})) \in \mathbb{N}_0^{|B|}$, and also use $\mathbf{c}(i)$ to refer to $\mathbf{c}(q_i)$. Intuitively. $\mathbf{c}(q_i)$ indicates how many processes are in state q_i. We denote by $\mathbf{u}_i$ the unit vector with $\mathbf{u}_i(i) = 1$ and $\mathbf{u}_i(j) = 0$ for $j \neq i$.

Example 3.2.1. *Consider the process templates from Figure* 2.1 *again, where A is the writer and B is the reader. Two possible configurations of the system $A||B^2$ are $s_1 = (\mathrm{nw}, (1, 1))$ and $s_2 = (\mathrm{w}, (0, 2))$. s_1 indicates that the A-process is currently in the local state* nw*, one B-process is in* nr *and one B-process is in the reading state* r*. For the configuration s_2, the A-process is currently in the writing state* w *while both B-process are in the reading state* r*.*

Given a configuration $s = (q_A, \mathbf{c})$, we say that a guard g of a local transition $(q_U, g, q'_U) \in \delta_U$ is *satisfied* in s, denoted by $s \vDash_{q_U} g$, 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 $\mathbf{c}(i) \geq 1$
(A takes the transition, a B-process is in g)

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

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

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

We also say that the local transition is *enabled* in s.

Definition 3.2.2. *(Configuration Space)*
Fix process templates A and B. The configuration space *of all systems $A||B^n$ for arbitrary $n \in \mathbb{N}$, is the transition system $M = (S, S_0, \Delta)$ where:*

- *$S \subseteq Q_A \times \mathbb{N}_0^{|B|}$ is the set of states,*

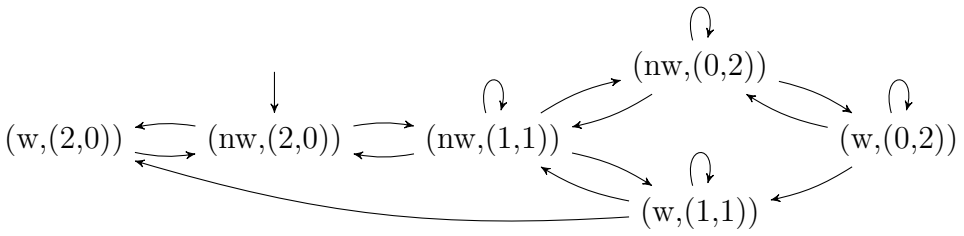


Figure 3.2: Configuration space of the system $A||B^2$, where A is the writer and B is the reader from Figure 2.1.

- $S_0 = \{(\mathsf{init}_A), \mathbf{c}) \mid \mathbf{c}(q) = 0 \text{ if } q \neq \mathsf{init}_B\}$ *is the set of initial states,*
- $\Delta \subseteq S \times S$ *is the set of transitions where* $((q_A, \mathbf{c}), (q'_A, \mathbf{c}')) \in \Delta$ *iff one of the following holds:*
  1. $\mathbf{c} = \mathbf{c}'$ *and* $\exists (q_A, g, q'_A) \in \delta_A$ *with* $(q_A, \mathbf{c}) \vDash_{q_A} g$
     (transition of the A-process)
  2. $q_A = q'_A$ *and* $\exists (q_i, g, q_j) \in \delta_B$ *with* $\mathbf{c}(i) \geq 1 \wedge \mathbf{c}' = \mathbf{c} = \mathbf{u}_i + \mathbf{u}_j$ *and* $(q_A, \mathbf{c}) \vDash_{q_i} g$
     (transition of a B-process)

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

Let $s, s' \in S$ be states of M and $U \in \{A, B\}$. For a transition $(s, s') \in \Delta$, we also write $s \to s'$. If the transition is based on local transition $t_U = (q_U, g, q'_U) \in \delta_U$, we also write $s \xrightarrow{t_U} s'$ or $s \xrightarrow{g} s'$. Let $\Delta^{\text{local}}(s) = \{t_U \mid \exists s' \in S : s \xrightarrow{t_U} s'\}$, i.e., the set of all enabled outgoing transitions from s, and let $\Delta(s, t_U) = s'$ if $s \xrightarrow{t_U} s'$. Note that in the remainder of the thesis, we assume wlog. that each guard g is a singleton. This is not a restriction as any local transition $(q_U, g, q'_U) \in \delta_U$ where $|g| > 1$ can be split into $|g|$ transitions $(q_U, g_1, q'_U), \ldots, (q_U, g_{|g|}, q'_U)$ where for all $i \leq |g| : q_i \in g$ is a singleton guard.

Analogously to Section 2.1, we can define a *path* and a *run* of a counter system. A *path* of a counter system is a (finite or infinite) sequence of states $x = s_1, s_2, \ldots$ such that $s_m \to s_{m+1}$ for all $m < |x|$. A *maximal path* is a path that cannot be extended, and a *run* is a maximal path starting in an initial state. We say that a run is *globally deadlocked* if it is finite. Note that every run $s_1, s_2, \ldots$ of the counter system corresponds to a run of a fixed $A||B^n$, i.e. the number of processes does not change during a run. Given a set of error states $ERR \subseteq S$, an *error path* is a finite path that starts in an initial state and ends in ERR.

Example 3.2.2. *Figure* 3.2 *shows the configuration space for one writer and two copies of the reader process. The initial configuration is* $s_0 = (\text{nw}, (2, 0))$,

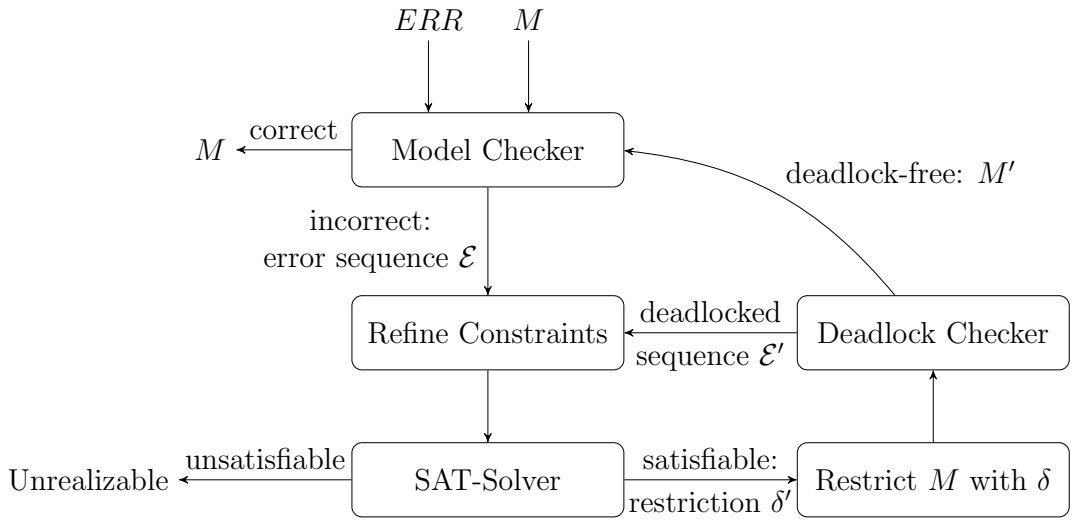


Figure 3.3: Parameterized repair of concurrent systems

i.e., none of the processes is writing or reading. In s_0 *the writer can either enter the writing state resulting in the global state* $s_1 = (\mathrm{w}, (2, 0))$*, or one of the reader processes can enter the reading state resulting in the global state* $s_2 = (\mathrm{nw}, (1, 1))$*. Furthermore, the configuration space includes the configurations* $(\mathrm{nw}, (0, 2))$*,* $(\mathrm{w}, (1, 1))$ *and* $(\mathrm{w}, (0, 2))$ *with the depicted transitions. The system is deadlock-free since it contains no globally deadlocked run.*

3.2.2 Parameterized Repair

In the following, we define the parameterized repair problem. Then, we show a high-level parameterized repair algorithm and discuss its challenges.

Problem 3.2.1. *(Parameterized Repair Problem)*
Let $M = (S, S_0, \Delta)$ *be the counter system for process templates* $A = (Q_A, \mathsf{init}_A, \delta_A)$*,* $B = (Q_B, \mathsf{init}_B, \delta_B)$*, and a set of error states* $ERR \subseteq Q_A \times \mathbb{N}_0^{|B|}$*. The* parameterized repair problem *is to decide if there exist process templates* $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ *with* $\delta'_A \subseteq \delta_A$ *and* $B' = (Q_B, \mathsf{init}_B, \delta'_B)$ *with* $\delta'_B \subseteq \delta_B$ *such that the counter system* M' *for* A' *and* B' *is globally deadlock-free and does not reach any state in* ERR*.*

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

Figure 3.3 gives an overview of the parameterized repair algorithm. For a given counter system M that is based on process templates A and B, the algorithm checks if any error state in ERR is reachable. If the model checker detects no reachable error state, the counter system M is already correct. Otherwise, the model checker returns an error sequence $\mathcal{E}$, i.e. one or more error paths that

start in an initial state in M and end in a state in ERR. Then, the algorithm refines constraints over A and B such that any error path in $\mathcal{E}$ is avoided. These constraints are forwarded to a SAT-solver to find refinements of A and B that satisfy the constraint system and avoid any error path in $\mathcal{E}$. The generated restriction $\delta' = \delta'_a \cup \delta'_b$ restricts the non-deterministic transition relations of A and B, i.e., it consists of subsets $\delta'_a \subseteq \delta_a$ and $\delta'_b \subseteq \delta_b$ of the local transition relations. Then, δ' is used to restrict M. If the SAT-solver is unable to find a restriction, then the counter system cannot be repaired for ERR. In the next step, the algorithm checks if the restriction δ' introduced any global deadlocks. The deadlock checker works similarly to the model checker by checking if any deadlocked state is reachable rather than an error state. If a deadlock can be detected, a deadlocked sequence $\mathcal{E}'$ is encoded into constraints to refine the constraint system. Otherwise the restricted counter system M' is sent to the model checker for the next iteration.

In Section 3.3, we show how the model checker generates error sequences that allow us to refine the constraint system to avoid any error path. Furthermore, we present how the parameterized deadlock checker supplies similar information as the model checker. In Section 3.4, we show how to encode the supplied information from model checking and deadlock detection into constraints such that the resulting restriction avoids any error path that has already been found. Further, we present the paramaterized repair algorithm.

3.3 Parameterized Model Checking

In this section, we show how we can efficiently model check counter systems. Before introducing a parameterized model checking algorithm, we present how counter systems can be framed as well-structured transition systems (WSTS). Last, we show how the algorithm can be modified to detect deadlocked states.

3.3.1 Counter Systems as WSTS

For a given set of error states ERR, a counter system M based on process templates A and B is correct, if for every n, $A||B^n$ reaches no state in ERR. The standard model checking method, shown in Section 2.4, performs a reachability analysis to check for finite executions that end in an error state. This method works for all finite systems. However, a counter system has an infinite state space since it contains the configuration space of all systems $A||B^n$. In Section 3.3.2, we present a parameterized model checking algorithm that works for counter systems. Therefore, we need to frame counter systems as WSTS. We start by introducing a *well-quasi-order*.

Definition 3.3.1. *(well-quasi-order)*
Given a set of states S, a binary relation $\preceq \subseteq S \times S$ is a well-quasi-order *(wqo) if $\preceq$ is reflexive, transitive, and if any infinite sequence $s_0, s_1, \ldots \in S^\omega$ contains a pair $s_i \preceq s_j$ with $i < j$.*

A subset $R \subseteq S$ is an *antichain* if any two distinct elements of R are incomparable wrt. $\preceq$. Therefore, $\preceq$ is a wqo on S iff it is well-founded and has no infinite antichain.

Example 3.3.1. *The ordering $\leq$ is a wqo on $\mathbb{N}$. However, $\leq$ is not a wqo on $\mathbb{Z}$, because the infinite sequence $s = 0, -1, -2, -3, \ldots$ is infinitely decreasing. For s, there is no pair $s_i \leq s_j$ with $i < j$.*

For well-quasi-orders, we can define the following property which helps us to find a finite basis for an infinite set:

Definition 3.3.2. *(upwards closure)*
Let $\preceq$ be a wqo on S. The upwards closure *of a set $R \subseteq S$, denoted $\uparrow R$, is the set $\{s \in S \mid \exists s' \in R : s' \preceq s\}$.*

We say that R is *upwards-closed* if $\uparrow R = R$. If R is upwards-closed, then we call $B \subseteq S$ a *basis* of R if $\uparrow B = R$. If $\preceq$ is also antisymmetric, then any basis of R has a unique subset of minimal elements. We call this set the *minimal basis* of R, denoted $minBasis(R)$.

Example 3.3.2. *Consider the wqo $\lesssim$ on $\mathbb{N}^2$ where $\lesssim$ is the component-wise ordering on vectors. For $R = \{(2,1),(1,3)\}$, the vector $(2,2)$ is in the upwards closure $\uparrow R$ since $(2,1) \lesssim (2,2)$, However, $(1,2) \notin \uparrow R$ since $(2,1) \not\lesssim (1,2)$ and $(1,3) \not\lesssim (1,2)$. A minimal basis for $\uparrow R$ is $\{(2,1),(1,3)\}$.*

In the following, we define if a wqo is compatible with a given system, i.e. if a wqo can be used to compare different states of the system. For parameterized systems, a compatible wqo can be used to compare states and their successors for systems of a different size. Further, this allows us to define a WSTS.

Definition 3.3.3. *(compatible)*
Given a counter system $M = (S, S_0, \Delta)$, a wqo $\preceq S \times S$ is compatible *with Δ iff the following holds:*

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

In Definition 3.3.3, $r \to^* r'$ denotes that r' is reachable from r by taking one or more transitions. We say that $\preceq$ is *strongly compatible* with Δ if the above holds with $r \to r'$ instead of $r \to^* r'$.

Definition 3.3.4. *(WSTS)*
For a given counter system $M = (S, S_0, \Delta)$, $(M, \preceq)$ is a well-structured transition system *if $\preceq$ is a wqo on S that is compatible with Δ.*

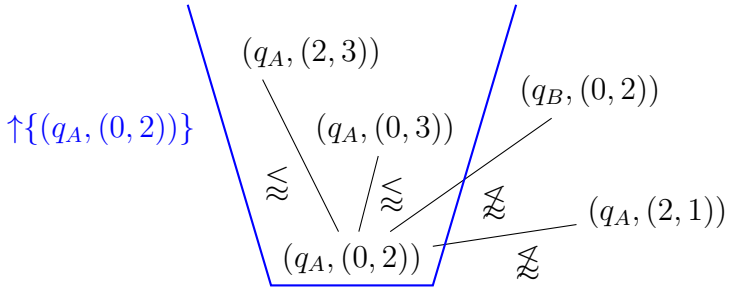


Figure 3.4: Counter Systems as WSTS with $\lessapprox$

The following lemma shows a wqo $\lessapprox$ that is compatible with a counter system M such that $(M, \lessapprox)$ is a WSTS:

Lemma 3.3.1. [36] *Let $M = (S, S_0, \Delta)$ be a counter system for process templates A, B, and let $\lessapprox \subseteq S \times S$ be the binary relation defined by:*

$$(q_A, \mathbf{c}) \lessapprox (q'_A, \mathbf{c}') \Leftrightarrow (q_A = q'_A \wedge \mathbf{c} \lesssim \mathbf{c}'),$$

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

Example 3.3.3. *Figure 3.4 illustrates the wqo from Lemma 3.3.1. For $R = \{(q_A, (0,2))\}$ the states $(q_A, (0,3))$ and $(q_A, (2,3))$ are in $\uparrow R$. However, the configuration $(q_B, (0,2))$ is not in the upwards closure of R since the local state for process A is in a different state. Further, $(q_A, (2,1))$ is not in $\uparrow R$ because $(0,2) \not\lesssim (2,1)$.*

By framing a counter system as a WSTS, we can represent and compare states of different systems size. Further, the upwards closure of an infinite set of states can be represented by a finite minimal basis. With this representation, we can compute the predecessor of an upwards closure. This is essential for the model checking algorithm.

Definition 3.3.5. *(Predecessor)*
Let $M = (S, S_0, \Delta)$ be a counter system and let $R \subseteq S$. Then the set of immediate predecessors *of R is*

$$\mathit{pred}(R) = \{s \in S \mid \exists r \in R : s \to r\}.$$

A WSTS $(M, \lessapprox)$ has *effective pred-basis* if there exists an algorithm that takes as input any finite set $R \subseteq S$ and returns a finite basis of $\uparrow \mathit{pred}(\uparrow R)$. For a given set $R \subseteq S$ that is upwards-closed with respect to $\lessapprox$, $\mathit{pred}(R)$ is upwards-closed iff $\lessapprox$ is strongly compatible with Δ.

Algorithm 1 Parameterized Model Checking

```
1:  procedure ModelCheck(M, ERR)
2:      temporarySet ← ERR, E_0 ← ERR, i ← 1, visistedSet ← ∅
3:      //If temporarySet = visitedSet then a fixed point is reached
4:      while temporarySet ≠ visitedSet do
5:          visitedSet ← temporarySet
6:          E_i ← minBasis(↑pred(↑E_{i−1}))
7:          //Check intersection with initial states
8:          if E_i ∩ S_0 ≠ ∅ then
9:              return (False,{E_0, ..., E_1 ∩ S_0})
10:         temporarySet ← minBasis(visitedSet ∪ E_i)
11:         i ← i + 1
12:     return (True, ∅)
```

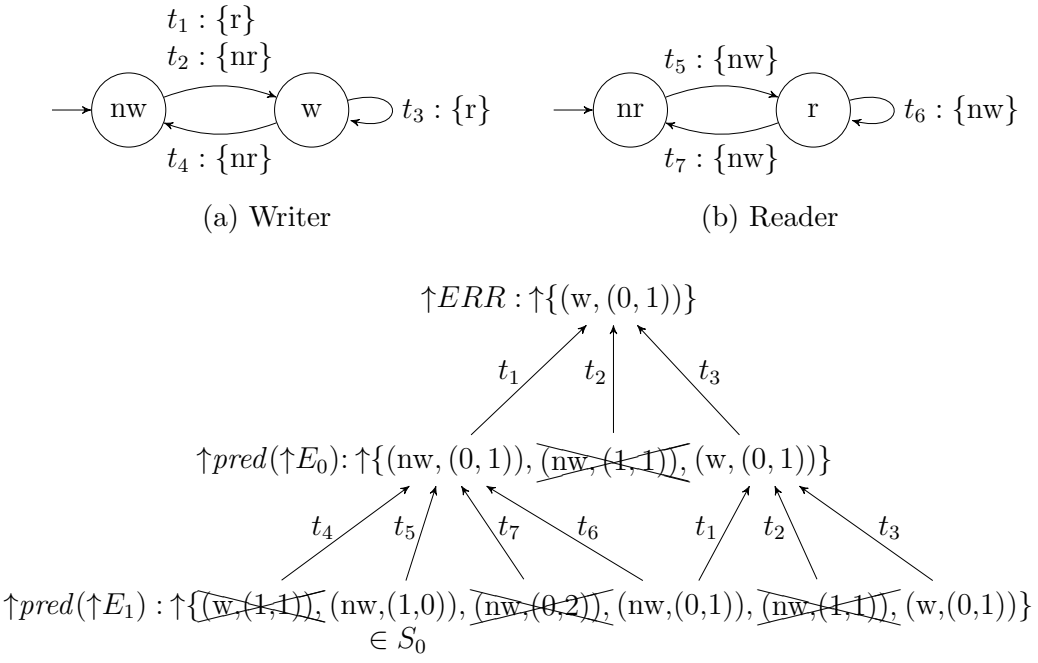
For a given set of error states R, the model checking algorithm needs to perform a backwards reachability analysis to check if it can reach an initial state. Therefore, it is essential to compute $pred^*(R)$ as the limit of the sequence $R_0 \subseteq R_1 \subseteq \ldots$ where $R_0 = R$ and $R_{i+1} = R_i \cup pred(R_i)$. If we have strong compatibility and effective pred-basis, then we can compute $pred^*(R)$ for any upwards-closed set R and reachability of arbitrary upwards-closed set is decidable. Lemma 3.3.2 states that we can effectively compute the predecessors for counter systems. This can be seen in detail in the model checking algorithm in the following Section 3.3.2.

Lemma 3.3.2. [36] *Let $M = (S, S_0, \Delta)$ be a counter system for process templates A and B. Then $(M, \lessapprox)$ has effective pred-basis.*

3.3.2 Parameterized Model Checking Algorithm

For a given counter system $M = (S, S_0, \Delta)$, based on process templates A and B, and a finite basis ERR of the set of error states, the parameterized model checking algorithm checks if there exists an $n \in \mathbb{N}_0$, such that an error state is reachable in $A||B^n$. The algorithm performs a backwards reachability analysis and returns an *error sequence*, from which we can derive concrete error paths. The following algorithm has been shown to be correct and to terminate [36].

Algorithm 1 shows how to perform parameterized model checking by iteratively computing the set of predecessors until it reaches an initial state, or a fixed point. If a fixed point is reached, then the algorithm returns *True*, i.e. the system is safe. Otherwise the procedure returns an error sequence $E_0, \ldots, E_k$, where $E_0 = ERR$, $\forall 0 < i < k : E_i = minBasis(\uparrow pred(\uparrow E_{i-1}))$, and $E_k = minBasis(\uparrow pred(\uparrow E_{k-1})) \cap S_0$. Intuitively, every E_i contains a minimal basis of the states that can reach ERR in i steps.



(a) Writer

(b) Reader

(c) Predecessor Computation

Figure 3.5: Parameterized Model Checking

Example 3.3.4. *Consider the writer-reader system in Figures* 3.5a *and* 3.5b. *We assume that the error states are all states where the writer is in* w *while at least one reader is in the reading state* r*, i.e.,* $\uparrow ERR = \{(\text{w}, (i_0, i_1)) \mid (\text{w}, (0,1)) \lessapprox (\text{w}, (i_0, i_1))\}$. *A finite basis for the error states is* $ERR = E_0 = \{(\text{w}, (0,1))\}$. *For* E_0*, the algorithm iteratively computes the set of predecessors, as described in Lemma* 3.3.2. *Figure* 3.5c *depicts the predecessor computation. States of* $\uparrow E_0$ *can be reached by taking one of the local transitions* t_1, t_2 *or* t_3*, i.e.,* $\uparrow pred(\uparrow E_0) = \uparrow\{(\text{nw}, (0,1)), (\text{nw}, (1,1)), (\text{w}, (0,1))\}$. *A minimal basis for the set of predecessors is* $E_1 = \{(\text{nw}, (0,1)), (\text{w}, (0,1))\}$*, indicated by pruning the states. Since* E_1 *contains no initial state and the algorithm has not reached a fixed point, we continue by computing the next set of predecessors for* $\uparrow E_1$*. States of* $\uparrow E_1$ *can be reached by taking one of the local transitions* t_i *for* $0 < i \leq 7$*. A minimal basis is* $E_2 = \{(\text{nw}, (1,0)), (\text{nw}, (0,1)), (\text{w}, (0,1))\}$. *Since* $E_0 \cap S_0 \neq \emptyset$*, an error state is reachable and the error sequence* $\{E_0, E_1, \{(\text{nw}, (1,0))\}\}$ *is returned.*

3.3.3 Deadlock Detection

When repairing concurrent systems, it needs to be guaranteed that a repair does not introduce a deadlock. In the following, we show how the paramaterized model checking algorithm can be adapted to detect deadlocks.

First, note that that we cannot directly use the model checking algorithm to check reachability of deadlocked state. To see this, let $s = (q_A, \mathbf{c})$ be a deadlocked state. Thus, $\mathbf{c}(i) = 0$ for every q_i that appears in a guard of an outgoing local transition from s. For a global state $s' = (q'_A, \mathbf{c}')$ with $s \lessapprox s'$ where $\mathbf{c}'(i) > 0$ for one of these q_i, some transition for s' is enabled. Thus, s' is not deadlocked and the set of deadlocked set is not upwards-closed under $\lessapprox$ from Section 3.3.1. Hence, we need a refined wqo for deadlock detection. We assume wlog. that δ_B does not contain any transition where q_i is guarded by q_i, i.e. a transition of the form $(q_i, \{q_i\}, q_j)$. This is not a restriction since any system can be transformed into one that satisfies the assumption, with a linear blowup in the number of states, and preserves reachability properties.

Definition 3.3.6. *(Refined wqo)*
Let $\lesssim_0 \subseteq \mathbb{N}_0^{|B|} \times \mathbb{N}_0^{|B|}$ *where* $\mathbf{c} \lesssim_0 \mathbf{c}'$ *iff* $\mathbf{c} \lesssim \mathbf{c}'$ *and* $\forall i \leq |B| : (\mathbf{c}(i) = 0) \Leftrightarrow (\mathbf{c}'(i) = 0)$*. Then, the* refined wqo $\lessapprox_0 \subseteq S \times S$ *for deadlock detection is defined as:*

$$(q_A, \mathbf{c}) \lessapprox_0 (q'_A, \mathbf{c}') \Leftrightarrow (q_A = q'_A \wedge \mathbf{c} \lesssim_0 \mathbf{c}').$$

Note that any set of deadlocked states is upwards-closed with respect to $\lessapprox_0$.

Example 3.3.5. *Consider the states* $s_0 = (q_A, (0, 2, 1))$*,* $s_1 = (q_A, (0, 2, 3))$ *and* $s_2 = (q_A, (1, 2, 3))$*. It holds that* $s_0 \lessapprox s_1$ *and* $s_0 \lessapprox s_2$*. Further,* $s_0 \lessapprox_0 s_1$*. However,* $s_0 \not\lessapprox_0 s_2$*, since at the first position the value for the vector for* s_2 *is not 0. Thus,* $\lessapprox_0$ *additionally checks if for every position, the value of the state vector for process* B *does not differ if it is 0.*

The following lemma states that a counter system is a WSTS for $\lessapprox_0$.

Lemma 3.3.3. [36] *Let* $M = (S, S_0, \Delta)$ *be a counter system for process templates* A *and* B*. Then,* $(M, \lessapprox_0)$ *is a WSTS.*

As shown in Section 3.3.1, for a given set $R \subseteq S$ that is upwards-closed with respect to $\lessapprox$, $pred(R)$ is upwards-closed iff $\lessapprox$ is strongly compatible with Δ. Since $\lessapprox_0$ is not strongly compatible with Δ, $pred(R)$ is not upwards closed, for any upwards-closed set R with respect to $\lessapprox_0$. Thus, we cannot use upwards-closed sets for computing $pred^*(R)$ when checking reachability of deadlocked states. Therefore, we introduce an overapproximation of $pred(R)$ that is upwards-closed with respect to $\lessapprox_0$. Furthermore, the following overapproximation is safe in the sense that every state in the overapproximation is backwards reachable in a number of steps from R.

Definition 3.3.7. *(O-Predecessor)*
Let $M = (S, S_0, \Delta)$ *be a counter system for process templates* A, B *and let* $R \subseteq S$*. Then the set of* O-predecessors *of* R *is*

$$opred(R) = pred(R) \cup \{(q_A, \boldsymbol{c}) \in S \mid \exists (q'_A, \boldsymbol{c}') \in R, t_B = (q_i, g, q_j) \in \delta_B :$$

$$(q_A, \boldsymbol{c}) \xrightarrow{t_B}{}^{+} (q'_A, \boldsymbol{c}') \wedge (\boldsymbol{c}(j) = 0 \vee \boldsymbol{c}'(i) = 0)\},$$

where $(q_A, \boldsymbol{c}) \xrightarrow{t_B}{}^{+} (q'_A, \boldsymbol{c}')$ *denotes that* $(q'_A, \boldsymbol{c}')$ *is reachable from* $(q_A, \boldsymbol{c})$ *by executing the local transition* t_B *one or more times.*

Lemma 3.3.4. [36] *Let* $R \subseteq S$ *be upwards-closed with respect to* $\lessapprox_0$. *Then* $opred(R)$ *is upwards-closed with respect to* $\lessapprox_0$.

A WSTS $(M, \lessapprox_0)$ has *effective opred-basis* if there exists an algorithm that takes as input any finite set of set $R \subseteq S$ and returns a finite basis of $\uparrow opred(\uparrow R)$. Lemma 3.3.5 shows how to compute a basis of $\uparrow opred(\uparrow R)$ from a basis R.

Lemma 3.3.5. [36] *Let* $M = (S, S_0, \Delta)$ *be a counter system for process templates* A *and* B. *Then* $(M, \lessapprox_0)$ *is a WSTS with effective opred-basis.*

Based on these results, Theorem 3.3.1 shows decidability for deadlock detection in disjunctive systems.

Theorem 3.3.1. [36] *Deadlock detection in disjunctive systems is decidable in NEXP-TIME.*

By modifying Algorithm 1, we can perform deadlock detection in a counter system M. Instead of a set of error states ERR, we pass a basis of the deadlocked states. In Line 6, the overapproximation *opred* is computed instead of *pred*, as described in Lemma 3.3.5. Furthermore, the computation of a minimal basis needs to be done with respect to the refined wqo $\lessapprox_0$ in Lines 6 and 10. By following the proof idea of Theorem 3.3.1, the algorithm terminates and runs in 2EXPTIME.

3.4 Parameterized Repair Algorithm

In this section, we present a parameterized repair algorithm that interleaves the backwards parameterized model checking algorithm from Section 3.3 with a forwards reachability analysis to generate candidate repairs.

3.4.1 Reachable Error Sequence

For a given counter system M of process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$, and a set of error states ERR, a parameterized repair algorithm generates a refinement δ' of M such that no state in ERR is reachable iff the system can be repaired, i.e., there exists a counter system M' of $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$ that reaches no state in ERR. As described in Section 3.2.2, a parameterized model checker is used to generate error sequences. By performing a forward reachability analysis, we can extract concrete *reachable* error paths. Before defining a *reachable error sequence*, we need to define the set of immediate successors.

Definition 3.4.1. *(Successor)*
Let $M = (S, S_0, \Delta)$ be a counter system and $R \subseteq S$. Then the set of immediate successors *of R is*

$$Succ(R) = \{s' \in S \mid \exists s \in R : s \to s'\}.$$

For $s \in S$, let $\Delta^{\text{local}}(s, R) = \{t_U \in \delta \mid t_U \in \Delta^{\text{local}}(s) \wedge \Delta(s, t_U) \in R\}$.

Definition 3.4.2. *(Reachable error sequence)*
Given an error sequence $E_0, \ldots, E_k$, the reachable error sequence *$\mathcal{RE} = RE_0, \ldots, RE_k$ is defined by $RE_k = E_k$ and $RE_{i-1} = Succ(RE_i) \cap \uparrow E_{i-1}$ for $1 \leq i \leq k$.*

Note that by definition, E_k only contains initial states. Intuitively, $\mathcal{RE}$ represents a set of concrete error paths of length k since each RE_i is a set of states that can reach $\uparrow ERR$ in i steps, and RE_i is reachable from S_0 in $k - i$ steps.

Example 3.4.1. *Consider Example* 3.3.4 *with the counter system for the process templates in Figure* 3.5 *and $ERR = \{(\text{w},(0,1))\}$. The parameterized model checking algorithm returned the error sequence E_0, E_1, E_2, where $E_0 = ERR$, $E_1 = \{(\text{nw}, (0, 1)), (\text{w}, (0, 1))\}$ and $E_2 = \{(\text{nw}, (1, 0))\}$. For this error sequence, we can compute the reachable error sequence $\mathcal{RE} = RE_0, RE_1, RE_2$ where $RE_2 = E_2$. Furthermore, $RE_1 = Succ(RE_2) \cap \uparrow E_1 = \{(\text{nw}, (0, 1))\}$ and $RE_0 = Succ(RE_1) \cap \uparrow E_0 = \{(\text{w}, (0, 1))\}$.*

3.4.2 Constraint Solving for Candidate Repairs

The parameterized model checking algorithm generates candidate repairs until the counter system is correct. Each candidate repair has to avoid all error paths that have been discovered so far. Therefore, every reachable error sequence is encoded into constraints such that the corresponding concrete error paths are unreachable. The SAT-based generation of candidate repairs is guided by constraints over the local transitions δ as atomic propositions of the underlying process process templates. A satisfying assignment of the constraints corresponds to the candidate repair δ', where only transition that are assigned *true* remain in δ'.

Algorithm 2 shows how to build constraints for a given reversed reachable error sequence $\mathcal{RE} = RE_k, \ldots, RE_0$ such that a candidate repair avoids error paths of $\mathcal{RE}$. The algorithm performs a forward reachability analysis for every initial state $s \in RE_k$ to build constraints for each concrete error path that starts in s. An error path is unreachable if for any step, all local transitions that lead to the successor, are removed by the candidate repair.

Example 3.4.2. *Consider the reversed reachable error sequence $\mathcal{RE} = RE_2, RE_1, RE_0$ from Example* 3.4.1. *A candidate repair δ' has to remove local transitions*

Algorithm 2 Build Constraints

```
1: //RE is a reachable error sequence
2: procedure BuildConstraints(RE)
3:     constraint ← true
4:     //build constraints for each error path starting in the initial state s
5:     for s ∈ RE[0] do
6:         //RE[1 :] is a list obtained by removing the first element from RE
7:         constraint ← constraint ∧ BuildConstraint(s, RE[1 :])
8:     return constraint
9:
10: procedure BuildConstraint(s, RE)
11:     if RE[1 :] isEmpty then
12:         //if t_U ∈ Δ^local(s) leads to RE[0], delete it
13:         return ⋀_{t_U ∈ Δ^local(s,RE[0])} ¬t_U
14:     else
15:         //if t_U leads to RE[0], delete t_U or ensure unreachability for the
16:         //remaining error sequence RE[1 :] in Δ(s, t_U)
17:         return ⋀_{t_U ∈ Δ^local(s,RE[0])} (¬t_U ∨ BuildConstraint(Δ(s, t_U), RE[1 :]))
```

from the process templates in Figures 3.5a *and* 3.5b *such that all concrete error paths of* $\mathcal{RE}$ *are unreachable. Since* RE_2 *only contains one initial state, i.e.* $s_0 = \{(\text{nw}, (1, 0))\}$*, the algorithm only checks for error paths starting in* s_0*.* s_0 *can reach a state of* RE_1 *by taking the local transition* t_5 *resulting in state* $s_1 = \{(\text{nw}, (0, 1))\}$*. For* s_1*, the error state* $\{(w, (0, 1))\}$ *is reachable by taking* t_1*. Thus, the algorithm generates the constraint* $\neg t_5 \vee \neg t_5$ *and a candidate repair* δ' *has to remove at least* t_1 *or* t_5*.*

To avoid the construction of candidate repairs that violate the totality assumption, i.e. every local state in $Q_A \cup Q_B$ has at least one local outgoing transition, every repair has to additionally satisfy the following constraint:

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

In Example 3.4.2, a candidate repair δ' is not allowed to delete t_5, because otherwise δ' would remove all local transitions of the local state nr. Furthermore, the constraint system can be extended with user-designed constraints such that a repair conforms with the designer's requirements. For example, the designer could add constraints to ensure that certain states remain reachable in the repair.

Algorithm 3 Parameterized Repair

```
1:  procedure ParameterizedRepair(M, ERR, initConstraint)
2:      M' ← M, accumConstraint ← initConstraint, isCorrect ← False
3:      // loop until a repair is found or unrealizability is detected
4:      while !isCorrect do
5:          (isCorrect, [E_0, ..., E_k]) ← ModelCheck(M', ERR)
6:          if !isCorrect then
7:              //E_k contains only initial states
8:              RE_k ← E_k, i ← k − 1
9:              while i ≠ 0 do
10:                 RE_i ← Succ(RE_{i+1}) ∩ ↑E_i
11:                 i ← i − 1
12:             //for every state in RE_k compute the corresponding constraints
13:             newConstraint ← BuildConstraints([RE_k, ..., RE_0])
14:             //append current constraints to previous iterations' constraints
15:             accumConstraint ← accumConstraint ∧ newConstraint
16:             (δ',isSat) ← SAT(accumConstraint)
17:             if !isSat then
18:                 return Unrealizable
19:             compute a new candidate using δ'
20:             M' ← Restrict(M, δ')
21:         else
22:             //repair is found
23:             return δ'
```

3.4.3 Parameterized Repair Algorithm

Algorithm 3 shows how to construct a parameterized repair for a given counter system M, a set of error states ERR and initial boolean constraints *initConstraint* on the local transition relations including the totality constraint *TRConstr*. The algorithm interleaves the backwards model checking algorithm with a forwards reachability analysis and the computation of candidate repairs. The algorithm starts by using a parameterized model checker to generate an error sequence in Line 5 following Algorithm 1. After computing the reachable error sequence $\mathcal{RE}$, the constraint system is updated with the constraints generated by Algorithm 2 in Line 13. Then, a SAT-solver is used to find a candidate repair δ' for the updated constraint system in Line 16. Then these steps are repeated for the restricted counter system with respect to δ'. The algorithm returns either a repair if the model checker detects no more error sequences or *Unrealizable* to denote that no repair exists. The algorithm always terminates as shown in [36]. A detailed example of the parameterized model checking algorithm is shown in Example 3.5.1.

Note that Algorithm 3 does not include a deadlock detection to avoid repairs that introduce deadlocks. However, it can be extended with a subprocedure for deadlock detection based on the approach in Section 3.3.3. Then, the subprocedure is called in an interleaving way with the parameterized model checker as described in Section 3.2.2.

The following Theorems 3.4.1 and 3.4.2 state that Algorithm 3 is sound and complete.

Theorem 3.4.1. [36]*(Soundness). For every repair δ' returned by Algorithm* 3*:*

- *Restrict(M, δ') is safe, i.e., $\uparrow ERR$ is unreachable, and*
- *the transition relation of Restrict(M, δ') is total.*

Theorem 3.4.2. [36]*(Completeness). If Algorithm* 3 *returns Unrealizable, then the paramaterized system has no repair.*

3.5 Extensions and Limitations

In this section, we discuss how the presented parameterized repair approach can be extended and modified to repair systems that go beyond disjunctive systems, for general safety properties. Furthermore, we show the limitations of this approach and state the open problems that are solved in the remainder of the thesis.

3.5.1 Beyond Reachability

The presented Algorithm 3 can also be used for repairing general safety properties based on the automata-theoretic model-checking approach described in Section 2.4. To this end, the safety property φ is encoded into a bad-prefix automaton $\mathcal{A}_{\neg\varphi}$ that accepts all runs of the counter system $A||B^n$ that violate φ. We build the product of the original system M with the automaton $\mathcal{A}_{\neg\varphi}$ and explicit copies of B that appear in φ. By defining a refined wqo that additionally checks if the automaton is in the same local state for two global states, we can run Algorithm 1 to check for executions that violate φ. Analogously, we can modify the repair algorithm that generates candidate repairs with respect to the refined wqo to find a repair that satisfies φ.

Example 3.5.1. *Consider the parameterized system consisting of one writer and reader-processes from Figure* 2.1*. Assume that the local transition* $(\mathrm{nr}, \{\mathrm{nw}\}, \mathrm{r})$*, is an unguarded transition instead, i.e., the transition* $(\mathrm{nr}, Q_A \cup Q_B, \mathrm{r})$*. We want to repair the system for the general safety property* $\varphi = \square((\mathrm{w} \wedge \mathrm{nr}_1) \rightarrow (\mathrm{nr}_1 \,\mathcal{W}\, \mathrm{nw}))$ *from Example* 2.3.1*. Figure* 2.4 *depicts the bad-prefix automaton $\mathcal{A}$ equivalent to $\neg\varphi$. We run the parameterized repair algorithm on the product $M \times B \times \mathcal{A}$ and the error states $\{((-,-,(*,*)), q_e)\}$, where $(-,-)$ means any writer and any*

reader state, and $*$ means 0 or 1. Thus, the system is not allowed to have an execution reaching the error state q_e of the finite automaton. The model checker may return the following error sequences:

$$E_0 = \{((-,-,(*,*)),q_e)\}$$
$$E_1 = \{((\mathrm{w},\mathrm{r}_1,(0,0)),q_1)\}$$
$$E_2 = \{((\mathrm{w},\mathrm{nr}_1,(0,0)),q_0),((\mathrm{w},\mathrm{nr}_1,(0,1)),q_0),((\mathrm{w},\mathrm{nr}_1,(1,0)),q_0)\}$$
$$E_3 = \{((\mathrm{nw},\mathrm{nr}_1,(0,0)),q_0),((\mathrm{nw},\mathrm{nr}_1,(0,1)),q_0),((\mathrm{w},\mathrm{r}_1,(0,0)),q_0),$$
$$((\mathrm{w},\mathrm{r}_1,(0,1)),q_0),((\mathrm{w},\mathrm{r}_1,(1,0)),q_0)\}$$

After updating the constraint system, the SAT-solver finds out that the error sequence can be avoided by removing the local transitions $(\mathrm{nr},\{\mathrm{nr}\},\mathrm{r}),(\mathrm{nr},\{\mathrm{r}\},\mathrm{r})$ *and* $(\mathrm{nr},\{\mathrm{w}\},\mathrm{r})$. *The next call to the model checker assures that the restricted system is safe. Note that some states were omitted from error sequences in this example for a simpler presentation.*

However, the presented approach is not able to guarantee any liveness properties, like termination or the absence of undesired loops. The main problem is that liveness checking cannot be reduced to a reachability problem. For liveness properties, we need a parameterized model checking algorithm that checks for executions with an infinite cycle violating the property. In Chapter 4, we present a parameterized model checking algorithm for liveness properties and show how the parameterized repair algorithm can be modified to include liveness repair.

3.5.2 Beyond Disjunctive Systems

The parameterized repair algorithm can be extended to other system classes that can be framed as WSTS. These systems include conjunctive systems, rendezvous systems and systems based on broadcast protocols. For synchronous transitions, we have to modify the initial constraint *TRConstr* to ensure that the repair is total for pairwise rendezvous and broadcast systems. Furthermore, a modified version of the procedure BuildConstraints has to be used when a transition relation comprises synchronous actions. When repairing synchronous systems, the main challenge is how to exclude deadlocks. While deadlock detection is decidable for rendezvous system by reduction to reachability in vector addition systems (VASS) [17, 31], Theorem 3.5.1 shows that deadlock detection for broadcast protocols is undecidable. However, for the over-approximation of lossy broadcast systems, deadlock detection is decidable [18].

Theorem 3.5.1. [36] *Deadlock detection in broadcast protocols is undecidable.*

3.5.3 Limitations

If a given parameterized system cannot be repaired, the presented approach offers no additional feedback. Instead, the designer has to add more non-determinism or

allow for more communication between processes, and run the algorithm again. This may be a non-trivial and exhausting task. Therefore, in Chapter 5, we present an operation-based repair approach, where the repair algorithm can also introduce more communication between processes. Since the designer usually wants a repair that is close to the implemented system, we introduce *minimal* repairs in Chapter 4. Intuitively, a minimal repair is a repair that only applies changes that are necessary. Since the generation of minimal repairs is done with constraints on the local transitions, the presented repair algorithm can easily be modified to minimally repair a system as well.

Chapter 4

Refinement-Based Parameterized Repair of Guarded Protocols for Liveness Properties

In this chapter, we show how to modify the parameterized repair algorithm from Jacobs, Sakr and Völp [36] to minimally repair guarded protocols for liveness properties. We introduce cutoffs to reduce the paramaterized model checking problem to model checking of finite state systems. Furthermore, we show how to extend the constraint system to generate minimal candidate repair that are globally deadlock-free. Last, we discuss the limitations of our approach. We start by giving a motivating example.

4.1 Motivating Example

Consider the parameterized system for one writer and reader processes from Figure 2.1. As shown in Example 3.5.1, the system is safe with respect to the property $\varphi_{\text{safe}} = \square((\text{w} \wedge \text{nr}_1) \rightarrow (\text{nr}_1 \,\mathcal{W}\, \text{nw}))$. However, possible system executions still include runs where the writer eventually stays in either the writing state w or in the initial state nw forever. This may not satisfy the designer's intent. By adding the additional property $\varphi_{\text{live}} = \square\diamondsuit \text{nw} \wedge \square\diamondsuit \text{w}$, the designer wants to repair the system and to guarantee that the writer does not eventually remain in the same state forever. However, the existing repair approach presented in Chapter 3 does not work for liveness properties. Since every property can be decomposed into a safety and a liveness property by Theorem 2.2.1, we need a parameterized repair algorithm for liveness properties. In the following, we show how we can modify the existing repair approach such that we can repair parameterized systems for liveness properties. A repair for φ_{live} is shown in Figure 4.1. By removing the transition $(\text{w}, \text{r}, \text{w})$, the writer cannot stay in w forever. Further, by removing the transitions $(\text{r}, \text{nw}, \text{r})$, $(\text{r}, \text{nw}, \text{nr})$, $(\text{r}, \text{nr}, \text{nr})$ and $(\text{r}, \text{r}, \text{nr})$, the writer has to eventu-

(a) Writer (b) Reader

Figure 4.1: A possible repair for φ_{live}

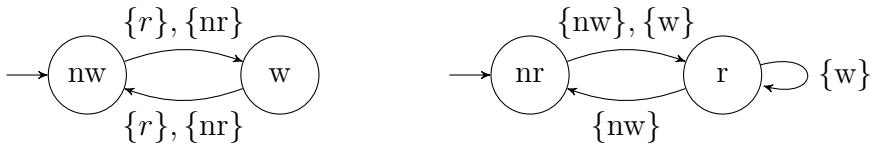
ally change its current state. Thus, the writer cannot stay in the writing state w or in the initial state nw forever at some point. Another refinement which would have also repaired the system with respect to φ_{live}, could additionally delete the transition $(\text{nw}, \text{r}, \text{w})$. Then, the process templates would still be total, the system would not be deadlocked and the system satisfies φ_{live}. However, the designer is usually interested most in repairs that allow for as much communication as possible. Therefore, we show how to *minimally* repair a system such that no transition is removed that is not responsible for an incorrect execution.

4.2 Problem Statement

In this section, we define the parameterized minimal repair problem and give a high-level overview about the modifications of the parameterized repair algorithm such that it can repair guarded protocols for liveness properties. We start by defining a minimal repair.

In the following, we only consider disjunctive systems. In Section 4.6, we explain how the repair algorithm can be modified to repair other system classes where liveness checking is decidable. Liveness properties are formulated as a parameterized specification φ. As defined in Section 2.2, φ is an LTL\X formula over atomic propositions from Q_A and indexed propositions from $Q_B \times \{1, \ldots, k\}$. The next-time operator has to be excluded such that liveness checking of disjunctive systems is decidable [13]. Furthermore, the explicit representation of parameterized systems from Section 2.1 is used rather than the counter system, since our approach relies on techniques that use the explicit representation. These techniques include model checking of finite-state systems.

Definition 4.2.1. *(Minimal Repair)*
Given the process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ *and* $B = (Q_B, \mathsf{init}_B, \delta_B)$*, and a parameterized specification* φ *defined over atomic propositions from* Q_A *and indexed propositions from* $Q_B \times \{1, \ldots, k\}$*. A set of transitions* $\delta' = \delta'_A \cup \delta'_B$ *with* $\delta'_A \subseteq \delta_A$ *and* $\delta'_B \subseteq \delta_B$ *is a* repair *for* A, B *and* φ *iff* $\forall n \geq k : A'||B'^n \vDash \varphi$*,* $A'||B'^n$ *is globally deadlock-free, and* A', B' *are total for* $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ *and* $B' = (Q_B, \mathsf{init}_B, \delta'_B)$*. A repair* δ' *is* minimal *iff there exists no repair* δ'' *with*



(a) Writer (b) Reader

Figure 4.2: Process templates used in Lemma 4.2.1

$|\delta''| > |\delta'|$.

If δ' is a minimal repair, we also say that δ' minimally repairs A and B for φ. Intuitively a repair is minimal if there is no repair that removes less transitions, i.e. only transitions are removed that are responsible for incorrect executions.

Example 4.2.1. *Consider the process templates in Figure* 2.1 *and the liveness property* $\varphi = \square\diamondsuit \mathrm{w}$. *The system violates the specification since there are executions where the writer remains in the initial state forever, i.e., for one reader process the run* $((\mathrm{nw},\mathrm{nr}),(\mathrm{nw},\mathrm{r}))^\omega$. *A repair for* φ *is* $\delta' = \delta'_A \cup \delta'_B$ *with* $\delta'_A = \delta_A \backslash \{(\mathrm{w},\mathrm{r},\mathrm{w})\}$ *and* $\delta'_B = \{(\mathrm{nr},\mathrm{nw},\mathrm{r}),(\mathrm{r},\mathrm{w},\mathrm{r})\}$. *However,* δ' *is not a minimal repair since there exists the repair* $\delta'' = \delta_A \cup \delta'_B$ *with* $|\delta''| > |\delta'|$. *In fact, the repair* δ'' *is minimal.*

While a minimal repair ensures that only transitions are removed that are responsible for an incorrect execution, the following lemma states that minimal repairs are not unique.

Lemma 4.2.1. *There exist process templates* A, B, *a parameterized specification* φ *and minimal repairs* δ_1, δ_2 *for* A, B, *and* φ *with* $\delta_1 \neq \delta_2$.

Proof. Let $A = (Q_A, \mathrm{nw}, \delta_A)$ and $B = (Q_B, \mathrm{nr}, \delta_B)$ with $Q_A = \{\mathrm{nw}, \mathrm{w}\}$, $Q_B = \{\mathrm{nr}, \mathrm{r}\}$, $\delta_A = \{(\mathrm{nw}, \{\mathrm{nr}\}, \mathrm{w}), (\mathrm{nw}, \{\mathrm{r}\}, \mathrm{w}), (\mathrm{w}, \{\mathrm{nr}\}, \mathrm{nw}), (\mathrm{w}, \{\mathrm{r}\}, \mathrm{nw})\}$ and $\delta_B = \{(\mathrm{nr}, \{\mathrm{nw}\}, \mathrm{r}), (\mathrm{nr}, \{\mathrm{w}\}, \mathrm{r}), (\mathrm{r}, \{\mathrm{w}\}, \mathrm{r}), (\mathrm{r}, \{\mathrm{nw}\}, \mathrm{nr})\}$. Figure 4.2 illustrates the process templates where A is the writer and B the reader. For $\varphi = \square\diamondsuit \mathrm{w}$, $\forall n \geq 1 : A||B^n \not\models \varphi$ since there exists a run where only process B_1 moves by taking the transitions $(\mathrm{nr}, \{\mathrm{nw}\}, \mathrm{r})$ and $(\mathrm{r}, \{\mathrm{nw}\}, \mathrm{nr})$. Thus, w never holds. $\delta_1 = \delta_A \cup \delta_B \backslash \{(\mathrm{nr}, \{\mathrm{nw}\}, \mathrm{r})\}$ and $(\mathrm{r}, \{\mathrm{nw}\}, \mathrm{nr})$ are minimal repairs for A, B and φ since for both restrictions a B-process eventuall can only move when A is in w. Since $\delta_1 \neq \delta_2$, minimal repairs are not necessarily unique. □

Before giving an overview about the modifications to the repair algorithm from Chapter 3 such that it can generate minimal repairs for parameterized systems for liveness properties, we define the parameterized minimal repair problem.

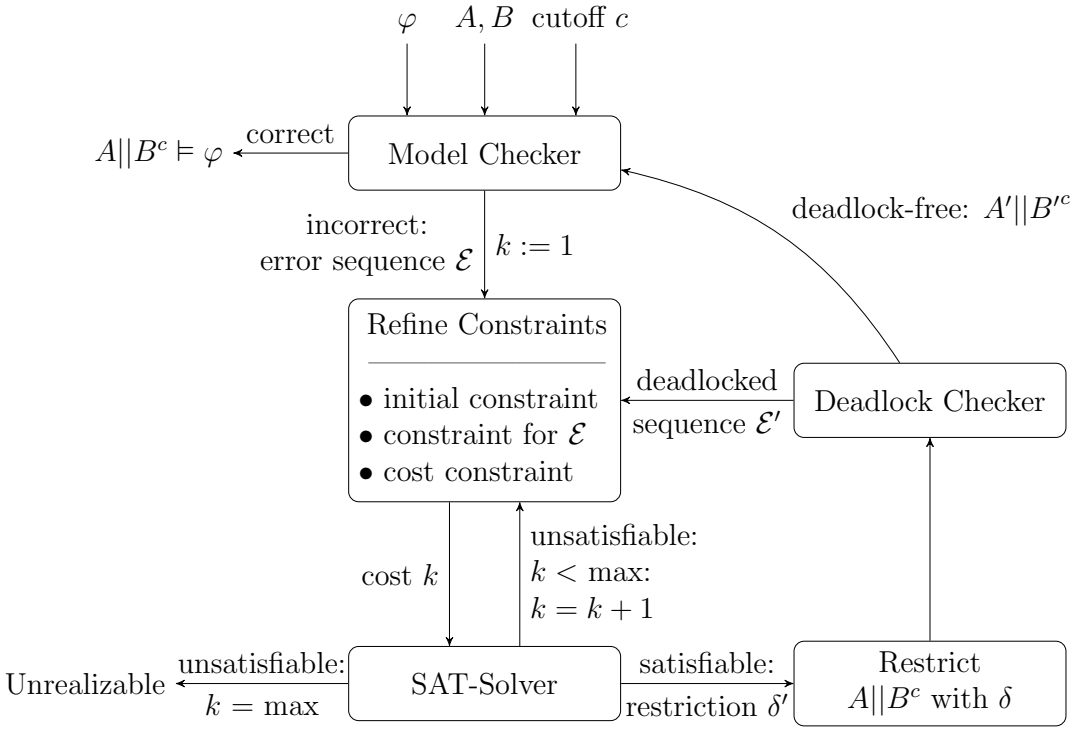


Figure 4.3: Parameterized minimal repair for liveness properties

Problem 4.2.1. *(Parameterized Minimal Repair Problem)*
Given the process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ *and* $B = (Q_B, \mathsf{init}_B, \delta_B)$*, and a parameterized specification* φ *defined over atomic propositions from* Q_A *and indexed propositions from* $Q_B \times \{1, \ldots, k\}$*. The* parameterized minimal repair problem *is to decide if there exists a minimal repair* $\delta' = \delta'_A \cup \delta'_B$ *for* A, B *and* φ*.*

Figure 4.3 shows a high-level overview about the modifications to the repair algorithm such that it can minimally repair disjunctive systems for liveness properties. The modifications preserve the general structure of the repair approach. Instead of giving a set of error states as input, the algorithm is given a parameterized specification φ expressing a liveness property. Furthermore, the inputs include the process templates A, B and a cutoff c. A cutoff c is a bound that ensures that if the system $A||B^c$ satisfies φ then $\forall n \geq c : A||B^n \vDash \varphi$. In Section 4.3.1, cutoffs are defined and it is showed how to compute a cutoff for a given specification φ and process templates A, B. Cutoffs allow to reduce the parameterized model checking problem to model checking the finite-state system $A||B^c$. By replacing the parameterized model checker with a traditional finite-state model checker, the repair algorithm from Chapter 3 can repair parameterized systems for liveness properties. By extending the constraint system, minimal repairs can be constructed. Therefore, the refined constraint system includes a cost constraint for a given cost k, that counts the number of removed

transitions. If there exists no repair that deletes at most k many transitions then the cost bound is increased and the constraints are refined. A maximal cost bound ensures that the algorithm terminates if there exists no repair. If a repair for bound k is found by the SAT-solver, the process templates are restricted and forwarded to the deadlock checker. Deadlock detection can be done using the techniques explained in Section 3.3.3. The remaining steps work analogously to the existing approach presented in Section 3.2.2.

4.3 Parameterized Model Checking for Liveness Properties

In this section, we show how to model check paramaterized systems for liveness properties. We reduce parameterized model checking to model checking finite-state systems by introducing cutoffs.

4.3.1 Cutoff

A common approach for parameterized model checking is to reduce the problem to model checking cutoff-sized systems. The definitions are taken from Jacobs and Sakr [35].

Definition 4.3.1. *(Cutoff)*
Given a class of process templates T, and a class of properties P. A cutoff *is a number $c \in \mathbb{N}$ such that for all $A, B \in T$, $\varphi \in P$ and $n \geq c$:*

$$A||B^n \vDash \varphi \Leftrightarrow A||B^c \vDash \varphi.$$

Intuitively, a cutoff is a number c that guarantees that a property φ that is satisfied or violated in the system $A||B^c$, is also satisfied or violated in any system $A||B^n$ with $n \geq c$. Note that the existence of a cutoff implies that the parameterized model checking and parameterized deadlock detection problems are decidable iff model checking and deadlock detection for their cutoff-sized systems are decidable.

Let c be a cutoff for process templates $A = (Q_A, \mathsf{init}_A, \delta_A), B = (Q_B, \mathsf{init}_B, \delta_B)$ and the paramaterized specification φ. Then, by Definitions 4.2.1 and 4.3.1 it follows that if δ is a minimal repair for process templates A, B and φ, then there exists no $n \geq c$ and $\delta' = \delta'_A \cup \delta'_B$ with $\delta'_A \subseteq \delta_A$, $\delta'_B \subseteq \delta_B$ and $|\delta'| < |\delta|$ such that $A'||B'^n \vDash \varphi$ for $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$.Intuitively, this states that if δ is a minimal repair for the cutoff-sized system $A||B^c$ then δ is a minimal repair for all systems $A||B^n$ with $n \geq c$ and vice versa. Note that this does not hold for systems $A||B^m$ with $m \leq c$ since for a small m, it may be sufficient to remove fewer transitions.

The following theorem, shows a cutoff for model checking disjunctive systems for parameterized specifications.

Theorem 4.3.1. [21] *For any disjunctive system based on process templates A and B, and any parameterized specification φ over local runs of A and k copies of B, a cutoff is $|Q_B| + k + 1$.*

Note that for disjunctive systems, there exist more and better cutoffs as shown by Außerlechner et al. [6], and Jacobs and Sakr [35].

4.3.2 Parameterized Model Checking Algorithm

By reducing the parameterized model checking problem for liveness properties to model checking of cutoff-sized systems, we can use the model checking approach presented in Section 2.4. For given process templates A, B and the parameterized specification φ, we compute a cutoff c, obtained by Theorem 4.3.1. Then, the product of the system $A||B^c$ and the nondeterministic Büchi automaton $\mathcal{A}_{\neg\varphi}$ for the negated formula is built. By performing a nested depth-first search, the algorithm checks for a reachable run that contains a cycle visiting an accepting state of $\mathcal{A}_{\neg\varphi}$. Then, this run visits an accepting state infinitely often and the specification is violated, i.e. $A||B^c \not\models \varphi$. This error run can then be returned by the model checking algorithm to obtain an error sequence. Note that this error sequence is a reachable error sequence $s_0, \ldots, s_i, s_{i+1}, \ldots, s_j$ with $s_{i+1} = s_j$ where $\forall 0 \leq k \leq j : s_k \in Q_A \times (Q_B)^c$. This sequence represents an infinite run violating φ, where $s_0, \ldots, s_i$ is a finite path that reaches the cycle $(s_{i+1}, \ldots, s_{j-1})^\omega$. If the algorithm detects no accepting cycle, the system satisfies φ, i.e., $A||B^c \models \varphi$.

4.4 Parameterized Minimal Repair

In this section, a modified version of Algorithm 3 is presented that minimally repairs disjunctive systems for liveness properties. We start by introducing the constraints to generate candidate repairs that only remove a bounded number of transitions.

4.4.1 Constraint Solving for Minimal Candidate Repairs

A minimal repair for given process templates A, B and a parameterized specification φ removes exactly those transitions that lead to violating runs. To generate a candidate repair that only removes at most k transitions for a given bound k, the constraint system needs to be extended with constraints that guarantee the deletion of at most k transitions. By increasing the cost bound, minimal candidate repairs can be constructed.

$$\exists\{cost_{c,i} | c \in \{0,\dots,k+1\}, i \in \{1,\dots,m\}\} : \phi_{\text{cost}}$$

$$\phi_{\text{cost}} = \bigwedge_{i\in\{1,\dots,m\} c\leq k} delTrans_{i,c} \wedge notDelTrans_{i,c} \wedge \neg cost_{k+1,i}$$

$$delTrans_{i,c} = \begin{cases} \neg t_i \rightarrow cost_{1,i} & \text{if } i = 1 \\ cost_{c,i-1} \wedge \neg t_i \rightarrow cost_{c+1,i} & \text{if } i > 1 \end{cases}$$

$$notDelTrans_{i,c} = \begin{cases} t_i \rightarrow cost_{0,i} & \text{if } i = 1 \\ cost_{c,i-1} \wedge t_i \rightarrow cost_{c,i} & \text{if } t > 1 \end{cases}$$

Figure 4.4: The constraint ϕ_{cost} ensures that at most k transitions are removed.

Figure 4.4 shows the constraint systems that guarantees that at most k-many transitions are removed by the candidate repair. Wlog. assume that $A = (Q_A, \mathsf{init}_A, \{t_1,\dots,t_n\})$ and $B = (Q_B, \mathsf{init}_B, \{t_{n+1},\dots,t_m\})$. By using an implicit ordering over the local transitions of A and B, the constraint system counts the number of deleted local transitions. The variable $cost_{c,i}$ is true if c-many transitions of $\{t_1,\dots,t_i\}$ are deleted so far. This bookkeeping is done by the constraints $delTrans_{i,c}$ and $notDelTrans_{i,c}$. To bound the number of removed transitions by k, ϕ_{cost} requires that $cost_{k+1,i}$ is false for all transitions.

4.4.2 Parameterized Minimal Repair Algorithm

For given process templates A, B, a parameterized specification φ and initial constraints *initConstraint*, Algorithm 4 shows how to construct a minimal repair. The algorithm is a modified version of Algorithm 3 that also works for liveness properties. Note that *initConstraint* contains totality constraints and can contain further user-designed constraints. The algorithm starts by computing a cutoff c for A, B and φ following the computation in Theorem 4.3.1. By model checking the cutoff sized-system $A||B^c$ in Line 7, an error sequence is generated iff one exists. Note that the generated error sequence is a reachable error sequence where each set is a singleton. If the model checker detects a violating run, the constraint system is updated with constraints to avoid reachability of the generated error sequence by the candidate repair (Line 10). By increasing the number *currentCost*, the algorithm checks for candidate repairs that remove at most *currentCost*-many transitions (Lines 13 - 24). In Line 16, the constraint system is updated with the corresponding cost constraints, following the construction in Figure 4.4. If the SAT-Solver does not find a candidate repair for the current bound *currentCost*, *currentCost* is increased and the algorithm checks for a candidate repair for the updated bound. Otherwise, the process templates

Algorithm 4 Parameterized Minimal Repair

```
1: procedure ParameterizedMinimalRepair(A, B, φ, initConstraint)
2:     A' ← A, B' ← B, accumConstraint ← initConstraint, isCorrect ← False
3:     // compute the cutoff
4:     cutoff ← ComputeCutoff(A, B, φ)
5:     // loop until a minimal repair is found or unrealizability is detected
6:     while !isCorrect do
7:         (isCorrect, [RE_0, ..., RE_k]) ← ModelCheck(A', B', cutoff, φ)
8:         if !isCorrect then
9:             //for every state in RE_k compute the corresponding constraints
10:            newConstraint ← BuildConstraints([RE_k, ..., RE_0])
11:            //append current constraints to previous iterations' constraints
12:            accumConstraint ← accumConstraint ∧ newConstraint
13:            currentCost ← 1, candidateFound ← False
14:            // check for a candidate repair that only removes k transitions
15:            while currentCost ≤ maxCostBound(A, B) do
16:                costConstraint ← BuildCostConstr(A, B, currentCost)
17:                (δ', isSat) ← SAT(accumConstraint ∧ costConstraint)
18:                if !isSat then
19:                    currentCost ← currentCost+1
20:                else
21:                    candidateFound ← True, (A', B') ← Restrict(A, B, δ')
22:                    break
23:            if !candidateFound then
24:                return Unrealizable
25:        else
26:            //repair is found
27:            return δ'
```

are restricted with respect to the found candidate repair and the model checker checks for correctness of the restricted system. If the SAT-solver is unable to find a repair even for the maximal bound *maxCostBound* for A and B, the system cannot be repaired and the algorithm returns *Unrealizable*. A trivial maximal bound is $|\delta_A| + |\delta_B|$, that allows a repair to remove all transitions. A more refined one is $|\delta_A| + |\delta_B| - |Q_A| - |Q_B|$ which still guarantees the restricted process templates to be total. By introducing a maximal bound for deleted transitions, termination of the algorithm follows from [36].

Note that Algorithm 4 does not include a deadlock detection to avoid repairs that introduce deadlocks. However, the algorithm can be extended with a deadlock detector to generate repairs that are deadlock-free, following the approach in Section 3.3.3.

$$\exists\{reach_s | s \in S\} : \phi_{\textit{deadlock-free}}$$

$$\phi_{\textit{deadlock-free}} = reach_{\mathsf{init}_S} \wedge \bigwedge_{s \in S} Reach_s$$

$$Reach_s = reach_s \rightarrow (\bigvee_{t \in \delta_A \cup \delta_B} ExistsSucc_{s,t}) \wedge (\bigwedge_{s' \in S, t \in \delta_A \cup \delta_B} Succ_{s,t,s'})$$

$$ExistsSucc_{s,t} = \begin{cases} t & \text{if } \exists s' \in S' : (s,t,s') \in \Delta \\ \bot & \text{else} \end{cases}$$

$$Succ_{s,t,s'} = \begin{cases} t \rightarrow reach_{s'} & \text{if } (s,t,s') \in \Delta \\ \top & \text{else} \end{cases}$$

Figure 4.5: The constraint $\phi_{\textit{deadlock-free}}$ ensures the restricted system to be deadlock-free.

Theorem 4.4.1 states that Algorithm 4 is sound. This follows from Theorem 3.4.1 and by bounding the number of removed transitions of the generated candidate repairs. Further, from Theorem 3.4.2 it follows that Algorithm 4 is complete which is stated by Theorem 4.4.2.

Theorem 4.4.1. [36]*(Soundness). For every repair δ' returned by Algorithm 4:*

- *δ' is a minimal repair for A, B and φ, and*
- *the transition relation of Restrict(A, B, δ') is total.*

Theorem 4.4.2. [36]*(Completeness). If Algorithm 4 returns Unrealizable, then the paramaterized system has no repair.*

4.5 Deadlock Detection

In this section, a SAT-based approach is presented to generate repairs that are deadlock-free. By extending the constraint system in Algorithm 4 with the following constraints, only candidate repairs that do not introduce global deadlocks, are constructed.

Assume that Algorithm 4 wants to repair the disjunctive system based on process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$ for the parameterized specification φ. For any disjunctive system there exists a cutoff $c \in \mathbb{N}$ such that if $A||B^c$ is globally deadlock-free then $\forall n \geq c : A||B^n$ is globally deadlock-free. For example, a cutoff c' for global deadlock detection in disjunctive systems is

$c' = 2|Q_B| - 1$ [6]. By defining constraints over transitions of the cutoff-sized system $A||B^c$, the SAT-solver guarantees to generate deadlock-free candidate repairs.

Figure 4.5 shows the constraint system $\phi_{\textit{deadlock-free}}$ used to generate deadlock-free repairs for a given system $A||B^c = (S, \mathsf{init}_S, \Delta)$. Intuitively, the constraints perform a reachability analysis over the global states of the cutoff-sized system. The restricted system is deadlocked if there exists a reachable state where no successor state is reachable. Thus, if a global state of the restricted system is reachable, then at least one of its successor states has to be reachable. For each global state $s \in S$, the variable reach_s indicates if s is reachable in the repaired system. The constraint requires $\mathit{reach}_{\mathsf{init}_S}$ to hold, i.e., the initial state has to be reachable. The constraint system checks for every global state s that if s is reachable, then there has to exist a reachable successor. This bookkeeping is done with the constraints $\mathit{ExistsSucc}_{s,t}$ and $\mathit{Succ}_{s,t,s'}$. $\mathit{ExistsSucc}_{s,t}$ checks if the candidate repairs contains an enabled transition that leads to a successor of s. The constraint $\mathit{Succ}_{s,t,s'}$ updates the reachable successor states, i.e., if a transition t is enabled and leads to the successor s' for s, then $\mathit{reach}_{s'}$ has to hold. By building these constraints for all global states, any candidate repair that is constructed is guaranteed to be global deadlock-free.

4.6 Extensions and Limitations

In the following, we discuss how the presented repair algorithm can be modified to minimally repair system classes that go beyond disjunctive systems. Furthermore, the limitations of the repair approach are shown.

Algorithm 4 can be used to repair systems for any property that can be expressed as an LTL\X-formula. These properties include safety and liveness properties. Further, the algorithm can be extended to other system classes where there exists a cutoff for model checking, including conjunctive systems by interpreting the guards conjunctively [35]. The algorithm can also be modified for pairwise-rendezvous system if there exists a cutoff for a given system and specification. However, this is not necessarily the case as shown by Aminof et al. [2]. If there exists a cutoff, for synchronous transitions the initial constraint has to be modified to ensure that the repair is total, as described in Section 3.5.2. Furthermore, the procedure BuildConstraints has to be modified to avoid found error sequences. The modified procedures for conjunctive and pairwise-rendezvous systems are presented by Jacobs and Sakr [36]. However, the repair algorithm cannot be extended to repair broadcast systems for liveness properties since the parameterized model checking problem is undecidable for broadcast systems and liveness properties [13]. The deadlock detection approach, introduced in Section 4.5, works for all system classes where there exists a cutoff for

global deadlock detection. For synchronous transitions the constraints have to be modified in a similar way as for the procedure BuildConstraints. This also holds for the cost constraints ϕ_{cost} to construct minimal candidate repairs for synchronous transitions.

As described in Section 3.5.3, if a given system cannot be repaired for a given specification, the designer can add transitions for more communication between the processes, and run the algorithm again. If the designer is interested in a repair that removes as many of these added transitions as possible, the constraint system could be extended with a cost constraint $\phi_{cost'}$. This constraint works similar to the constraint for minimal repairs, but checks for repairs that delete as many transitions as possible, for the added transitions. A repair approach that can automatically add transitions for more communication, is shown in Chapter 5.

Chapter 5

Operation-Based Parameterized Repair of Guarded Protocols

In this chapter, we introduce a paramaterized repair approach that applies a set of operations to repair guarded protocols. This operation-based parameterized repair approach is inspired by the explainable reactive synthesis approach introduced by Baumeister et al. [10]. We present a paramaterized minimal repair algorithm where correctness of the repaired system is witnessed by an annotation function. Furthermore, we discuss possible extensions and limitations of this approach. We start by giving a motivating example.

5.1 Motivating Example

Consider the parameterized system for one writer and an arbitrary number of reader processes, depicted in Figure 5.1. Possible system executions include runs where the writer does not move and stays in the initial state forever or eventually remains in the writing state. Thus, the parameterized system violates the specification $\varphi_{\text{live}} = \square\diamondsuit \text{nw} \wedge \square\diamondsuit \text{w}$ that requires the writer to change its state infinitely often. However, the presented repair approach in Chapter 4 is unable to repair the system since by only removing transitions, the writer is unable to access the initial state once it starts writing. Furthermore, the guard for the transition of the reader process from r to nr needs to be changed to avoid runs where the writer stays in the initial state forever. This is also not possible for the refinement-based repair approach. Instead, the designer has to change the system by adding transitions for more communication between the processes, and run the repair algorithm again. For concurrent systems, when allowing for more communication, the designer needs to be very careful to not introduce system executions that violate the designer's intent. Since this may be a non-trivial task, there is a need for a parameterized minimal repair approach that can automatically add transitions for more communication if needed. The following approach shows how

(a) Writer (b) Reader

Figure 5.1: A parameterized system violating φ_{live}

to repair systems by applying a set of operations, including transition redirects and changing transition guards. The system in Figure 5.1 can be repaired for φ_{live} by redirecting the transition loop in the writing state to the initial state. Furthermore, by changing the transition guard for the reader process from r to nr, the system in Figure 4.1 is obtained which has been shown to satisfy φ_{live}.

5.2 Problem Statement

In this section, we lay the foundation of the operation-based minimal repair approach. We start by defining the possible operations and consistent transformations that include a set of operations. After formulating the minimal repair problem, we show a high-level parameterized repair algorithm that applies the presented operations.

5.2.1 Operations

In the following, we specify systems with parameterized specifications. The given process templates A and B are interpreted disjunctively and we use the explicit representation of parameterized systems from Section 2.1. Furthermore, we do not assume that every transition guard is a singleton. However, we assume that every process template $P \in \{A, B\}$ is complete, i.e., there exists a transition from every state $s \in Q_P$ to each state $s' \in Q_P$, and that P is deterministic, i.e., there exists no $q_P, q_{P_1}, q_{P_2} \in Q_P$ and $g_1, g_2 \in \mathcal{P}(A \dot\cup B)$ where $q_{P_1} \neq q_{P_2}$, $(q_P, g_1, q_{P_1}) \in \delta_P$, $(q_P, g_2, q_{P_2}) \in \delta_P$ and $g_1 \cap g_2 \neq \emptyset$. Note that this is not a restriction since every state q_P where q'_P is not an immediate successor can be represented by the transition $(q_P, \emptyset, q'_P)$. In the following, we define two possible operations that change a transition guard or redirect a transition.

For the process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$, an *operation* o is either a *changed transition guard* or a *redirection of a transition* of $P \in \{A, B\}$. The process templates A' and B' that result from applying an operation o to A, B, is denoted by $(A', B') = \mathit{apply}((A, B), o)$.

A changed transition guard for $P \in \{A, B\}$ is denoted by the tuple $o_{\text{guard}} = (q_P, q'_P, g)$, where $q_P, q'_P \in Q_P$ and $g \in \mathcal{P}(A \dot\cup B)$. A changed transition guard

operation changes the guard of the transition from q_P to q'_P to guard g. For a changed transition guard $o_{\text{guard}} = (q_P, q'_P, g)$, the resulting process templates $(A', B') = \mathit{apply}((A, B), o_{\text{guard}})$ are defined as:

- If $P = A$, then $B' = B$ and $A' = (Q_A, \mathsf{init}_A, \delta'_A)$, where $(q_P, g', q'_P) \in \delta'_A$. For all $q_A, q'_A \in Q_A, g' \in \mathcal{P}(A \dot{\cup} B)$ with $q_A \neq q_P$ or $q'_A \neq q'_P$: $(q_A, g', q'_A) \in \delta'_A$ iff $q'_A \neq q'_P$: $(q_A, g', q'_A) \in \delta_A$.
- If $P = B$, then $A' = A$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$, where δ'_B is defined analogously.

Note that changed transition guard operations are sufficient to repair any system as shown in Lemma 5.2.1. However, in practice, it is often efficient to repair systems by redirecting a transition instead. Furthermore, transition redirections are necessary to repair labeled process templates as discussed in Section 5.5.

A transition redirection for $P \in \{A, B\}$ is denoted by the tuple $o_{\text{transition}} = (q_P, q'_P, g)$, where $q_P, q'_P \in Q_P$, $g \in \mathcal{P}(A \dot{\cup} B)$ and $\forall q \in g : \exists q''_P \in Q_P, g' \in \mathcal{P}(A \dot{\cup} B)$ with $(q_P, g', q''_P) \in \delta_P$ and $q \in g'$. For a transition redirection $o_{\text{transition}} = (q_P, q'_P, g)$, the resulting process templates $(A', B') = \mathit{apply}((A, B), o_{\text{transition}})$ are defined as:

- If $P = A$, then $B' = B$ and $A' = (Q_A, \mathsf{init}_A, \delta'_A)$. For all $g' \in \mathcal{P}(A \dot{\cup} B)$, $(q_P, g' \cup g, q'_P) \in \delta'_A$ iff $(q_P, g', q'_P) \in \delta_A$. For all $q_A \in Q_A$ with $q_A \neq q'_P$ and for all $g' \in \mathcal{P}(A \dot{\cup} B)$ it holds that $(q_P, g' \backslash g, q_A) \in \delta'_A$ iff $(q_P, g', q_A) \in \delta_A$. Furthermore, for all $q_A, q'_A \in Q_A$ where $q_A \neq q_P$ and for all $g' \in \mathcal{P}(A \dot{\cup} B)$ it holds that $(q_A, g', q'_A) \in \delta'_A$ iff $(q_A, g', q'_A) \in \delta_A$.
- If $P = B$, then $A' = A$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$, where δ'_B is defined analogously.

Intuitively, a transition redirection $o_{\text{transition}} = (q_P, q'_P, g)$ redirects all transitions from q_P to q'_P for guard g. Thus, when applying $o_{\text{transition}}$, the guard g needs to be added for the transition from q_P to q'_P. Furthermore, g needs to be removed from the guard for all other transitions starting in q_P.

A finite set of operations ξ is called a *transformation*. A transformation for process templates A and B is called *consistent* if there exists no $o_1, o_2 \in \xi$ such that $\mathit{apply}(\mathit{apply}((A, B), o_1), o_2) \neq \mathit{apply}(\mathit{apply}((A, B), o_2), o_1)$, i.e., the resulting templates do not differ depending on the order in which operations are applied. An example for a transformation that is not consistent, is any transformation containing two different changed transition guards operations for the same transition. Then, the resulting templates differ if the changed transition guard operations are applied in a different order. For a consistent transformation ξ for A, B, the process templates A', B' that are reached when applying every transformation in ξ, is denoted by $(A', B') = \mathit{apply}^*((A, B), \xi)$.

Note that for any transition redirection $o_{\text{transition}} = (q_P, q'_P, g)$, there exists a transformation ξ such that $\mathit{apply}((A, B), o_{\text{transition}}) = \mathit{apply}^*((A, B), \xi)$, where ξ only contains changed transition guard operations. A transition redirection $o_{\text{transition}} = (q_P, q'_P, g)$ redirects all transitions for g from q_P to q'_P for $P \in \{A, B\}$. Thus, for the transformation $\xi = \{o_{\text{guard}}(q_P, q''_P, g' \backslash g) \mid \exists q''_P \in Q_P, g' \in \mathcal{P}(A \dot\cup B) : q''_P \neq q'_P \wedge (q_P, g', q''_P) \in \delta_P\} \cup \{o_{\text{guard}}(q_P, q'_P, g \cup g') \mid \exists g' \in \mathcal{P}(A \dot\cup B)\} : (q_P, g'', q'_P) \in \delta_P\}$, it holds that $\mathit{apply}((A, B), o_{\text{transition}}) = \mathit{apply}^*((A, B), \xi)$. Intuitively, the transition redirection can be represented by a set of changed guard transitions such that each guard is removed by all transitions from q_P to any state $q''_P \neq q'_P$ and is added to the guard for the transition from q_P to q'_P. However, by including transition redirections, we obtain more efficient minimal repairs that provide a more visual feedback for the designer.

Example 5.2.1. *Consider the writer A and reader B in Figure* 5.1*. By applying the consistent transformation $\xi = \{o_{\text{guard}}, o_{\text{transition}}\}$ with $o_{\text{guard}} = (\text{nr}, \text{r}, \{\text{w}\})$ and $o_{\text{transition}} = (\text{w}, \text{nw}, Q_A \cup Q_B)$, the writer A' and reader B' in Figure* 4.1 *is reached, i.e., $(A', B') = \mathit{apply}^*((A, B), \xi)$.*

5.2.2 Minimal Repair Transformations

Based on consistent transformations, we can define minimal repair transformations.

Definition 5.2.1. *(Minimal Repair Transformation)*
Given the process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$, and the parameterized specification φ defined over atomic propositions from Q_A and indexed propositions from $Q_B \times \{1, \ldots, k\}$. A consistent transformation ξ is a repair transformation *for A, B and φ iff $\forall n \geq k : A'||B'^n \vDash \varphi$ and $A'||B'^n$ is globally deadlock-free where $(A', B') = \mathit{apply}^*((A, B), \xi)$. A repair ξ is* minimal *if there exists no repair ξ' with $|\xi'| < |\xi|$.*

In the remainder of this chapter, we simply call a repair transformation, a repair. We only make the distinction where necessary. For example, the transformation in Example 5.2.1 is a minimal repair for $\varphi_{\text{live}} = \square\diamondsuit \text{nw} \wedge \square\diamondsuit \text{w}$ for the process templates in Figure 5.1.

Lemma 5.2.1 states that for the given process templates A, B and the parameterized specification φ defined over states of A and k processes of B, there exists a repair iff there exists A', B' with $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (B_A, \mathsf{init}_B, \delta'_B)$ such that $n \geq k : A'||B'^n \vDash \varphi$. Thus, it shows that the defined operations are sufficient to repair any system iff φ is realizable for templates with states Q_A and Q_B. This is important since this property does not hold for the refinement-based approach in Chapter 4.

Lemma 5.2.1. *Given process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$, $B = (Q_B, \mathsf{init}_B, \delta_B)$ and the parameterized specification φ defined over atomic propositions from Q_A and indexed propositions from $Q_B \times \{1, \ldots, k\}$. There exists a repair ξ for A, B and φ iff there exists $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$ such that $\forall n \geq k : A'||B'^n \vDash \varphi$.*

Proof. Let $A = (Q_A, \mathsf{init}_A, \delta_A)$, $B = (Q_B, \mathsf{init}_B, \delta_B)$ and the parameterized specification φ is defined over atomic propositions from Q_A and indexed propositions from $Q_B \times \{1, \ldots, k\}$. The lemma holds iff both implications hold.

Let ξ be a repair for A, B and φ. Then, for $(A', B') = \mathit{apply}^*((A, B), \xi)$ it follows from Definition 5.2.1 that $\forall n \geq k : A'||B'^n \vDash \varphi$.

Let $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$ such that $\forall n \geq k : A'||B'^n \vDash \varphi$. By assumption δ'_A and δ'_B are complete. Then, ξ is a consistent transformation defined as $\xi = \{o_{\text{guard}}(q_A, q'_A, g) \mid (q_A, g, q'_A) \in \delta'_A\} \cup \{o_{\text{guard}}(q_B, q'_B, g) \mid (q_B, g, q'_B) \in \delta'_B\}$, i.e., ξ changes the guard of every transition to the guard for the transition in δ'_A and δ'_B. Since $(A', B') = \mathit{apply}^*((A, B), \xi)$, ξ is a repair for A, B and φ. □

While a minimal repair ensures that only operations are applied that minimally remove incorrect system executions, Lemma 5.2.2 states that minimal repairs are not unique. Note that the proof of Lemma 5.2.2 works analogously to the proof of Lemma 4.2.1.

Lemma 5.2.2. *There exist process templates A, B, a parameterized specification φ and minimal repairs ξ_1, ξ_2 for A, B, and φ with $\xi_1 \neq \xi_2$.*

Proof. Let $A = (Q_A, \mathrm{nw}, \delta_A)$ and $B = (Q_B, \mathrm{nr}, \delta_B)$ with $Q_A = \{\mathrm{nw}, \mathrm{w}\}$, $Q_B = \{\mathrm{nr}, \mathrm{r}\}$, $\delta_A = \{(\mathrm{nw}, \{\mathrm{nr}\}, \mathrm{w}), (\mathrm{nw}, \{\mathrm{r}\}, \mathrm{w}), (\mathrm{w}, \{\mathrm{nr}\}, \mathrm{nw}), (\mathrm{w}, \{\mathrm{r}\}, \mathrm{nw})\}$ and $\delta_B = \{(\mathrm{nr}, \{\mathrm{nw}\}, \mathrm{r}), (\mathrm{nr}, \{\mathrm{w}\}, \mathrm{r}), (\mathrm{r}, \{\mathrm{w}\}, \mathrm{r}), (\mathrm{r}, \{\mathrm{nw}\}, \mathrm{nr})\}$. Figure 4.2 illustrates the process templates where A is the writer and B the reader. For $\varphi = \square\diamondsuit \mathrm{w}$, $\forall n \geq 1 : A||B^n \nvDash \varphi$ since there exists a run where only process B_1 moves by taking the transitions $(\mathrm{nr}, \{\mathrm{nw}\}, \mathrm{r})$ and $(\mathrm{r}, \{\mathrm{nw}\}, \mathrm{nr})$. Thus, w never holds. $\xi_1 = \{o_{\text{guard}}(\mathrm{nr}, \mathrm{r}, \{\mathrm{w}\})\}$ and $\xi_2 = \{o_{\text{guard}}(\mathrm{r}, \mathrm{nr}, \emptyset)\}$ are minimal repairs for A, B and φ since for both restrictions a B-process eventuall can only move when A is in w. Since $\xi_1 \neq \xi_2$, minimal repairs are not necessarily unique. □

Since the following parameterized repair algorithm generates a transformation for a cutoff-sized system, the following observation should be noted. Let c be a cutoff for process templates $A = (Q_A, \mathsf{init}_A, \delta_A), B = (Q_B, \mathsf{init}_B, \delta_B)$ and the paramaterized specification φ. Then, by Definitions 5.2.1 and 4.3.1 it follows that if ξ is a minimal repair for process templates A, B and φ, then there exists no $n \geq c$ and ξ' with $|\xi'| < |\xi|$ such that for $(A', B') = \mathit{apply}^*((A, B), \xi')$ it holds $A'||B'^n \vDash \varphi$. Intuitively, this states that if ξ is a minimal repair for the cutoff-sized system $A||B^c$ then ξ is a minimal repair for all systems $A||B^n$ with $n \geq c$ and vice versa. Note that this does not hold for systems $A||B^m$ with $m \leq c$ since for a small m, it may be sufficient to apply fewer operations.

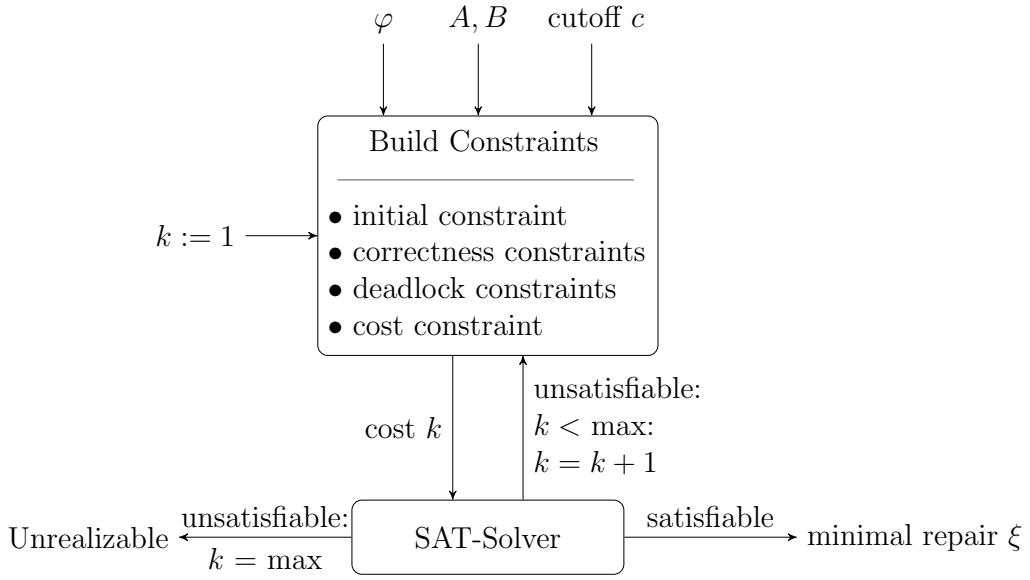


Figure 5.2: Parameterized minimal repair based on consistent transformations

5.2.3 Parameterized Minimal Repair Problem

Before giving an overview about the repair algorithm generating consistent transformations that minimally repair disjunctive systems, we define the parameterized minimal repair problem.

Problem 5.2.1. *(Parameterized Minimal Repair Problem)*
Given the process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ *and* $B = (Q_B, \mathsf{init}_B, \delta_B)$*, and a parameterized specification* φ *defined over atomic propositions from* Q_A *and indexed propositions from* $Q_B \times \{1, \ldots, k\}$*. The* parameterized minimal repair problem *is to decide if there exists a minimal repair* ξ *for* A, B *and* φ*.*

Figure 5.2 shows a high-level overview of the parameterized minimal repair algorithm that constructs consistent transformations. For a given parameterized specification φ, process templates A, B and a cutoff c, the algorithm returns a minimal repair ξ iff one exists. In contrast to the repair approach in Chapter 4, the transformation generated by the SAT-solver is guaranteed to satisfy the specification. Thus, the algorithm does not generate candidate repairs that need to be model checked. Instead, correctness of the generated transformation is guaranteed by adding constraints defined over the resulting cutoff-sized system as explained in the following Section 5.3. Furthermore, the constraint system contains deadlock constraints to avoid transformation that introduce global deadlocks, following the approach in Section 4.5. To obtain a minimal repair, the algorithm bounds the number of operations by encoding a cost constraints similar to the constraints shown in Section 4.4.1. If the SAT-solver cannot generate a transformation for the maximal bound of operations, the system cannot

be repaired for the given specification.

5.3 Verification of Finite-State Systems

In this section, we show how to verify correctness of finite-state systems using annotation functions. By encoding a valid annotation function, the parameterized repair algorithm generates transformations such that the resulting system satisfies the specification.

For a given specification φ, bounded synthesis automatically produces size-optimal transition systems that satisfy φ [26]. Let $\mathcal{A}_\varphi$ be a universal co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Note that $\mathcal{A}_\varphi$ can be obtained by complementing the nondeterministic Büchi automaton $\mathcal{A}_{\neg\varphi}$ for the negated formula that is obtained by the construction in Theorem 2.3.1. A transition system $\mathcal{T}$ satisfies φ iff every trace of $\mathcal{T}$ satisfies φ. Correctness of $\mathcal{T}$ can be verified by checking if every run of $\mathcal{T}$ on the universal co-Büchi automaton $\mathcal{A}_\varphi$ visits a rejecting state finitely often. This acceptance of $\mathcal{T}$ by $\mathcal{A}_\varphi$ is witnessed by an annotation function. The bounded synthesis approach produces systems satisfying φ by solving a constraint system that asserts the existence of a valid annotation function. Hence, by extending the constraint system of the parameterized repair approach with these constraints, it constructs transformations such that the resulting system satisfies φ. The following definitions are taken from Faymonville et al. [26]. Note that the definitions are adjusted to the setting of guarded protocols.

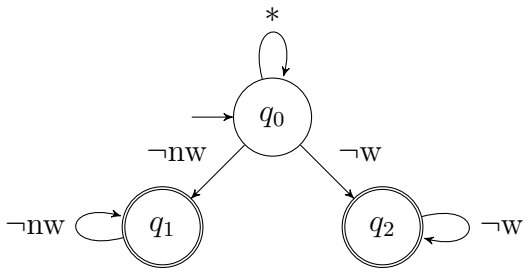
Definition 5.3.1. *(Annotation Function)*
Given the system $A||B^n = (S, \mathsf{init}_S, \Delta)$ for process templates A, B, and a universal co-Büchi automaton $\mathcal{A} = (Q, q_0, \delta, F)$. The annotation function $\lambda : S \times Q \to \{\bot\} \cup \mathbb{N}$ *is a function that maps each state $s \in S \times Q$ to $\bot$ or a natural number.*

Using an annotation function λ, it can be checked if the rejecting states are visited only finitely often by simulating the product of the system and the universal co-Büchi automaton. If this holds for λ, we call λ *valid*.

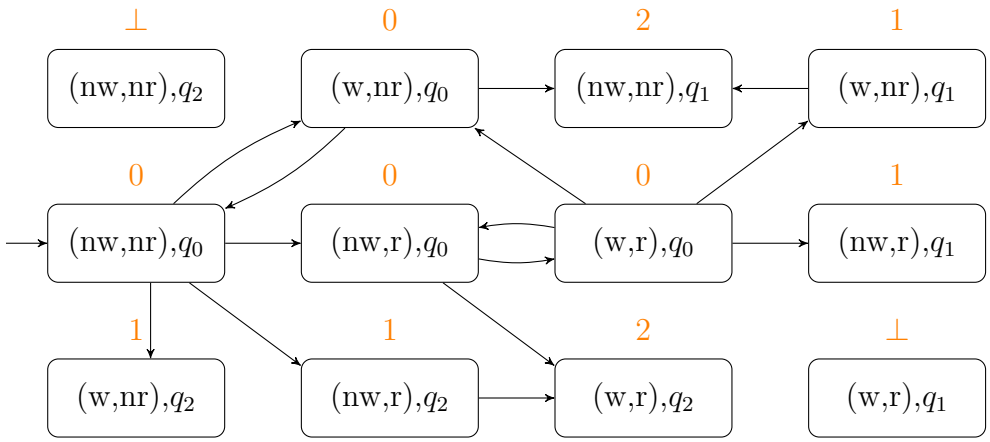
Definition 5.3.2. *(Valid Annotation Function)*
Given the system $A||B^n = (S, \mathsf{init}_S, \Delta)$ for process templates A, B, and a universal co-Büchi automaton $\mathcal{A} = (Q, q_0, \delta, F)$. The annotation function $\lambda : S \times Q \to \{\bot\} \cup \mathbb{N}$ is valid *if it satisfies the following conditions:*

- $\lambda(\mathsf{init}_S, q_0) \neq \bot$
- $\forall s \in S, q \in Q : \lambda(s, q) = k \neq \bot \to \forall s' \in S, q' \in Q : (s, s') \in \Delta \wedge (q, s, q') \in \delta \to \lambda(s', q') \rhd_{q'} k$, *where $\rhd_{q'} := >$ if $q' \in F$ and $\geq$ otherwise*

Thus, a valid annotation function requires the initial state of the product not to be mapped to $\bot$ to indicate that this state is reachable. Furthermore,



(a) A universal co-Büchi automaton $\mathcal{A}$ for φ_{live}



(b) A valid annotation function for $\mathcal{A}$ and $A||B^1$

Figure 5.3: Verification of $A||B^1$ for φ_{live} , where A is the writer and B the reader process from Figure 4.1

a reachable state in the product is labeled with a greater number than all of its predecessors when reaching a rejecting state. Otherwise, it is labeled with a number greater or equal. This guarantees that if a valid annotation function exists, then no run of the system visits a rejecting state infinitely only, as shown in Theorem 5.3.1.

Theorem 5.3.1. [26] *Given a system $A||B^n = (S, \mathsf{init}_S, \Delta)$ for process templates A, B, and a universal co-Büchi automaton $\mathcal{A} = (Q, q_0, \delta, F)$. $A||B^n$ holds on $\mathcal{A}$ iff a valid $(|S| \cdot |\mathcal{A}|)$-bounded annotation function exists.*

Example 5.3.1. *Consider the system $A||B^1$ for the writer process A and one reader process B from Figure* 4.1*, and the specification $\varphi_{\text{live}} = \square\diamondsuit \text{nw} \wedge \square\diamondsuit \text{w}$. Figure* 5.3a *shows a universal co-Büchi automaton $\mathcal{A}$ for φ_{live}. By Theorem* 5.3.1*, $A||B^1$ satisfies φ_{live} if there exists a valid annotation function for $A||B^1$ and $\mathcal{A}$. Figure* 5.3b *illustrates the product of $A||B^1$ and $\mathcal{A}$ where the acceptance is witnessed by a valid annotation function that is represented by the orange labels. Since the states* ((nw,nr),q_2) *and* ((w,r),q_1) *are unreachable, they are labeled with*

$\perp$. *The states* ((nw,nr),q_0),((w,nr),q_0), ((nw,r),q_0) *and* ((w,r),q_0) *are labeled with* 0 *because all of their respective predecessors contain no rejecting states and have visited no rejecting state, yet. The states* ((w,nr),q_2), ((nw,r),q_2), ((nw,r),q_1) *and* ((w,nr),q_1) *are labeled with* 1 *since they visit a rejecting state and have to be labeled strictly greater than all of their respective predecessors which are labeled with* 0. *Furthermore, the annotation function labels the states* ((nw,nr),q_1) *and* ((w,r),q_2) *with* 2 *such that they are labeled strictly greater than all of their predecessors. Thus, the annotation function is valid and* $A||B^1$ *holds on* $\mathcal{A}$.

5.4 Parameterized Minimal Repair

In this section, the constraints for generating consistent transformations are presented. Furthermore, a parameterized minimal repair algorithm is shown that applies a minimal number of operations.

Given a parameterized specification φ, process templates $A = (Q_A, \mathsf{init}_A, \delta_A)$ and $B = (Q_B, \mathsf{init}_B, \delta_B)$, and a cutoff c for φ, A and B. By Lemma 5.2.1, there exists a repair ξ for φ and $A||B^c$ iff there exist process templates $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$ with $A'||B'^c \vDash \varphi$. Thus, the SAT-solver needs to find δ'_A and δ'_B such that the resulting system satisfies φ, is globally deadlock-free and can be obtained from A, B with a minimal consistent transformation. In the following, we present the corresponding constraints.

5.4.1 Constraint Solving for Valid Annotation Functions

Verifying if the generated system satisfies φ, can be done by checking the existence of a valid annotation function, as described in Section 5.3. Figure 5.4 shows the modified basic encoding from bounded synthesis to encode a valid annotation function for a system $A'||B'^c$ and the universal co-Büchi automaton $\mathcal{A} = (Q_\mathcal{A}, q_{\mathcal{A},0}, \delta_\mathcal{A}, F)$ for φ [26]. The generated process templates $A' = (Q_A, \mathsf{init}_A, \delta'_A)$ and $B' = (Q_B, \mathsf{init}_B, \delta'_B)$ are represented by the variables $\delta'_{A_{q,g,q'}}$ and $\delta'_{B_{q,g,q'}}$. Assume for better readability that all transition guards only contain one state, in contrast to the assumption in Section 5.2. This is not a contradiction or restriction since both system classes are equally expressive and can be transformed into each other. The annotation function is represented by the variables $\lambda^{\mathbb{B}}_{s,q_\mathcal{A}}$ and $\lambda^{\#}_{s,q_\mathcal{A}}$, where $\lambda^{\mathbb{B}}_{s,q_\mathcal{A}}$ denotes if the state $(s, q_\mathcal{A})$ of the product is reachable and $\lambda^{\#}_{s,q_\mathcal{A}}$ denotes the annotated number. Note that the annotation function is defined over states of the automaton and global states $s \in S$ of the cutoff-sized system, i.e., $S = Q_A \times (Q_B)^c$. The constraint $\phi_{correct}$ ensures that the annotation function is valid. Thus, the initial state of the product has to be reachable, i.e., $\lambda^{\mathbb{B}}_{\mathsf{init}_S,q_{\mathcal{A},0}} = \top$. Furthermore, $\phi_{correct}$ includes the constraint *totality* to ensure that both resulting process templates are total. The constraints $lambdaA_{s,q_\mathcal{A}}$ and

$$\exists\{\delta'_{A_{q,g,q'}} | q, q' \in Q_A, g \in Q_B\}$$
$$\exists\{\delta'_{B_{q,g,q'}} | q, q' \in Q_B, g \in Q_A \dot{\cup} Q_B\}$$
$$\exists\{\lambda^{\mathbb{B}}_{s,q_\mathcal{A}}, \lambda^{\#}_{s,q_\mathcal{A}} | s \in S, q_\mathcal{A} \in Q_\mathcal{A}\} : \phi_{correct}$$

$$\phi_{correct} = \lambda^{\mathbb{B}}_{\mathsf{init}_S, q_{\mathcal{A},0}} \wedge totality \wedge \bigwedge_{s \in S} \bigwedge_{q_\mathcal{A} \in Q_\mathcal{A}} lambdaA_{s,q_\mathcal{A}} \wedge lambdaB_{s,q_\mathcal{A}}$$

$$totality = \bigwedge_{q \in Q_A} \bigvee_{g \in Q_B} \bigvee_{q' \in Q_A} \delta'_{A_{q,g,q'}} \wedge \bigwedge_{q \in Q_B} \bigvee_{g \in Q_A \dot{\cup} Q_B} \bigvee_{q' \in Q_B} \delta'_{B_{q,g,q'}}$$

$$lambdaA_{s,q_\mathcal{A}} = \lambda^{\mathbb{B}}_{s,q_\mathcal{A}} \rightarrow \bigwedge_{q'_\mathcal{A} \in Q_\mathcal{A}} \bigwedge_{q \in Q_A} \bigwedge_{g \in Q_B} \bigwedge_{q' \in Q_A} \bigwedge_{s' \in S} (\delta'_{A_{q,g,q'}} \rightarrow$$
$$\begin{cases} (\lambda^{\mathbb{B}}_{s',q'_\mathcal{A}} \wedge \lambda^{\#}_{s',q'_\mathcal{A}} \geq \lambda^{\#}_{s,q_\mathcal{A}})) & \text{if } (q_\mathcal{A}, s, q'_\mathcal{A}) \in \delta_\mathcal{A} \wedge s \overset{(q,g,q')}{\rightarrow} s' \wedge q'_\mathcal{A} \notin F \\ (\lambda^{\mathbb{B}}_{s',q'_\mathcal{A}} \wedge \lambda^{\#}_{s',q'_\mathcal{A}} > \lambda^{\#}_{s,q_\mathcal{A}})) & \text{if } (q_\mathcal{A}, s, q'_\mathcal{A}) \in \delta_\mathcal{A} \wedge s \overset{(q,g,q')}{\rightarrow} s' \wedge q'_\mathcal{A} \in F \\ \top) & \text{else} \end{cases}$$

$$lambdaB_{s,q_\mathcal{A}} = \lambda^{\mathbb{B}}_{s,q_\mathcal{A}} \rightarrow \bigwedge_{q'_\mathcal{A} \in Q_\mathcal{B}} \bigwedge_{q \in Q_B} \bigwedge_{g \in Q_A \dot{\cup} Q_B} \bigwedge_{q' \in Q_B} \bigwedge_{s' \in S} (\delta'_{B_{q,g,q'}} \rightarrow$$
$$\begin{cases} (\lambda^{\mathbb{B}}_{s',q'_\mathcal{A}} \wedge \lambda^{\#}_{s',q'_\mathcal{A}} \geq \lambda^{\#}_{s,q_\mathcal{A}})) & \text{if } (q_\mathcal{A}, s, q'_\mathcal{A}) \in \delta_\mathcal{A} \wedge s \overset{(q,g,q')}{\rightarrow} s' \wedge q'_\mathcal{A} \notin F \\ (\lambda^{\mathbb{B}}_{s',q'_\mathcal{A}} \wedge \lambda^{\#}_{s',q'_\mathcal{A}} > \lambda^{\#}_{s,q_\mathcal{A}})) & \text{if } (q_\mathcal{A}, s, q'_\mathcal{A}) \in \delta_\mathcal{A} \wedge s \overset{(q,g,q')}{\rightarrow} s' \wedge q'_\mathcal{A} \in F \\ \top) & \text{else} \end{cases}$$

Figure 5.4: The constraint $\phi_{correct}$ ensures that the generated system satisfies φ.

$lambdaB_{s,q_\mathcal{A}}$ check if the annotation function is valid for the reachable states of the product following the conditions from Definition 5.3.2. For these constraints, $s \overset{(q,g,q')}{\rightarrow} s'$ denotes that the global state s reaches s' by taking the local transition (q, g, q'). If a SAT-solver finds δ'_A and δ'_B such that there exists valid annotation function, then the generated system satisfies φ.

$$\exists\{costTrA_{q,q',c} | c \in \{0,\dots,k+1\}, q, q' \in Q_A\}$$
$$\exists\{costTrB_{q,q',c} | c \in \{0,\dots,k+1\}, q, q' \in Q_B\}$$
$$\exists\{costGuardA_{q,g,c} | c \in \{0,\dots,k+1\}, q \in Q_A, g \in Q_B\}$$
$$\exists\{costGuardB_{q,g,c} | c \in \{0,\dots,k+1\}, q \in Q_B, g \in Q_A \dot\cup Q_B\} : \phi_{cost}$$

$$\begin{aligned}\phi_{cost} = &\bigwedge_{q,q' \in Q_A, c \le k} rdTransA_{q,q',c} \wedge notRdTransA_{q,q',c} \wedge \neg costTrA_{q,q',k+1} \\ &\bigwedge_{q,q' \in Q_B, c \le k} rdTransB_{q,q',c} \wedge notRdTransB_{q,q',c} \wedge \neg costTrB_{q,q',k+1} \\ &\bigwedge_{q \in Q_A, g \in Q_B, c \le k} chGuardA_{q,g,c} \wedge notChGuardA_{q,g,c} \wedge \neg costGuardA_{q,g,k+1} \\ &\bigwedge_{q \in Q_B, g \in Q_A \dot\cup Q_B, c \le k} chGuardB_{q,g,c} \wedge notChGuardB_{q,g,c} \wedge \neg costGuardB_{q,g,k+1}\end{aligned}$$

Figure 5.5: The constraint ϕ_{cost} ensures that at most k-many operations are applied.

5.4.2 Constraint Solving for Minimal Repairs

A minimal repair for a system $A||B^n$ and φ, is a repair ξ such that there is no repair ξ' with $|\xi'| < |\xi|$. While $\phi_{correct}$ guarantees that the generated system $A'||B'^n$ satisfies φ, the constraint ϕ_{cost} in Figure 5.5 checks if for a given bound k, there is a consistent transformation ξ with $(A', B') = apply^*((A, B), \xi)$ and $|\xi| = k$. For better readability, assume wlog. that $Q_A = \{0,\dots,i\}$ and $Q_B = \{i+1,\dots,j\}$ for some $i, j \in \mathbb{N}$ with $i < j$. By using an implicit ordering over the possible operations, the constraints count the number of applied operations. The constraints start by counting the number of transition redirections of A, followed by the ones for B. Then, it counts the number of changed transition guards for A, followed by the ones for B. The variables $costTrA_{q,q',c}$, $costTrB_{q,q',c}$, $costGuardA_{q,g,c}$ and $costGuardB_{q,g,c}$ are true if the number of applied operations so far equals c. For $costTrA_{q,q',c}$ and $costTrB_{q,q',c}$, the current operation is a transition redirection for A or B from q to q'. Further, for $costGuardA_{q,g,c}$ and $costGuardB_{q,g,c}$, the current operation is a changed transition guard of A or B starting from q for g. The concrete operations and transformation can be extracted by comparing the given process templates A and B with the process templates returned by the SAT-solver. The bookkeeping to update the current cost is done by the constraints shown in Figure 5.6.

The constraints in Figure 5.7 check which operations are applied. The generated templates A', B' can be minimally obtained from A, B with the following consistent transformation. The constraints $transA_{q,q'}$ and $transB_{q,q'}$ represent a transition redirection from q to q' for all transitions $(q, g, q') \in \delta'_A$ that have been enabled for A or B in q but have reached a successor state different from q'.

$$rdTransA_{q,q',c} = \begin{cases} transA_{q,q'} \rightarrow costTrA_{q,q',1} & \text{if } q = q' = \mathsf{init}_A \\ transA_{q,q'} \wedge costTrA_{q-1,|Q_A|-1,c} \rightarrow costTrA_{q,q',c+1} & \text{if } q \neq \mathsf{init}_A, q' = \mathsf{init}_A \\ transA_{q,q'} \wedge costTrA_{q,q'-1,c} \rightarrow costTrA_{q,q',c+1} & \text{else} \end{cases}$$

$$notRdTransA_{q,q',c} = \begin{cases} \neg transA_{q,q'} \rightarrow costTrA_{q,q',0} & \text{if } q = q' = \mathsf{init}_A \\ \neg transA_{q,q'} \wedge costTrA_{q-1,|Q_A|-1,c} \rightarrow costTrA_{q,q',c} & \text{if } q \neq \mathsf{init}_A, q' = \mathsf{init}_A \\ \neg transA_{q,q'} \wedge costTrA_{q,q'-1,c} \rightarrow costTrA_{q,q',c} & \text{else} \end{cases}$$

$$rdTransB_{q,q',c} = \begin{cases} transB_{q,q'} \wedge costTrA_{|Q_A|-1,|Q_A|-1,c} \rightarrow costTrB_{q,q',c+1} & \text{if } q = q' = \mathsf{init}_B \\ transB_{q,q'} \wedge costTrB_{q-1,|Q_B|-1,c} \rightarrow costTrB_{q,q',c+1} & \text{if } q \neq \mathsf{init}_B, q' = \mathsf{init}_B \\ transB_{q,q'} \wedge costTrB_{q,q'-1,c} \rightarrow costTrB_{q,q',c+1} & \text{else} \end{cases}$$

$$notRdTransB_{q,q',c} = \begin{cases} \neg transB_{q,q'} \wedge costTrA_{|Q_A|-1,|Q_A|-1,c} \rightarrow costTrB_{q,q',c} & \text{if } q = q' = \mathsf{init}_B \\ \neg transB_{q,q'} \wedge costTrB_{q-1,|Q_B|-1,c} \rightarrow costTrB_{q,q',c} & \text{if } q \neq \mathsf{init}_B, q' = \mathsf{init}_B \\ \neg transB_{q,q'} \wedge costTrB_{q,q'-1,c} \rightarrow costTrB_{q,q',c} & \text{else} \end{cases}$$

$$chGuardA_{q,g,c} = \begin{cases} guardA_{q,g} \wedge costTrB_{|Q_B|-1,|Q_B|-1,c} \rightarrow costGuardA_{q,g,c+1} & \text{if } q = \mathsf{init}_A \wedge g = \mathsf{init}_B \\ guardA_{q,g} \wedge costGuardA_{q-1,|Q_B|-1,c} \rightarrow costGuardA_{q,g,c+1} & \text{if } q \neq \mathsf{init}_A \wedge g = \mathsf{init}_B \\ guardA_{q,g} \wedge costGuardA_{q,g-1,c} \rightarrow costGuardA_{q,g,c+1} & \text{else} \end{cases}$$

$$notChGuardA_{q,g,c} = \begin{cases} \neg guardA_{q,g} \wedge costTrB_{|Q_B|-1,|Q_B|-1,c} \rightarrow costGuardA_{q,g,c} & \text{if } q = \mathsf{init}_A \wedge g = \mathsf{init}_B \\ \neg guardA_{q,g} \wedge costGuardA_{q-1,|Q_B|-1,c} \rightarrow costGuardA_{q,g,c} & \text{if } q \neq \mathsf{init}_A \wedge g = \mathsf{init}_B \\ \neg guardA_{q,g} \wedge costGuardA_{q,g-1,c} \rightarrow costGuardA_{q,g,c} & \text{else} \end{cases}$$

$$chGuardB_{q,g,c} = \begin{cases} guardB_{q,g} \wedge costGuardA_{|Q_A|-1,|Q_B|-1,c} \rightarrow costGuardB_{q,g,c+1} & \text{if } q = \mathsf{init}_B \wedge g = \mathsf{init}_A \\ guardB_{q,g} \wedge costGuardB_{q-1,|Q_B|-1,c} \rightarrow costGuardA_{q,g,c+1} & \text{if } q \neq \mathsf{init}_B \wedge g = \mathsf{init}_A \\ guardB_{q,g} \wedge costGuardB_{q,g-1,c} \rightarrow costGuardA_{q,g,c+1} & \text{if } q \neq \mathsf{init}_B \wedge g \neq \mathsf{init}_A \wedge g \in Q_A \\ guardB_{q,g} \wedge costGuardB_{q,|Q_A|-1,c} \rightarrow costGuardB_{q,g,c+1} & \text{if } g = \mathsf{init}_B \\ guardB_{q,g} \wedge costGuardB_{q,g-1,c} \rightarrow costGuardB_{q,g,c+1} & \text{else} \end{cases}$$

$$notChGuardB_{q,g,c} = \begin{cases} \neg guardB_{q,g} \wedge costGuardA_{|Q_A|-1,|Q_B|-1,c} \rightarrow costGuardB_{q,g,c} & \text{if } q = \mathsf{init}_B \wedge g = \mathsf{init}_A \\ \neg guardB_{q,g} \wedge costGuardB_{q-1,|Q_B|-1,c} \rightarrow costGuardA_{q,g,c} & \text{if } q \neq \mathsf{init}_B \wedge g = \mathsf{init}_A \\ \neg guardB_{q,g} \wedge costGuardB_{q,g-1,c} \rightarrow costGuardA_{q,g,c} & \text{if } q \neq \mathsf{init}_B \wedge g \neq \mathsf{init}_A \wedge g \in Q_A \\ \neg guardB_{q,g} \wedge costGuardB_{q,|Q_A|-1,c} \rightarrow costGuardB_{q,g,c} & \text{if } g = \mathsf{init}_B \\ \neg guardB_{q,g} \wedge costGuardB_{q,g-1,c} \rightarrow costGuardB_{q,g,c} & \text{else} \end{cases}$$

Figure 5.6: Constraints for updating the cost of applied operations used for ϕ_{cost}

Further, the constraints $guardA_{q,g}$ and $guardB_{q,g}$ represent a changed transition guard operation of A or B in q for guard g where either g is disabled in A' or B' for all transitions in q, and g is enabled for some transition of A or B in q, or g is enabled in A', B' for some transition starting in q, and g is disabled in A, B for all transitions starting in q. If a SAT-solver can find a transformation ξ with $\xi = k$ that satisfies ϕ_{cost}, then $(A', B') = apply^*((A, B), \xi)$. By increasing the bound k of allowed operations, the repair algorithm constructs minimal repairs.

5.4.3 Deadlock Detection

To generate minimal repairs that are globally deadlock-free, the constraint system can be extended with a constraint $\phi_{deadlock\text{-}free}$. The constraint works analogously to the one in Figure 4.5. It encodes a reachability analysis of the global state space of $A'||B'^c$. For every global state $s \in Q_A \times (Q_B)^c$ that is reachable, there

$$transA_{q,q'} = \bigvee_{g \in Q_B} \begin{cases} \delta'_{A_{q,g,q'}} & \text{if } (q,g,q') \notin \delta_A \wedge \exists q'' \in Q_A : (q,g,q'') \in \delta_A \\ \bot & \text{else} \end{cases}$$

$$transB_{q,q'} = \bigvee_{g \in Q_A \dot{\cup} Q_B} \begin{cases} \delta'_{B_{q,g,q'}} & \text{if } (q,g,q') \notin \delta_B \wedge \exists q'' \in Q_B : (q,g,q'') \in \delta_B \\ \bot & \text{else} \end{cases}$$

$$guardA_{q,g} = \begin{cases} \bigwedge_{q' \in Q_A} \neg \delta'_{A_{q,g,q'}} & \text{if } \exists q'' \in Q_A : (q,g,q'') \in \delta_A \\ \bigvee_{q' \in Q_A} \delta'_{A_{q,g,q'}} & \text{else} \end{cases}$$

$$guardB_{q,g} = \begin{cases} \bigwedge_{q' \in Q_B} \neg \delta'_{B_{q,g,q'}} & \text{if } \exists q'' \in Q_B : (q,g,q'') \in \delta_B \\ \bigvee_{q' \in Q_B} \delta'_{B_{q,g,q'}} & \text{else} \end{cases}$$

Figure 5.7: Constraints for checking which operations are applied

has to be a local transition of A' or B' that is enabled in s. Otherwise, s is globally deadlocked.

5.4.4 Parameterized Minimal Repair Algorithm

Given process templates A, B, a parameterized specification and initial constraints *initConstraint*, Algorithm 5 shows how to construct a consistent transformation that minimally repairs A, B for φ. The initial constraint is a user-designed constraint to specify the resulting system, e.g., to preserve certain transitions. The algorithm starts by computing a cutoff c for A, B and φ, and by building a universal co-Büchi automaton $\mathcal{A}$ for φ. Then, the constraint system is extended with the constraint $\phi_{correct}$ and the modified constraint $\phi_{deadlock\text{-}free}$ from Figures 5.4 and 4.5 to verify correctness of the generated system $A'||B'^c$ and to guarantee the absence of global deadlocks (Lines 8-11). To obtain a minimal repair, the algorithm checks for a consistent transformation that applies a bounded number of operations *currentCost*. For each bound, ϕ_{cost} from Figures 5.5, 5.6 and 5.7 is built in Line 14. If the SAT-solver finds a transformation ξ satisfying the constraints, then ξ is a minimal repair for A, B and φ. Otherwise, the algorithm checks for a transformation for the increased bound. If the SAT-solver cannot find a repair for the maximal bound of operations, the algorithm return *Unrealizable* to signal that there exists no repair for A, B and φ. By Lemma 5.2.1, a maximal bound is $(|Q_A| \cdot |Q_A|) \cdot (|Q_B| \cdot |Q_B|)$ when changing the guard for every transition. By introducing the maximal bound of applied operations, the algorithm always terminates.

Theorem 5.4.1 states that Algorithm 5 is sound. This follows from Theorem 5.3.1 and by bounding the number of operations of the generated transformation. Furthermore, from Lemma 5.2.1 it follows that Algorithm 5 is completed which is stated by Theorem 5.4.2.

Theorem 5.4.1. [26]*(Soundness). For every repair ξ returned by Algorithm 5:*

Algorithm 5 Parameterized Minimal Repair

```
1:  procedure ParameterizedMinimalRepair(A, B, φ, initConstraint)
2:      A' ← A, B' ← B, accumConstraint ← initConstraint, currentCost ← 1
3:      // compute the cutoff
4:      cutoff ← ComputeCutoff(A, B, φ)
5:      //build the universal co-Büchi automaton 𝒜 for φ
6:      𝒜 ← BuildAutomaton(φ)
7:      //build φ_correct and φ_deadlock-free
8:      correctConstraint ← BuildCorrectnessConstraint(A, B, cutoff, 𝒜)
9:      accumConstraint ← accumConstraint ∧ correctConstraint
10:     deadlockConstraint ← BuildDeadlockConstraint(A, B, cutoff)
11:     accumConstraint ← accumConstraint ∧ deadlockConstraint
12:     //check for transformations that apply currentCost-many operations
13:     while currentCost ≤ maxCostBound(A, B) do
14:         costConstraint ← BuildCostConstr(A, B, currentCost)
15:         (ξ, isSat) ← SAT(accumConstraint ∧ costConstraint)
16:         if !isSat then
17:             currentCost ← currentCost+1
18:         else
19:             return ξ
20:     return Unrealizable
```

- *ξ is a minimal repair for A, B and φ, and*
- *the transition relation of apply*$((A, B), \xi)$ is total.*

Theorem 5.4.2. *(Completeness). If Algorithm 5 returns Unrealizable, then the paramaterized system has no repair.*

5.5 Extensions and Limitations

In the following, we discuss how the presented operation-based repair algorithm can be modified to minimally repair system classes that go beyond disjunctive systems. Furthermore, the limitations of the repair approach are shown.

Algorithm 5 can be used to repair systems for any property that can be expressed as an LTL\X-formula. These properties include safety and liveness properties. In contrast to the refinement-based repair algorithms, i.e., Algorithms 3 and 4, Algorithm 5 generates a repair transformation if and only if there exist process templates preserving the given structure such that the resulting paramaterized system satisfies the given specification. This is shown by Theorems 5.4.1

and 5.4.2, and Lemma 5.2.1. Further, the algorithm can be extended to other system classes where there exists a cutoff for model checking, including conjunctive systems by interpreting the guards conjunctively [35]. The algorithm can also be modified for pairwise-rendezvous system if there exists a cutoff for a given system and specification. However, as mentioned in Section 4.6, this is not necessarily the case as shown by Aminof et al. [2]. If there exists a cutoff, for synchronous transitions, the totality constraint of ϕ_{correct} has to be modified to ensure that the repair is total, as described in Section 3.5.2. Furthermore, similar operations and constraints have to be defined for synchronous transitions. However, Algorithm 5 cannot be extended to repair broadcast systems for liveness properties since the parameterized model checking problem is undecidable for broadcast systems and liveness properties [13]. Furthermore, there exist cutoffs for disjunctive and conjunctive systems for labeled process templates, i.e., for labeled process templates, a transition reacts to a corresponding input from the environment [6]. While it is possible to convert labeled process templates to process templates for Definition 2.1.1 that are equivalent under LTL-properties, this comes at the cost of a blow-up. Thus, it is more efficient to find a repair transformation for labeled process templates instead. In contrast to repairing the presented systems, to repair systems for labeled process templates, transition redirections are essential. When modifying the operations to include inputs for transitions, Algorithm 5 can minimally repair disjunctive and conjunctive systems for labeled process templates. For process templates labeling states, the operation-based repair approach needs to introduce a *state labeling change* operation to repair any system. Intuitively, this operation changes the label of a local state and can be defined analogously to the state labeling change operation introduced by Baumeister et al. [10].

Chapter 6

Implementation and Evaluation

In this chapter, we present our prototype implementation of Algorithms 4 and 5 to repair disjunctive systems. Furthermore, we evaluate both repair algorithms on a range of benchmarks and discuss the observable results.

6.1 Prototype Implementation

We implemented the refinement-based and operation-based parameterized repair approach, i.e., Algorithms 4 and 5, into the bounded synthesis tool **BoSy** [27] implemented in the programming language Swift [32]. Thereby, we can use the existing structures to generate constraint systems. Furthermore, we can make use of the algorithms for automata construction and SAT-based constraint solving that are provided by **BoSy**. To repair disjunctive systems, we modified the program input to include a specification φ and two process templates A and B. Since both algorithms verify correctness for the cutoff-sized system $A||B^c$, we implemented a procedure that computes the global state space of $A||B^c$. To significantly reduce the number of global states, we represent global states by configurations. In contrast to the configurations in Definition 3.2.1, we explicitly store the local state for each B-process that is specified by φ. This representation is also used in Example 3.5.1. Note that while for the model checking algorithm in Algorithm 4 only the reachable global state space is computed, the prototype implementation computes the entire global state space for encoding a valid annotation function used in Algorithm 5. Since Algorithm 4 contains no deadlock detection, the constraint $\phi_{\text{deadlock-free}}$ from Figure 4.5 is added to the constraint system to generate deadlock-free repairs.

6.2 Experimental Results

In this section, we present the benchmarks used to evaluate our prototype implementation for the refinement-based and operation-based parameterized repair

approach. After giving the technical details, we discuss the observations of the experimental results.

6.2.1 Benchmarks

The experimental results presented in Table 6.1, correspond to the following benchmarks:

- *ReaderWriter*: This disjunctive system is our running example with specifications used throughout this thesis. The systems for the benchmarks denoted by $ReaderWriter_{safe}$, are repaired for safety properties, whereas the ones for $ReaderWriter_{live}$ are repaired for liveness properties. The benchmark is scaled by adding states dummy states for the reader process that need to be unreachable for a minimal repair.

- *SmokeDetector*: This disjunctive system consists of a controller for an alarm system and a parameterized number of smoke detectors. The systems are repaired for a specification that requires the controller to trigger an alarm if smoke is detected. The benchmark is scaled by adding states that detect different types of smoke.

- *Observer*: This disjunctive system consists of an observer process and a parameterized number of worker processes that have to solve different global tasks. The specification requires the observer process to signal when every task is completed. The systems for the benchmarks denoted by $Observer_{complete}$, can be repaired for both repair approaches. However, for $Observer_{incomplete}$, there only exist repair transformations. The benchmark is scaled by adding more tasks that need to be completed.

6.2.2 Technical Details

We instantiate **BoSy** to use **ltl3ba** [7] as the converter from an LTL-specification to an automaton. As both constraint systems only contain existential quantifiers, CryptoMiniSat [40] is used as the SAT-solver. The benchmark results were obtained on a single dual-core Intel i5 processor with 3.10GHz and 16 GB RAM. A timeout of 5 minutes is used.

6.2.3 Observations

Table 6.1 shows the experimental results for our prototype implementation. For each benchmark, the table contains the combined number of local states and transitions for both process templates. Furthermore, it records the number of global states for the entire global state space of the cutoff-sized system. It also

Benchmark	**Size**				**Refinement**		**Operation**	
	local states	transitions	global states	states autom.	removed trans.	time in sec.	chG/rdT	time in sec.
$ReaderWriter_{safe}$	4	13	24	4	3	1.46	1/0	**1.29**
$ReaderWriter_{safe}$	5	20	180	4	4	**2.46**	1/0	5.21
$ReaderWriter_{safe}$	6	27	1120	4	5	**27.91**	1/0	50.90
$ReaderWriter_{live}$	4	13	10	3	3	1.38	1/0	**1.24**
$ReaderWriter_{live}$	5	18	42	3	4	2.28	1/0	**1.81**
$ReaderWriter_{live}$	6	23	168	3	5	**4.12**	1/0	5.89
$ReaderWriter_{live}$	7	28	660	3	6	209.18	1/0	**43.56**
SmokeDetector	4	6	16	3	2	**1.11**	1/0	1.12
SmokeDetector	5	8	90	3	3	**1.51**	2/0	2.02
SmokeDetector	6	10	448	3	4	**2.35**	2/0	11.03
SmokeDetector	7	12	2100	3	5	**19.46**	2/0	110.75
SmokeDetector	8	14	9504	3	6	**215.23**	-	timeout
$Observer_{complete}$	6	9	63	2	2	**1.26**	1/0	1.51
$Observer_{complete}$	8	12	336	2	3	**2.12**	1/0	4.48
$Observer_{complete}$	10	15	1650	2	4	**4.05**	1/0	55.02
$Observer_{complete}$	12	18	7722	2	5	**13.34**	-	timeout
$Observer_{incomplete}$	6	8	63	2	unrealizable		1/1	**1.58**
$Observer_{incomplete}$	8	10	336	2	unrealizable		1/1	**6.34**
$Observer_{incomplete}$	10	12	1650	2	unrealizable		1/1	**84.01**
$Observer_{incomplete}$	12	14	7722	2	unrealizable		-	timeout

Table 6.1: Benchmarking results of the refinement-based and operation-based parameterized repair approach

reports the size of the constructed automata. Note that the size of the nondeterministic Büchi automaton for the negated specification $\neg\varphi$ that is used for model checking finite-state systems, equals the universal co-Büchi automaton used for encoding an annotation function, as they are dual. For the refinement-based parameterized repair approach, Table 6.1 shows the number of transitions that is removed by the returned minimal repair. For the operation-based parameterized repair approach, the number of generated changed transition guard operations and transition redirections is recorded that minimally repair the system. For both algorithms, the runtime is reported in seconds.

The experimental results for $ReaderWriter_{safe}$ reveal that both approaches can minimally repair the reader and writer process in a similar time as Algorithm 3 which generates an arbitrary repair [36]. However, for safety properties and more complex benchmarks, we cannot compare our approaches with the prototype implementation of Algorithm 3 since it was only evaluated for broadcast protocols. The benchmark results for all benchmark families reveal that in general, the refinement-based parameterized repair approach scales significantly better than the operation-based one. This can be explained by the fact that the entire global state space used to encode annotation functions is significantly bigger than the reachable one used for model checking the cutoff-sized system. However, for cutoff-sized systems with a small global state space, Algorithm 5 can compete

with Algorithm 4. Furthermore, for the benchmark family $ReaderWrite_{\text{live}}$, Algorithm 5 scales significantly better because for these benchmarks, the reachable global state space is close to the entire one and multiple removed transitions can be represented by a single changed transition guard. This observation can be seen for all benchmarks since every minimal repair transformation applies less operations than the minimal repair generated by Algorithm 4 removes transitions. Moreover, note that all generated repair transformations only apply changed transition guard operations, except for the benchmark family $Observer_{\text{incomplete}}$. This is due to the fact, that all of these benchmarks contain process templates that include all transitions needed for a refinement-based parameterized repair. Thus, to obtain a minimal repair transformation it is most efficient to remove transitions with changed transition guard operations. However, when the given process templates need to add transitions for more communication, a minimal repair transformation also applies transition redirections, shown by the benchmark family $Observer_{\text{incomplete}}$. For these benchmarks, Algorithm 4 cannot generate a repair.

To summarize, Table 6.1 reveals that for disjunctive systems where the reachable state space is significantly smaller than the entire one, Algorithm 4 performs significantly better than Algorithm 5. To generate minimal repair transformations for complex systems more efficiently, it may be beneficial to introduce a parameterized repair approach that iteratively generates candidate transformations and model checks the transformed cutoff-size system, similar to Algorithm 4. Then, for verifying correctness, it is sufficient to compute the reachable global state space of the restricted system, rather than the entire one used for encoding a valid annotation in Algorithm 5.

Chapter 7

Related Work

The parameterized model checking problem is to decide if a system with a parametric number of processes n satisfies a specification, regardless of n. While this problem has been shown to be undecidable even when restricting systems to uniform finite state processes [41], there exist several approaches deciding the problem for restricted classes of systems and properties [1, 2, 3, 15, 21, 22, 23, 25, 31] that are collected in several surveys [13, 16, 24]. These parameterized model checking approaches include methods that reduce the problem to finite-state model checking for the cutoff-sized system. For these approaches, a cutoff c is a natural number which ensures that the same correctness guarantees hold for all systems containing at least c-many processes. Several cutoff results for guarded protocols are shown in [2, 6, 21, 22, 34, 35].

There have been considered many automatic repair approaches where most of them are restricted to monolithic systems [4, 14, 19, 29, 33, 37, 39], e.g., in [14], the repair removes edges from the transition system and in [37], an expression or a left-hand side of an assignment is assumed to be erroneous and replaced by one that satisfies the specification. Moreover, there are multiple approaches for repairing concurrent systems and synchronization synthesis. Some of them are based on automata-theoretic synthesis [9, 28], whereas other approaches are based on a counterexample-guided synthesis or repair method [11, 43]. The repair approach in [30] performs verification and repair simultaneously. By using a learning-based algorithm, for every iteration the system is changed in a way that brings it closer to satisfying the specification. In contrast to our repair approaches, the algorithm is not guaranteed to terminate. To the best of our knowledge, the approaches introduced in this thesis and the approach in [36] are the only one that provide correctness guarantees for systems with a parametric number of processes. Whereas the synthesis approach in [28] produces size-optimal reactive systems that inherently satisfy a given specification, e.g., by encoding a valid annotation function [26], the parameterized synthesis approach investigates the synthesis problem for distributed architectures with a parametric number of finite-state components [5, 12, 20]. While the parameterized synthesis

approach generates an arbitrary system satisfying the given specification, the parameterized repair approach constructs a repaired system that is close to a given one.

Chapter 8

Conclusion

In this thesis, we introduced a parameterized repair approach for properties expressed as LTL\X-formulas including safety and liveness properties. For a given specification and incorrect parameterized system that is composed of an arbitray number n of processes, the approach generates a repair that provides correctness guarantees that hold regardless of n. These systems include systems with disjunctive and conjunctive guards. The presented parameterized repair algorithm is a modification of the repair approach for safety properties, introduced by Jacobs et al. [36], where a repair restricts a given nondeterministic system to eliminate faulty behavior. The modified algorithm interleaves the generation of candidate repairs with model checking of the cutoff-sized system. By extending the constraint system for constructing candidate repairs, repairs that introduce deadlocks are avoided. Furthermore, the algorithm guarantees to return minimal repairs by bounding the number of removed transitions for the candidate repair. The modified algorithm is shown to be sound, complete and to terminate.

Moreover, to repair any faulty implementation, we introduced a paramaterized repair approach that can automatically add behavior including more communication between processes. It has been shown that by constructing a repair transformation that applies a set of operations for an incorrect system, any system can be repaired if and only if there exists a system that satisfies the given specification while preserving the given structure. We presented a parameterized repair algorithm that generates repair transformations where correctness of the repaired cutoff-sized system is witnessed by encoding a valid annotation function. Furthermore, by extending the constraint system and bounding the number of operations applied by the generated transformation, the algorithm constructs minimal repair transformations that are deadlock-free. The algorithm is shown to be sound, complete and to terminate.

We implemented both approaches as an extension to the synthesis tool **BoSy** [27] and evaluated them on a range of benchmarks. The experimental results revealed that repairing systems by restricting their behavior is more beneficial than by applying a set of operations, in cases where the reachable global state

space of the cutoff-sized system is significantly smaller than the entire one.

In future work, we plan to investigate the parameterized repair problem for larger classes of systems that communicate with different synchronizations. Furthermore, we would like to repair real-time systems where correctness could be verified by timed automata. Moreover, we want to develop an explainable repair approach for parameterized systems that provides rich visual feedback to the designer, explaining the generated repair. Such an explanation could consist of incorrect system executions that are eliminated by the repair. Further types of explanations could include quantitive and symbolic explanations. For parameterized systems, we plan to develop a paramaterized synthesis approach that is based on the techniques and methods introduced in this thesis. The parameterized synthesis approach automatically constructs a system inherently satisfying a given specification, regardless of the number of processes.

Bibliography

[1] Benjamin Aminof, Swen Jacobs, Ayrat Khalimov, and Sasha Rubin. Parameterized model checking of token-passing systems. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 262–281. Springer, 2014.

[2] Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. Parameterized model checking of rendezvous systems. *Distributed Computing*, 31(3):187–222, 2018.

[3] Benjamin Aminof and Sasha Rubin. Model checking parameterised multi-token systems via the composition method. In *International Joint Conference on Automated Reasoning*, pages 499–515. Springer, 2016.

[4] 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.

[5] Paul C Attie and E Allen Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):51–115, 1998.

[6] Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. Tight cutoffs for guarded protocols with fairness. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 476–494. Springer, 2016.

[7] Tomáš Babiak, Mojmír Křetínskỳ, Vojtěch Řehák, and Jan Strejček. Ltl to büchi automata translation: Fast and more deterministic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 95–109. Springer, 2012.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] Suguman Bansal, Kedar S Namjoshi, and Yaniv Sa'ar. Synthesis of coordination programs from linear temporal specifications. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[10] Tom Baumeister, Bernd Finkbeiner, and Hazem Torfah. Explainable reactive synthesis. In *International Symposium on Automated Technology for Verification and Analysis*, pages 413–428. Springer, 2020.

[11] 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.

[12] Roderick Bloem and Swen Jacobs. Parameterized synthesis. *Logical Methods in Computer Science*, 10, 2014.

[13] 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*, 6(1):1–170, 2015.

[14] Borzoo Bonakdarpour and Bernd Finkbeiner. Program repair for hyperproperties. In *International Symposium on Automated Technology for Verification and Analysis*, pages 423–441. Springer, 2019.

[15] Edmund Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by network decomposition. In *International Conference on Concurrency Theory*, pages 276–291. Springer, 2004.

[16] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

[17] Wojciech Czerwiński, Sławomir Lasota, Ranko Lazić, JÉrôme Leroux, and Filip Mazowiecki. The reachability problem for petri nets is not elementary. *J. ACM*, 68(1), 2020.

[18] Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR. LNCS, vol. 6269*, pages 313–327. Springer, 2010.

[19] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. *Acm sigplan notices*, 38(11):78–95, 2003.

[20] Allen Emerson and Jai Srinivasan. A decidable temporal logic to reason about many processes. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 233–246, 1990.

[21] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE*, 2000.

[22] E Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 361–370. IEEE, 2003.

[23] E Allen Emerson and Kedar S Namjoshi. On reasoning about rings. *International Journal of Foundations of Computer Science*, 14(04):527–549, 2003.

[24] Javier Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[25] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 352–359. IEEE, 1999.

[26] Peter Faymonville, Bernd Finkbeiner, Markus N Rabe, and Leander Tentrup. Encodings of bounded synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 354–370. Springer, 2017.

[27] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. Bosy: An experimentation framework for bounded synthesis. In *International Conference on Computer Aided Verification*, pages 325–332. Springer, 2017.

[28] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5):519–539, 2013.

[29] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, 2009.

[30] 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.

[31] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[32] James Goodwill and Wesley Matlock. The swift programming language. In *Beginning Swift Games Development for iOS*, pages 219–244. Springer, 2015.