Refinement of TSL Specifications for LTL Synthesis

Saarland University

Department of Computer Science

Bachelor's Thesis

submitted by

Jonas Linn

Saarbrücken, September 2022

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Malte Schledjewski
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Dr. Benjamin Kaminski
Submission:	September 12, 2022

Abstract

Due to its high complexity, the synthesis problem is still a major challenge to computer science. Even with modern computers, it is barely possible to synthesize reactive systems that handle even small amounts of data using classical synthesis methods. The introduction of Temporal Stream Logic as a specification language is a big step in scalable reactive synthesis. It provides useful abstractions for data handling while being similar to LTL. The synthesis of a TSL specification is possible by generating an underapproximation of the specification in LTL, which is then refined in a CEGAR manner and synthesized by using bounded synthesis. However, the suggested algorithms have not yet been implemented.

In this thesis, we optimize the TSL synthesis process for implementation, while also presenting a parallelized version of the algorithm. Furthermore, we propose a way of generating initially stronger underapproximations for speeding up the CEGAR-loop. Additionally, a more generalized way of generating additional assumptions is introduced. Finally, we introduce a set of test specifications on which the implementations are benchmarked.

Acknowledgements

I want to thank Prof. Bernd Finkbeiner for sparking my interest in temporal logics and giving me the chance to work on this topic for my thesis. Also, I want to thank him as well as Prof. Benjamin Kaminski for reviewing this thesis. Furthermore, I want to thank my advisor Malte Schledjewski who was always a big help to me and who has provided guidance to me for writing this thesis. Moreover, I want to thank my family for always supporting me and giving me the opportunity to go to university. Special thanks go to my fiancée Julia, who has always believed in me and my work.

Erklärung
Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement
I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, 12 September, 2022

Contents

Chapter 1

Introduction

In their early history, computational systems focused on implementing functions and therefore, very simplified, took an input and eventually terminated with an output. In contrast to that, we also know reactive systems, which are typically designed to run without ever terminating and to constantly interact with their environment. This includes hardware circuits and communication protocols, as well as embedded systems [9]. Hence, nearly all computational systems we use today, may it be a smartphone, an ATM, or the engine controller in a car, build upon reactive systems (and are ones themselves). Implementation errors can consequently threaten the integrity of many systems used by billions of people every day. This is especially problematic for safety-critical systems.

A possible technique to eliminate those is formal verification, which aims on checking some critical properties on already developed systems. But this approach does not prevent implementation errors in the first place and there is no generalized way of fixing an error if one exists. In this case, some manual reprogramming is necessary. The synthesis of reactive systems on the other hand focuses on the automatic generation of reactive systems based on formal specifications. Such a specification does not specify how the system should be constructed but rather what the system should do.

The problem of constructing a system from a specification was formalized by Church in 1957 and is known as *synthesis problem* or *Church's problem* [12]. Formally, this means "the construction of a finite-state procedure that transforms any input sequence α letter by letter into an output sequence β such that the pair (α, β) satisfies the given specification" [23]. This problem was first solved by Büchi and Landweber in 1969 using specifications in *monadic second-order logic of one successor* (S1S). Unfortunately, the construction from S1S turned out to be very complex. In 1977 Pnueli introduced the concept of temporal reasoning over non-terminating cyclic programs [18], which are known today as *reactive systems*, and invented *Linear Temporal Logic* (LTL) as a formal way of arguing over the behavior of such systems. LTL allows specifying whether a condition on an execution path of a system is satisfied at the current or (n-th) next time step as well as if it will hold on any timestep in the future or if it holds for every time step. The introduction of LTL is a valuable step to a practical solution to Church's problem since it is not only much more intuitive to use than S1S, but the synthesis based on temporal specifications is also much less complex [9].

Even though the complexity of the synthesis problem was significantly reduced by using LTL to specify properties instead of S1S, its complexity is still double exponential [9]. Therefore the computational effort for real-world problems with multiple inputs and outputs is huge. Considering that in the traditional synthesis approach data has to be encoded in the states of the system, even for simple applications synthesis becomes unrealistically expensive in terms of computing time as well as size.

A promising way to address this problem is to abstract from the actual data processed by the system. More precisely, it is assumed that the actual implementation of functions is handled by another approach and the focus lies solely on synthesizing the control graph of a system.

This can be done using *Temporal Stream Logic* (TSL) [11]. The grammar of TSL is similar to the one of LTL, but it has been adapted to make statements about when data is handled by a function. However, it only refers abstractly to data handling, which is useful for specifying systems that handle larger amounts of data since data is explicitly not encoded in the states of the final system. Though, due to this abstraction, the synthesis problem for TSL specifications is in general undecidable.

Yet, it is feasible to generate an underapproximation in LTL for TSL specifications and to consequently obtain a solution through *bounded synthesis*. If this approximation is realizable, the resulting system is a solution to the TSL synthesis problem, if not, the counter-strategy may be inconsistent with the more expressive semantics of TSL and therefore spurious. Then additional assumptions are generated in a *counter-example guided abstraction refinement* (CEGAR) [7] manner. Eventually, either a realizable system is returned or a consistent counter-strategy is found. The TSL specification is then either realizable within a given bound or unrealizable respectively. Nonetheless, this is a task that, in most real-life cases, cannot be done by hand due to its complexity.

In this thesis, we implement the algorithm suggested in [11] to approximate and refine TSL specifications in LTL while using BoSy [8] as a synthesis tool. Prior to this, we optimize it (Sect. 3.1) for implementation by taking some practical considerations into account. →Sec. 3.1, p. 15

Furthermore, we introduce an adapted algorithm that aims at partly parallelizing the spuriousness check in Sect. 3.2. Additionally we suggest a procedure for generating a stronger initial approximation in Sect. 3.3, as well an alternative for generating more generalized assumptions Sect. 3.4. →Sec. 3.2, p. 20
→Sec. 3.3, p. 22
→Sec. 3.4, p. 24

Using these algorithms, we conduct benchmarks in Chapter 5 on a set of test specifications and evaluate their runtime and memory use. →Chapter 5, p. 31

Chapter 2

Preliminaries

2.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) was introduced by Pnueli [18] as a formal system of reasoning over infinite words. An infinite word consists of an infinitely long, ordered sequence of letters α of an alphabet Σ. This can for example be an execution path of a reactive system with each position in the word corresponding to the state of the system at a specific time step. LTL was designed to unify notations for program verification of sequential and concurrent programs together with making proof methods for verification significantly more intuitive. Additionally, it turned out to be useful as a specification logic for synthesis.

In the following subsections, the grammar and semantics of LTL are introduced. The notation is based on *Principles of Model Checking* [3], as it is commonly used.

2.1.1 Syntax

Operators in LTL are either boolean ($\neg, \wedge$) or temporal ($\bigcirc$, $\mathcal{U}$)

Definition 2.1 (LTL Formula)

An *LTL formula* ϕ over a set of Atomic Propositions AP is made up by the grammar

Def. LTL formula

$$\phi ::= \top \mid a \in AP \mid \neg\phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathcal{U}\, \phi.$$

2.1.2 Semantics

The boolean operators in LTL are semantically identical to those in boolean logic. Intuitively, the unary temporal operator *next* ($\bigcirc$) works as follows: ϕ *holds exactly then when*

ϕ *holds in the next step* (Fig. 2.1). The binary operator *until* ($\mathcal{U}$) states that $\phi_1 \; \mathcal{U} \; \phi_2$ *holds iff* ϕ_2 *holds anywhere in the future and* ϕ_1 *holds from the current step until* ϕ_2 *holds* (Fig. 2.4).
Let σ be an infinite word over 2^{AP} ($\sigma \in (2^{AP})^{\omega}$), then

$$\sigma \models \top$$

$$\sigma \models a \iff a \in \sigma(0)$$

$$\sigma \models \phi_1 \wedge \phi_2 \iff \phi_1 \models \sigma \wedge \phi_2 \models \sigma$$

$$\sigma \models \neg\phi \iff \sigma \not\models \phi$$

$$\sigma \models \bigcirc\phi \iff \sigma[1...] \models \phi$$

$$\sigma \models \phi_1 \; \mathcal{U} \; \phi_2 \iff \exists j \geqslant 0 . \sigma[j...] \models \phi_2 \wedge \sigma[i...] \models \phi_1, \forall 0 \leqslant i < j$$

From there on, we can introduce all other boolean operators such as $\vee, \rightarrow, \iff$ according to boolean logic. Additionally, we can use the temporal operator $\mathcal{U}$ to introduce the new temporal operators *finally* ($\Diamond$) and *globally* ($\Box$). We can infer them depending on $\mathcal{U}$ by

$$\Diamond\phi = \top \; \mathcal{U} \; \phi$$

$$\Box\phi = \neg\Diamond\neg\phi.$$

$\Diamond\phi$ can be described as ϕ *holds eventually in a future state* (Fig. 2.2) and $\Box\phi$ states that ϕ *holds in every state from now on* (Fig. 2.3).

2.1.3 Examples

In the following, there are some common examples of paths satisfying LTL specifications.

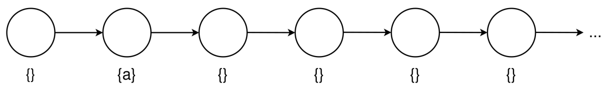


Figure 2.1: Path satisfying $\bigcirc a$

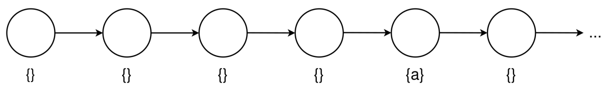


Figure 2.2: Path satisfying $\Diamond a$

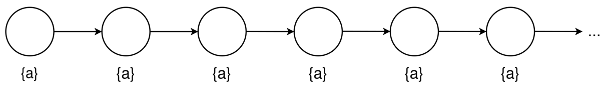


Figure 2.3: Path satisfying $\square a$

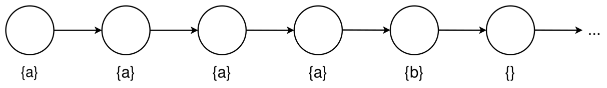


Figure 2.4: Path satisfying $a \ \mathcal{U} \ b$

2.2 Church's Problem

In contrast to classical programs - which are designed to eventually terminate with an output, given an input at the beginning - reactive systems transform an infinite stream of inputs word by word into an infinite stream of outputs. Formally, *Church's problem* [12] considers two distinct sets of inputs I and outputs O which are valuations of boolean variables. The problem is, given a specification $Spec \subseteq (2^{I \cup O})^{\omega}$, to generate a system that fulfills the given specifications. This problem was first solved for specifications in *monadic second-order logic of one successor* (S1S) [6], however, due to the expressiveness of S1S, the transformation from a specification to a system is very complex. Nonetheless, by using a temporal logic such as *Linear Temporal Logic* (LTL) as a specification language the complexity of the problem can be reduced.

2.3 Bounded Synthesis Problem

While theoretically, the implementation of an LTL specification can be up to doubly exponential in size, most realizable specifications can be implemented with much fewer states [9]. However, classical synthesis approaches do not necessarily produce minimal implementations. Furthermore, large implementations may not even be of interest because they may "violate design considerations, such as available memory" [20].

The *bounded synthesis problem* [20] is a modification of the synthesis problem that focuses on finding implementations within a predefined upper limit. The bounded synthesis problem can be reduced to the boolean SAT problem, which makes its solution comparatively efficient to implement. Moreover, it is possible to use bounded synthesis for finding minimal and, therefore, simply structured implementations. For distributed systems, where the synthesis problem is generally undecidable, bounded synthesis is an efficient semi-decision procedure.

In this thesis, bounded synthesis is used for all synthesis tasks.

2.4 Temporal Stream Logic (TSL)

Using LTL as a specification language for reactive synthesis comes with the downside of having to specify data handling in the system. Even in simple systems, the complexity of data handling can quickly lead to too large specifications to synthesize. However, in many cases, the data handling part of a system can be implemented by well-known and verified methods, which would make synthesizing these complicated parts unnecessary in the first place.

Temporal Stream Logic (TSL) [11] is designed to handle these problems by abstracting data handling, i.e., the specification only includes information on how data flows in the system, rather than on how it is computed. Its syntax and semantics build up on LTL, but it includes updates such as $[\![y \leftarrowtail f(x)]\!]$ and predicates over arbitrary function terms instead of atomic propositions. The implementation of functions and predicates is excluded from the synthesis problem. Hence, only the control flow graph of the system is synthesized. Therefore, this abstraction from actual data significantly reduces the number of states to be computed. However, the synthesis problem for TSL specifications is, in general, undecidable.

2.4.1 Terms in TSL

To formalize the handling of data, TSL introduces inputs $\mathbb{I}$, outputs $\mathbb{O}$, and cells $\mathbb{C}$. Additionally, there are function literals $\mathbb{F}$, and predicate literals $\mathbb{P}$ such that $\mathbb{P} \subseteq \mathbb{F}$.
Let $s_i \in \mathbb{I} \cup \mathbb{C}$, $s_o \in \mathbb{O} \cup \mathbb{C}$, $f \in \mathbb{F}$ and $p \in \mathbb{P}$ be symbolic representatives of their sets, then function terms τ_F, predicate terms τ_P and updates τ_U are defined as follows:

Definition 2.2 (Function Term)

Def. Function Term

A *function term* τ_F is either an input, a cell, or the application of a function to arbitrary many function terms.

$$\tau_F := s_i \mid f\ \tau_F^0\ \tau_F^1\ \dots\ \tau_F^{n-1}$$

Definition 2.3 (Predicate Term)

Def. Predicate Term

A *predicate term* τ_P is the application of a function to arbitrary many function terms.

$$\tau_P := p\ \tau_F^0\ \tau_F^1\ \dots\ \tau_F^{n-1}$$

Definition 2.4 (Update)

An *update* τ_U is an assignment from function terms τ_F to cells or outputs $s_o \in \mathbb{I} \cup \mathbb{O}$ **Def.** Update

$$\tau_U := [\![s_o \hookleftarrow \tau_F]\!]$$

The set of all function terms, predicate terms, and updates is denoted by $\mathcal{T}_F$, $\mathcal{T}_P$, and $\mathcal{T}_U$ respectively.

2.4.2 Syntax

Definition 2.5 (TSL Formula)

A *TSL formula* is made up by the following grammar: **Def.** TSL Formula

$$\phi := \tau \in \mathcal{T}_F \cup \mathcal{T}_U \mid \neg\phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathcal{U}\, \phi$$

2.4.3 Semantics

Definition 2.6 (Inputs)

A *momentary input* $i \in \mathcal{I} = \mathcal{V}^{[\mathbb{I}]}$ is an assignment of inputs $i \in \mathbb{I}$ to values $v \in \mathcal{V}$. An *Input stream* $\iota \in \mathcal{I}^\omega$ is an infinite sequence of momentary inputs. **Def.** Inputs **Def.** Input Stream

Definition 2.7 (Outputs)

A *momentary output* $o \in \mathcal{O} = \mathcal{V}^{[\mathbb{O}]}$ is an assignment of outputs $o \in \mathbb{O}$ to values $v \in \mathcal{V}$. An *output stream* $\rho \in \mathcal{O}^\omega$ is an infinite sequence of momentary outputs. **Def.** Outputs **Def.** Output Stream

Definition 2.8 (Computation)

A *computation step* $c \in \mathcal{C} = \mathcal{T}_F^{[\mathbb{O} \cup \mathbb{C}]}$ is an assignment of outputs and cells s_o to function terms $\tau_F \in \mathcal{T}_F$. A *computation* $\sigma \in \mathcal{C}^\omega$ is an infinite sequence of computation steps. **Def.** Computation Step **Def.** Computation

Definition 2.9 (Evalutation)

Let $\langle \cdot \rangle : \mathbb{F} \to \mathcal{F}$ be a function assignment and define $\forall c \in \mathbb{C} : \mathtt{init}_c \in \mathcal{F} \cap \mathcal{V}$ as an initial value for each cell.
The *evaluation function* $\eta_{\langle \cdot \rangle} : \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathcal{V}$ is given by: **Def.** Evaluation Function

$$\eta_{\langle\cdot\rangle}(\sigma,\iota,t,s_i) = \begin{cases} \iota(t)(s_i) & \text{if } s_i \in \mathbb{I} \\ \mathit{init}_{s_i} & \text{if } s_i \in \mathbb{C} \wedge t = 0 \\ \eta_{\langle\cdot\rangle}(\sigma,\iota,t-1,\sigma(t-1)(s_i)) & \text{if } s_i \in \mathbb{C} \wedge t > 0 \end{cases}$$

$$\eta_{\langle\cdot\rangle}(\sigma,\iota,t,f\ \tau_0\ \ldots\ \tau_{m-1}) = \langle f\rangle\ \eta_{\langle\cdot\rangle}(\sigma,\iota,t,\tau_0)\ldots\eta_{\langle\cdot\rangle}(\sigma,\iota,t,\tau_{m-1})$$

Then:

$$\forall t \in \mathbb{N}, o \in \mathbb{O} :$$

$$\rho_{\langle\cdot\rangle,\sigma,\iota}(t)(o) = \eta_{\langle\cdot\rangle}(\sigma,\iota,t,o)$$

Semantics Boolean operators follow standard boolean logic, temporal operators are defined analogously to LTL Sect. 2.1. Operators in TSL have the following semantics: → Sec. 2.1, p. 3

$$\begin{aligned}
\sigma,\iota,t \models_{\langle\cdot\rangle} p\ \tau_0\ \ldots\ \tau_{m-1} \quad &:\Leftrightarrow \eta_{\langle\cdot\rangle}(\sigma,\iota,t,p\ \tau_0\ \ldots\ \tau_{m-1}) \\
\sigma,\iota,t \models_{\langle\cdot\rangle} [\![s \leftarrow\!\!\prec \tau]\!] \quad &:\Leftrightarrow \sigma(t)(s) \equiv \tau \\
\sigma,\iota,t \models_{\langle\cdot\rangle} \neg\psi \quad &:\Leftrightarrow \sigma,\iota,t \not\models_{\langle\cdot\rangle} \psi \\
\sigma,\iota,t \models_{\langle\cdot\rangle} \vartheta \wedge \psi \quad &:\Leftrightarrow \sigma,\iota,t \models_{\langle\cdot\rangle} \vartheta \wedge \sigma,\iota,t \models_{\langle\cdot\rangle} \psi \\
\sigma,\iota,t \models_{\langle\cdot\rangle} \bigcirc\psi \quad &:\Leftrightarrow \sigma,\iota,t+1 \models_{\langle\cdot\rangle} \psi \\
\sigma,\iota,t \models_{\langle\cdot\rangle} \vartheta\ \mathcal{U}\ \psi \quad &:\Leftrightarrow \exists t'' \geqslant t.\ \forall t \leqslant t' < t''.\ \sigma,\iota,t' \models_{\langle\cdot\rangle} \vartheta \wedge \sigma,\iota,t'' \models_{\langle\cdot\rangle} \psi
\end{aligned}$$

Similar to LTL, other temporal and boolean operators can be derived.
A computation σ and an input stream ι satisfy a TSL formula φ if $\sigma,\iota,t \models_{\langle\cdot\rangle} \varphi$

Realizability The *realizability problem* of a TSL specification is defined over uninterpreted, arbitrary function terms, i.e. even though functions and predicates are part of a TSL specification, their definition is not considered for the TSL realizability problem. Note that since all functions and predicates are considered to be functional relations it must hold that they always evaluate to the same value given the same input.

A TSL Specification φ is realizable iff there is an implementation satisfying φ for all possible interpretations of functions and predicates.

Definition 2.10 (Realizability Problem for TSL)

The *realizability problem for TSL* is given by the following statement: Given a TSL formula φ, is there a strategy $\sigma \in \mathcal{C}^{[\mathcal{I}^+]}$. such that for every input $\iota \in \mathcal{I}^\omega$ and function implementation $\langle \cdot \rangle : \mathbb{F} \to \mathcal{F}$, the branch $\sigma \wr \iota$ satisfies φ, i.e.,

Def. Realizability Problem for TSL

$$\exists \sigma \in \mathcal{C}^{[\mathcal{I}^+]}.\ \forall \iota \in \mathcal{I}^\omega.\ \forall \langle \cdot \rangle : \mathbb{F} \to \mathcal{F}.\ \sigma \wr \iota, \iota \models_{\langle \cdot \rangle} \varphi$$

φ is called realizable iff a strategy σ exists.

Definition 2.11 (Realizability Problem for TSL)

The *synthesis problem for TSL* is to find concrete instantiation of σ in the realizability problem (Def. 2.10).

Def. Synthesis Problem for TSL
→ Def. 2.10, p. 11

2.4.4 Example

The following statement is an example of a TSL specification with an input i, cell c, function f and predicate p.

Example 2.4.1.

$$p(i)\ \mathcal{U}\ (\llbracket x \leftarrowtail f(i) \rrbracket \to \bigcirc p(x))$$

△

2.5 TSL Synthesis

To synthesize a TSL specification it is initially approximated by a weaker LTL specification which is then synthesized for a given bound using bounded synthesis. If this LTL specification is realizable, the solution is also a solution to the TSL synthesis problem. However, the LTL specification may be unrealizable even though the TSL specification is realizable. In this case, the counter-strategy given by the LTL-solver is called a *spurious counter-strategy*. The LTL approximation is then refined in a *counterexample-guided abstraction refinement* (CEGAR) [7] fashion until it is either realizable or a non-spurious counter-strategy is found. Note, that the specification may be only unrealizable for the given bound.

Theorem 1. *[11] If φ_{LTL} is realizable, then φ_{TSL} is realizable.*

2.5.1 Initial Approximation in LTL

For a TSL formula φ_{TSL}, the set of predicate terms, and the set of updates that appear in it are denoted by $\mathcal{T}_P$, $\mathcal{T}_U$ respectively. Then, the sets

$$\begin{aligned}
&\forall s_o \in \mathbb{O} \cup \mathbb{C}: && \mathcal{T}_U^{s_o} := \{[\![s_o \leftarrow\!\!\prec s_i]\!] \in \mathcal{T}_U \mid s_i \in \mathbb{I} \cup \mathbb{C}\},\\
&\forall c \in \mathbb{C}: && \mathcal{T}_{U/id}^{c} := \mathcal{T}_U^{c} \cup \{[\![c \leftarrow\!\!\prec c]\!]\},\\
&\forall o \in \mathbb{O}: && \mathcal{T}_{U/id}^{o} := \mathcal{T}_U^{o},\\
& && \mathcal{T}_{U/id} := \bigcup_{s_o \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}_{U/id}^{s_o}
\end{aligned}$$

can be derived.

The *syntactic conversion* to LTL of a TSL formula is simply given by the formula itself, however reinterpreting updates and predicate terms as atomic propositions in LTL. Since updates lose their semantic meaning in the conversion, it must be made sure that a cell or an output is only written to once per timestep.

Definition 2.12 (Initial LTL Approximation)

Def. Initial LTL Approximation

The *initial LTL approximation* φ_{LTL} of a TSL specification φ_{TSL} is given by:

$$\varphi_{\text{LTL}} := \Box \left(\bigwedge_{s_o \in \mathbb{O} \cup \mathbb{C}} \ \bigvee_{\tau \in \mathcal{T}_{U/id}^{s_o}} \left(\tau \wedge \bigwedge_{\tau' \in \mathcal{T}_{U/id}^{s_o} \setminus \{\tau\}} \neg \tau \right)\right) \wedge \text{SyntacticConversion}(\varphi_{\text{TSL}})$$

2.5.2 Spurious Counter-Strategies

The semantic meaning of predicate and function terms is lost in the LTL approximation. In synthesis, this may lead to a counter-strategy that is called spurious, since it is not a valid counter-strategy for the underlying TSL specification. Such a counter-strategy exploits the semantics lost in the approximation from TSL to LTL.

Definition 2.13 (Spurious Counter-Strategy)

A *counter-strategy is spurious*, iff there is a branch $\pi \wr \sigma$ for some computation $\sigma \in \mathcal{C}^\omega$, for which the strategy chooses an inconsistent evaluation of two equal predicates applied to the same value (that therefore evaluate equally according to TSL semantics) at different points in time.

Def. Spurious Counter-Strategy

$$\exists \sigma \in \mathcal{C}^\omega.\ \exists t, t' \in \mathbb{N}.\ \exists \tau_P, \tau_P' \in \mathcal{T}_P.$$

$$\tau_P \in \pi(\sigma(0)\sigma(1)...\sigma(t-1)) \wedge \tau_P' \notin \pi(\sigma(0)\sigma(1)...\sigma(t'-1)) \wedge$$

$$\forall \langle \cdot \rangle : \mathbb{F} \to \mathcal{F}.\ \eta_{\langle \cdot \rangle}(\sigma, \pi \wr \sigma, t, \tau_P) \ = \ \eta_{\langle \cdot \rangle}(\sigma, \pi \wr \sigma, t', \tau_P')$$

2.5.3 Refinement

The specification is synthesized using bounded synthesis (Sect. 2.3). Therefore, only systems up to a given bound b are considered. Since the counter-strategy is represented by a finite state transition system with m states, it is sufficient to only check the responses of the system up to a depth of $m * b$ for spurious behavior. Any further exploration would only result in repetitive checking of some states.

→ Sec. 2.3, p. 7

Syntactic equivalence of the evaluation over identity for two predicate terms τ_P, τ_P' on a computation ν at times t, t' is a sufficient condition for these predicate terms evaluating equally for every possible definition of predicates and functions. Hence, a counter-strategy that includes τ_P at time t, but not τ_P' at time t' is spurious.

Using the following algorithm, the spuriousness of a counter-strategy can be determined.

Algorithm 1: Check-Spuriousness

```
**Input:** bound $b$, counter-strategy $\pi : \mathcal{C}^* \to 2^{\mathcal{T}_P}$ (finitely represented using $m$ states)
1 for $v \in \mathcal{C}^{m*b}$, $\tau_P, \tau'_P \in \mathcal{T}_P$, $t, t' \in \{0, 1, \ldots, m*b-1\}$ do
2     if $\eta_{\langle\cdot\rangle id}(v, \iota_{id}, t, \tau_P) \equiv \eta_{\langle\cdot\rangle id}(v, \iota_{id}, t', \tau'_P) \wedge$
        $\tau_P \in \pi(v_0 \ldots v_{t-1}) \wedge \tau'_P \notin \pi(v_0 \ldots v_{t'-1})$ then
3         $w \leftarrow \texttt{reduce}(v, \tau_P, t)$;
4         $w' \leftarrow \texttt{reduce}(v, \tau'_P, t')$;
5         return $\square(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w'_i \to (\bigcirc^t \tau_P \iff \bigcirc^{t'} \tau'_P))$ ;
6 return *"non-spurious"* ;
```

Chapter 3

Optimizations and Adjustments

3.1 General Optimizations for Implementation

Alg. 1 iterates over all possible combinations of computations, predicate terms, and timesteps. However, the order of execution is not specified. No matter the order, in the case of a non-spurious counter-strategy the algorithm will always return after the maximum possible number of iterations. Even in the case of a spurious counter-strategy, its spuriousness may not be determined until the last possible iteration. Nonetheless, by fixing a specific nesting for the respective loops, it is possible to implement the algorithm in an optimized way.

3.1.1 Removing Redundant Iterations

Per definition of Alg. 1 some checks of spuriousness are redundant. For two distinct computations $\nu, \nu' \in \mathcal{C}^{m*b}$ and two timesteps $t, t' \in \{0, 1, \ldots, m * b - 1\}$ with $\nu[...max(t, t')] = \nu'[...max(t, t')]$, the algorithm iterates over both computations, even though it would be sufficient to check only one. To counteract this, the iteration over computations must be nested within the iteration over timesteps, while also limiting the length of a computation to $max(t, t')$, i.e. $\nu \in \mathcal{C}^{max(t,t')}$. This eliminates redundant iterations in Alg. 1.

3.1.2 Adjusting Spuriousness Check

The number of iterations necessary can be reduced further. For two pairs of predicate terms $(\tau_{P1}, \tau'_{P1}), (\tau_{P2}, \tau'_{P2}) \in \mathcal{T}_P$, two pairs of timesteps $(t_1, t'_1), (t_2, t'_2) \in \{0, 1, \ldots, m * b -$

$1\}^2$ and a computation $\nu \in \mathcal{C}^{max(t,t')}$ with $\tau_{P1} = \tau'_{P2},\ \tau'_{P1} = \tau_{P2},\ t_1 = t'_2,\ t'_1 = t_2$ the spuriousness check is partially symmetrical.

$$\begin{aligned}
&\eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t_1, \tau_{P1}) \equiv \eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t'_1, \tau'_{P1}) \wedge \tau_{P1} \in \pi(\nu_0 \ldots \nu_{t_1-1}) \wedge \tau'_{P1} \notin \pi(\nu_0 \ldots \nu_{t'_1-1})\\
\vee&\eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t_2, \tau_{P2}) \equiv \eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t'_2, \tau'_{P2}) \wedge \tau_{P2} \in \pi(\nu_0 \ldots \nu_{t_2-1}) \wedge \tau'_{P2} \notin \pi(\nu_0 \ldots \nu_{t'_2-1})\\
&\iff\\
&\eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t_1, \tau_{P1}) \equiv \eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t'_1, \tau'_{P1})\\
\wedge&(\tau_{P1} \in \pi(\nu_0 \ldots \nu_{t_1-1}) \wedge \tau'_{P1} \notin \pi(\nu_0 \ldots \nu_{t'_1-1}) \vee \tau'_{P1} \in \pi(\nu_0 \ldots \nu_{t'_1-1}) \wedge \tau_{P1} \notin \pi(\nu_0 \ldots \nu_{t_1-1}))\\
&\iff\\
&\eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t_1, \tau_{P1}) \equiv \eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t'_1, \tau'_{P1}) \wedge (\tau_{P1} \in \pi(\nu_0 \ldots \nu_{t_1-1}) \oplus \tau'_{P1} \in \pi(\nu_0 \ldots \nu_{t'_1-1}))
\end{aligned}$$

Furthermore, the generated assumptions for both cases are equal.
It holds that

$$\begin{aligned}
w_1 &= \mathtt{reduce}(\nu, \tau_{P1}, t_1) = \mathtt{reduce}(\nu, \tau'_{P2}, t'_2) = w'_2\\
w'_1 &= \mathtt{reduce}(\nu, \tau'_{P1}, t'_1) = \mathtt{reduce}(\nu, \tau_{P2}, t_2) = w_2
\end{aligned}$$

then,

$$\begin{aligned}
&\Box(\bigwedge_{i=0}^{t_1-1} \bigcirc^i w_{1_i} \wedge \bigwedge_{i=0}^{t'_1-1} \bigcirc^i w'_{1_i} \rightarrow (\bigcirc^{t_1}\tau_{P1} \iff \bigcirc^{t'_1}\tau'_{P1}))\\
=&\Box(\bigwedge_{i=0}^{t'_2-1} \bigcirc^i w'_{2_i} \wedge \bigwedge_{i=0}^{t_2-1} \bigcirc^i w_{2_i} \rightarrow (\bigcirc^{t'_2}\tau'_{P2} \iff \bigcirc^{t_2}\tau_{P2}))
\end{aligned}$$

→ Alg. 1, p. 14

By using the above-mentioned spuriousness condition in Alg. 1 some iterations turn redundant. This allows to reduce iterations over timesteps to $(t, t') \in \{(t, t') \in \{0, 1, \ldots, m * b - 1\}^2 \mid t \geqslant t'\}$

The same approach could be used to cut down iterations by reducing the number of pairs of predicate terms. However, this would introduce an additional comparison in the check, whereas the above-mentioned technique strips a logical "and" and introduces a "xor" comparison, keeping the overall number of logical computations the same while reducing the number of iterations.

3.1.3 Partinioning Predicate Terms

For a pair of predicate terms τ_P, τ'_P the spuriousness check is performed even if these terms are made up by different predicates. However, a counter-strategy cannot be spurious along them since they will never evaluate to the same value for all possible definitions of their respective predicate. Hence, it is sufficient to only check for spuriousness between predicate terms with the same predicate. Thus, predicate terms are partitioned into $\uplus_{p\in\mathbb{P}}\mathcal{T}_P^p$, and another loop, iteration over predicates is introduced. This optimization significantly reduces the overall iteration count when there is more than one predicate in the specification.

3.1.4 Caching Counter-strategy State

In every iteration of the algorithm, it is checked for two predicate terms whether they are an element of the counter-strategy at a specific computation. To perform this check it is necessary to play against the counter-strategy with the given computation. This reveals the set of all predicate terms that hold at this point. Hence, it is sufficient to perform the necessary computations only once per combination of (t, t') and $\nu \in \mathcal{C}^t$. The result can then be cached and reused for all iterations over predicate terms.

3.1.5 Timestep Order

Consider the TSL Specification

$$\begin{aligned}
\mathbb{I} &= \{x\}\\
\mathbb{C} &= \{y\}\\
\mathbb{O} &= \emptyset\\
\varphi_{TSL} &= \square(\llbracket y \leftarrowtail x \rrbracket \lor \llbracket y \leftarrowtail y \rrbracket) \land (\lozenge p(x) \rightarrow \lozenge p(y))
\end{aligned}$$

whose initial approximation in LTL is not realizable. When the counter-strategy of the initial approximation is checked for spuriousness on $(t, t') = (1, 2)$, it is correctly identified as spurious and the additional assumption

$$\square(\bigcirc\, i\_to\_a \rightarrow (\bigcirc\, p\_i \iff \bigcirc\bigcirc\, p\_a))$$

is generated. However, this additional assumption is neither minimal nor sufficient, i.e. the initial approximation in LTL is still not realizable under this assumption. Another refinement step is required to generate the additional assumption

$$\square(i\_to\_a \rightarrow (p\_i \iff \bigcirc\, p\_a)),$$

which would be sufficient by itself. Furthermore, since the complexity of synthesis is double exponential to the length of the LTL specification [17], the synthesis effort massively increases with an unnecessary large assumption. Hence, the order of timesteps for which a counter-strategy is checked heavily affects the performance of the algorithm. Yet, this only applies to checking counter-strategies that are spurious, since for a non-spurious counter-strategy all paths have to be checked for spuriousness before its non-spuriousness can be established.

Two predicate terms could evaluate inconsistently at any point. Nonetheless, timesteps can be ordered such that the generated assumption is minimal and no unnecessary assumptions are generated, which ensures that the number of refinements needed is minimal.

The order for iterations over timestep tuples is given by the statement:

$$\begin{aligned}&\forall (t,t'),(\tilde{t},\tilde{t}') \in \mathbb{N}^2 :\\ &(t,t') < (\tilde{t},\tilde{t}') := max(t,t') < max(\tilde{t},\tilde{t}') \vee max(t,t') = max(\tilde{t},\tilde{t}') \wedge min(t,t') < min(\tilde{t},\tilde{t}').\end{aligned}$$

Example 3.1.1. Ordered Timestep Tuples

$$(0,0),(0,1),(1,1),(0,2),(1,2),(2,2),(0,3),(1,3),(2,3),(3,3),(0,4),(1,4),(2,4),\ldots$$

$\triangle$

3.1.6 Optimized Algorithm

The stated optimizations also depend on a specific order of nesting of the loops and therefore define it. All these optimizations are implemented in Alg. 2.

Algorithm 2: Optimized Check-Spuriousness

```
   Input: bound $b$, counter-strategy $\pi : \mathcal{C}^* \to 2^{\mathcal{T}_P}$ (finitely represented using $m$
          states)
1  for $t' \in \{0, \dots, m * b - 1\}$ do
2      for $t \in \{0, \dots, t'\}$ do
3          for $v \in \mathcal{C}^t$ do
4              state $\leftarrow \pi(v_0 \dots v_{t-1})$ ;
5              state$' \leftarrow \pi(v_0 \dots v_{t'-1})$ ;
6              for $p \in \mathbb{P}$ do
7                  for $\tau_P, \tau'_P \in \mathcal{T}_P^p$ do
8                      if $\eta_{\langle \cdot \rangle \mathtt{id}}(v, \iota_{\mathtt{id}}, t, \tau_P) \equiv \eta_{\langle \cdot \rangle \mathtt{id}}(v, \iota_{\mathtt{id}}, t, \tau'_P) \wedge$
                        ( $\tau_P \in$ state $\oplus\ \tau'_P \in$ state) then
9                          $w \leftarrow \mathtt{reduce}(v, \tau_P, t)$ ;
10                         $w' \leftarrow \mathtt{reduce}(v, \tau'_P, t')$ ;
11                         return
                            $\square(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w'_i \to (\bigcirc^t \tau_P \iff \bigcirc^{t'} \tau'_P))$ ;
12 return "non-spurious" ;
```

3.2 Parallelization

All iterations performed in Alg. 2 are independent of each other. Consequently, it is possible to split iterations and run them in independent threads on multicore systems. Because of this independence, it would be possible to start a thread for every iteration. Anyway, the workload in each iteration is very small, so the parallelization itself must not generate too much overhead. It is furthermore important that as much workload as possible can be performed concurrently.

The following algorithm parallelizes Alg. 2 over pairs of timestamps (t, t'). The function *addOperation* adds the code inside to a queue, where it is executed in parallel. Its implementation is not discussed here since this would highly depend on the system the algorithm is run on. However, it is assumed, that it starts as many threads as it is possible to efficiently run to execute its given code blocks. Before more tasks are pushed to the queue, it is checked if there were already assumptions generated. Then, the algorithm returns immediately with the generated assumptions to prevent overhead.

Note, that read and write operations to assumptions have to be synchronized in order to prevent data races. Furthermore, the algorithm can return multiple assumptions at once, since the assumption generation itself is not, and should not be, synchronized since this would defeat the purpose of parallelization. However, as soon as an additional assumption is found it is written to a set of assumptions. All other threads then detect that an assumption has been found and terminate. The algorithm can then return the assumptions that have been generated.

Algorithm 3: Parallized Check-Spuriousness

```
Input: bound $b$, counter-strategy $\pi : \mathcal{C}^* \to 2^{\mathcal{T}_P}$ (finitely represented using $m$
       states)
assumptions $\leftarrow \{\}$ ;
for $t' \in \{0, \dots, m * b - 1\}$ do
    for $t \in \{0, \dots, t'\}$ do
        for $v \in \mathcal{C}^t$ do
            state $\leftarrow \pi(v_0 \dots v_{t-1})$ ;
            state$' \leftarrow \pi(v_0 \dots v_{t'-1})$ ;
            if $\neg$isEmpty(assumptions) then
                return (assumptions) ;
            addOperation( for $p \in \mathbb{P}$ do
                for $\tau_P, \tau'_P \in \mathcal{T}_P^p$ do
                    if $\eta_{\langle \cdot \rangle id}(v, \iota_{id}, t, \tau_P) \equiv \eta_{\langle \cdot \rangle id}(v, \iota_{id}, t, \tau'_P) \wedge$
                      ( $\tau_P \in$ state $\oplus\ \tau'_P \in$ state) then
                        $w \leftarrow$ reduce$(v, \tau_P, t)$ ;
                        $w' \leftarrow$ reduce$(v, \tau'_P, t')$ ;
                        assumptions.insert($\Box(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w'_i \rightarrow$
                          $(\bigcirc^t \tau_P \iff \bigcirc^{t'} \tau'_P)))$ ;
                        cancelAllOperations;
            )
        waitUntilAllOperationsAreFinished;
if isEmpty(assumptions) then
    return "non-spurious" ;
else
    return assumptions ;
```

3.3 Adding stronger initial assumptions

→ Alg. 1, p. 14

Spurious counter-strategies rely on an inconsistent evaluation of predicate terms and function terms in LTL since the LTL conversion loses its semantic meaning. In Alg. 1 this is counteracted by enforcing the correct behavior by adding an assumption based on the path of the spurious counter-strategy. However, it is possible to make further strengthening assumptions that prevent spurious behavior in some cases, solely based on the TSL specification without the necessity of path exploration of the counter-strategy. This is especially the case, but not limited, to TSL specifications that do not contain function terms. The idea behind this is to add assumptions that guarantee that predicate terms evaluate consistently with TSL semantics as long as the values are not modified by functions.

Let $\mathit{arg} : \mathcal{T}_U \to \mathcal{T}_F$ and $\mathit{dest} : \mathcal{T}_U \to \mathbb{O} \cup \mathbb{C}$ be defined by

$$\mathit{arg}([\![s_o \leftarrowtail \tau_F]\!]) := \tau_F$$

$$\mathit{dest}([\![s_o \leftarrowtail \tau_F]\!]) := s_o$$

Furthermore, let

$$\forall s_i \in \mathbb{I} \cup \mathbb{C} : \qquad \widetilde{\mathcal{T}_U}^{s_i} := \{[\![c \leftarrowtail s_i]\!]) \in \mathcal{T}_{U/id} \mid c \in \mathbb{C}\}$$

$$\begin{aligned}\widetilde{\mathcal{T}}_P := \{p(s_i^0 \dots s_i^{m-1}) \mid p \in \mathbb{P} \wedge p(\widetilde{s_i}^0 \dots \widetilde{s_i}^{m-1}) \in \mathcal{T}_P \wedge \\ \forall n < m.\ \exists \mu \in \mathcal{T}^*_{U/id}\ \exists t.\ \mathit{arg}(\mu(0)) = \widetilde{s_i}^n \wedge \\ \mathit{dest}(\mu(t)) = s_i^n \wedge \\ s_i^n \in \mathbb{C} \wedge \widetilde{s_i}^n \in \mathbb{I} \cup \mathbb{C}\}\end{aligned}$$

$$\begin{aligned}\forall p({s_i}^0 \dots {s_i}^{m-1}) \in \widetilde{\mathcal{T}}_P : \widetilde{\mathcal{T}_U}^{\tau_p} := \{(\tau_u^0 \dots \tau_u^{m-1}) \mid \forall n, n' < m.\ \tau_u^n \in \widetilde{\mathcal{T}_U}^{s_i^n} \wedge \\ (n = n' \vee \mathit{dest}(\tau_u^n) \neq \mathit{dest}(\tau_u^{n'}))\}\end{aligned}$$

Def. Stronger initial assumptions

Definition 3.1 (Stronger initial assumptions)
Stronger initial assumptions are given by

$$\bigwedge_{\tau_p \in \widetilde{\mathcal{T}}_P} \ \bigwedge_{(\tau_u^0 \dots \tau_U^{m-1}) \in \widetilde{\mathcal{T}_U}^{\tau_p}}$$

$$\square \left(\left(\bigwedge_{n=0}^{m-1} \tau_u^n \right) \to (p(\mathit{arg}(\tau_u^0) \dots \mathit{arg}(\tau_u^{n-1})) \iff \bigcirc p(\mathit{dest}(\tau_u^0) \dots \mathit{dest}(\tau_u^{n-1}))) \right)$$

This approach will most likely result in a more complex synthesis problem in LTL since it introduces new predicate terms. Additionally, since it may introduce predicate terms, the synthesis step is more expensive. Nevertheless, it may also result in a less expensive refinement process. Experimental results of this strategy are evaluated in Chapter 5. There are TSL formulas that are not expressible with LTL formulas of finite length [11]. However, when using bounded synthesis it is possible to generate assumptions that prevent all possible spurious counter-strategies. This is discussed in Sect. 3.5

→Chapter 5, p. 31

Corollary 2. *For any TSL specification φ with $\mathcal{T}_P = \mathcal{T}_F$ and $\forall \tau_u \in \mathcal{T}_U.\ \mathtt{arg}(\tau_u) \in \mathbb{O} \cup \mathbb{C}$ the synthesis problem of φ can be reduced to an LTL synthesis problem.*

3.4 Generalized Assumptions

→ Alg. 2, p. 19

The assumptions generated by Alg. 2 are specific for a sequence of computation steps. Another approach is to generate multiple assumptions that fix predicate values for the next time step based on their current evaluation and on updates. All assumptions combined then also imply the correct behaviour for the specific computation. However, this may introduce new predicate terms.

Algorithm 4: Optimized check-spuriousness with generalized assumptions

```
Input: bound $b$, counter-strategy $\pi : \mathcal{C}^* \to 2^{\mathcal{T}_P}$ (finitely represented using $m$
       states)
for $t' \in \{0, \dots, m * b - 1\}$ do
    for $t \in \{0, \dots, t'\}$ do
        for $\nu \in \mathcal{C}^t$ do
            state $\leftarrow \pi(\nu_0 \dots \nu_{t-1})$ ;
            state$' \leftarrow \pi(\nu_0 \dots \nu_{t'-1})$ ;
            for $p \in \mathbb{P}$ do
                for $\tau_P, \tau'_P \in \mathcal{T}_P^p$ do
                    if $\eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t, \tau_P) \equiv \eta_{\langle\cdot\rangle id}(\nu, \iota_{id}, t, \tau'_P) \wedge$
                      $(\tau_P \in$ state $\oplus\ \tau'_P \in$ state$)$ then
                        $w \leftarrow \mathtt{reduce}'(\nu, \tau_P, t)$ ;
                        $w' \leftarrow \mathtt{reduce}'(\nu, \tau'_P, t')$ ;
                        return $\square(\bigwedge_{i=0}^{t-1} w_i \to (p(arg(w_i)) \iff \bigcirc p(dest(w_i))))$;
return "non-spurious" ;
```

3.5 Generating Exhaustive Assumptions

Bounded synthesis is used to synthesize LTL approximations of TSL specifications. Thm. 1 states that a solution to the synthesis problem of the LTL approximation of a TSL specification is also a solution to the TSL synthesis problem. If a counter-strategy returned by the LTL solver is non-spurious, it also implies, that the TSL specification is unrealizable [11]. The fact that all spurious counter-strategies can be prevented by additional assumptions generated by Alg. 1 and that the number of possible additional assumptions is finite for a given bound leads to the following conclusion:

→ Thm. 1, p. 12

→ Alg. 1, p. 14

Corollary 3. *The bounded synthesis problem for TSL is decidable.*

The simplest method to generate exhaustive assumptions is to adapt Alg. 2 in a way that assumptions are added for all cases, independently of an existing counter-strategy. Note that in Alg. 2 t, and t' is limited by $m * b - 1$, where m is the size of the counter-strategy. However, the counter-strategy is omitted from the following algorithm. Any counter-strategy would nonetheless be also limited by b, as we use bounded synthesis. Hence, path exploration in the algorithm can be limited to a depth of $b^2 - 1$.

→ Alg. 2, p. 19

Algorithm 5: Generate exhaustive assumptions

```
   Input: bound $b$
1  assumptions ← {};
2  for $(t, t') \in \{(t, t') \in \{0, 1, \dots, b^2 - 1\}^2 \mid t \geqslant t'\}$ do
3      for $v \in \mathcal{C}^t$ do
4          for $p \in \mathbb{P}$ do
5              for $\tau_P, \tau'_P \in \mathcal{T}^p_P$ do
6                  if $\eta_{\langle\cdot\rangle id}(v, \iota_{id}, t, \tau_P) \equiv \eta_{\langle\cdot\rangle id}(v, \iota_{id}, t', \tau'_P)$ then
7                      $w \leftarrow$ reduce$(v, \tau_P, t)$ ;
8                      $w' \leftarrow$ reduce$(v, \tau'_P, t')$ ;
9                      assumptions.insert$(\square(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w'_i \rightarrow (\bigcirc^t \tau_P \iff \bigcirc^{t'} \tau'_P)))$ ;
10 return assumptions ;
```

The number of additional assumptions grows by the bound. Also, in almost any case many unnecessary assumptions are generated which would not be generated by Alg. 1. The reason for that is that additional assumptions are generated even for unreachable cases. Furthermore, the algorithm may generate many assumptions that are implied by smaller assumptions. Note, that this algorithm has not been implemented since its result would not be useful at all. Nonetheless, it is included here for the sake of completeness.

Chapter 4

Implementation

The TSL translation tool on which all tests were run is written in *Swift* [22]. The tool implements an object-oriented structure for TSL and LTL formulas. TSL specifications are parsed by an *ANTLR* [2] generated parser. LTL synthesis is handled by automatic calls to *BoSy* [8] [5]. In case a counter-strategy is returned by BoSy it is given as an *And-Inverter Graphs* (AIGs) [1]. The AIG is then parsed by the main tool, which also implements the exploration of the counter-strategy as needed for Alg. 2. → Alg. 2, p. 19

4.1 Main Algorithm

The algorithm starts by trying to synthesize the LTL conversion of the TSL specification by bounded synthesis with the given bound. In the case of a realizable strategy, the algorithm returns with a solution, if a counter-strategy is returned by BoSy it is checked for spuriousness. By definition of Alg. 2 a counter-strategy is checked up to depth $b*m-1$ where b is the bound and m is the number of states representing the counter-strategy. Since the counter-strategy returned by BoSy is given as an And-Inverter-Graph (AIG) [1], an upper bound for the number of states can be estimated by 2 to the power of the number of latches in the AIG. Note, that the implementation uses this estimate. Though it would be possible to determine the exact number of states by path exploration, which is not implemented in the tool due to its complexity.

→ Alg. 2, p. 19

```
let ltlTranslation = translator.getTranslation()
var bosy_result = Bosy.Result.UNKNOWN
var aigText = ""

while((bosy_result != Bosy.Result.REALIZABLE)){
    (bosy_result, aigText) = Bosy.synthesize(
        inputs: predicateTerms,
        outputs: translator.updates,
        bound: bound,
        assumptions: assumptions,
        guarantees: [ltlTranslation]
    )

    if (bosy_result == Bosy.Result.UNREALIZABLE)
    {
        let start_ref = Date()

        let aig = try AIG.parse(aag: aigText)!
        let m = aig.numberOfStates()

        var refiner : Refinement = Refinement(aig, updates,
   predicateTerms, bound*m)

        let assumptions_ = refiner.checkSpuriousness()
        predicateTerms = refiner.getPredicateTerms()

        // The counter-strategy is non-spurious if there are
   no additional assumptions
        if(assumptions_.isEmpty){
                print("non-spurious counter-strategy")
                break
            }

            assumptions = assumptions.concat(assumptions_)
        }

    }
```

Note, that some details of the implementation have been omitted from the code snippet above.

Predicate terms are treated as inputs for the LTL solver. This is because predicates are assumed to be controlled by the environment for LTL synthesis since their implementation is not fixed. BoSy does only accept alphanumericial values with underscores as names for atomic propositions. It is therefore not possible to keep the notation of updates, predicate terms, and function terms for their respective LTL conversion. Instead, they are transformed as follows:

$$\begin{aligned}
\mathrm{synCon}(f\ \tau_F^0\ \ldots\ \tau_F^{n-1}) &:= f + "\_" + \mathrm{synCon}(\tau_F^0) + "\_" + \ \ldots\ + \mathrm{synCon}(\tau_F^{n-1}) \\
\mathrm{synCon}(p\ \tau_F^0\ \ldots\ \tau_F^{n-1}) &:= p + "\_" + \mathrm{synCon}(\tau_F^0) + "\_" + \ \ldots\ + \mathrm{synCon}(\tau_F^{n-1}) \\
\mathrm{synCon}([\![s_o \leftarrow \tau_F]\!]) &:= \mathrm{synCon}(\tau_F) + "\_to\_" + \mathrm{synCon}(s_o)
\end{aligned}$$

4.1.1 BoSy

BoSy [8] is called by the main implementation over the command line. For this purpose, a helper class has been implemented that manages not only calls to BoSy but also parses the results. For more control over the process *BoSyBackend* is used as a synthesis tool. It is configured to use *ltl3ba* as an automaton tool, the search strategy is fixed to linear. The semantics for the output system is configured to *mealy*.

Chapter 5

Benchmarks

The benchmarks were conducted with *runsolver* [19] on an Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz with 32GB RAM. Timeout was set to 1800 seconds wallclock time and 3600 seconds CPU time. All tests were run at least three times to ensure the results were consistent. The values given in the graphs are average values from these runs.

The bound for specifications 1, 2, 3, and 4 was set to 16, while the bound for specification 5 was set to 8.

5.1 Specifications

This section introduces sets of test specifications, on which the benchmarks in Chapter 5 are performed. Note, that these specifications do not define any kind of useful system. Their only purpose is to serve as scalable examples for TSL synthesis.

The following notation is used for the n-times composition of a function with itself:

$$f^{\circ n} := \underbrace{f \circ \cdots \circ f}_{n \text{ times}}$$

Specification 1

$$\begin{aligned}
\mathbb{C} &:= \{c_0, ..., c_n\} \\
\mathbb{I} &:= \{in\} \\
\mathbb{O} &:= \{\} \\
spec^1_n &:= \Box(\llbracket c_0 \leftarrowtail in \rrbracket \vee (\bigvee_{\forall i \in \mathbb{N}, i < n} \llbracket c_{i+1} \leftarrowtail c_i \rrbracket) \vee (\bigvee_{\forall i \in \mathbb{N}, i \leqslant n} \llbracket c_i \leftarrowtail c_i \rrbracket)) \\
&\quad \wedge (\Diamond p(in) \rightarrow \Diamond p(c_n))
\end{aligned}$$

Specification 2

$$\begin{aligned}
\mathbb{C} &:= \{c_0, ..., c_n\} \\
\mathbb{I} &:= \{in\} \\
\mathbb{F} &:= \{f\} \\
\mathbb{O} &:= \{\} \\
spec^2_n &:= \Box(\llbracket c_0 \leftarrowtail in \rrbracket \vee (\bigvee_{\forall i \in \mathbb{N}, i < n} \llbracket c_{i+1} \leftarrowtail f(c_i) \rrbracket)) \wedge (\Diamond p(f^{\circ n}(in)) \rightarrow \Diamond p(c_i))
\end{aligned}$$

Specification 3 ($n > 1$)

$$\begin{aligned}
\mathbb{C} &:= \{c, c_0, ..., c_i\} \\
\mathbb{I} &:= \{in\} \\
\mathbb{O} &:= \{\} \\
spec^3_n &:= \Box(\llbracket c \leftarrowtail in \rrbracket \vee (\bigvee_{\forall i \in \mathbb{N}, i < n} \llbracket c_i \leftarrowtail c \rrbracket) \wedge \bigwedge_{\forall i \in \mathbb{N}, i < n} (p(c_i) \iff p(c_{i+1})))
\end{aligned}$$

Specification 4 ($n > 1$)

$$
\begin{aligned}
\mathbb{C} :=& \{c, c_0, ..., c_i\} \\
\mathbb{I} :=& \{in\} \\
\mathbb{O} :=& \{\} \\
spec^4_n :=& \Box(\llbracket c \hookleftarrow in \rrbracket \vee (\bigvee_{\forall i \in \mathbb{N}, i<n} \llbracket c_i \hookleftarrow f(c) \rrbracket \vee \llbracket c_i \hookleftarrow g(c) \rrbracket) \wedge \bigwedge_{\forall i \in \mathbb{N}, i<n} (p(c_i) \iff p(c_{i+1})))
\end{aligned}
$$

Specification 5

$$
\begin{aligned}
\mathbb{C} :=& \{a\} \\
\mathbb{I} :=& \{in\} \\
\mathbb{O} :=& \{\} \\
spec^5 :=& (\Box(\llbracket a \hookleftarrow in \rrbracket \vee \llbracket a \hookleftarrow a \rrbracket)) \wedge (\Diamond(p(i)) \to \Diamond(\neg p(a)))
\end{aligned}
$$

Note, that this specification is unrealizable. Therefore, only the benchmarks for parallel and classical spurious checking are performed on it.

5.2 Results

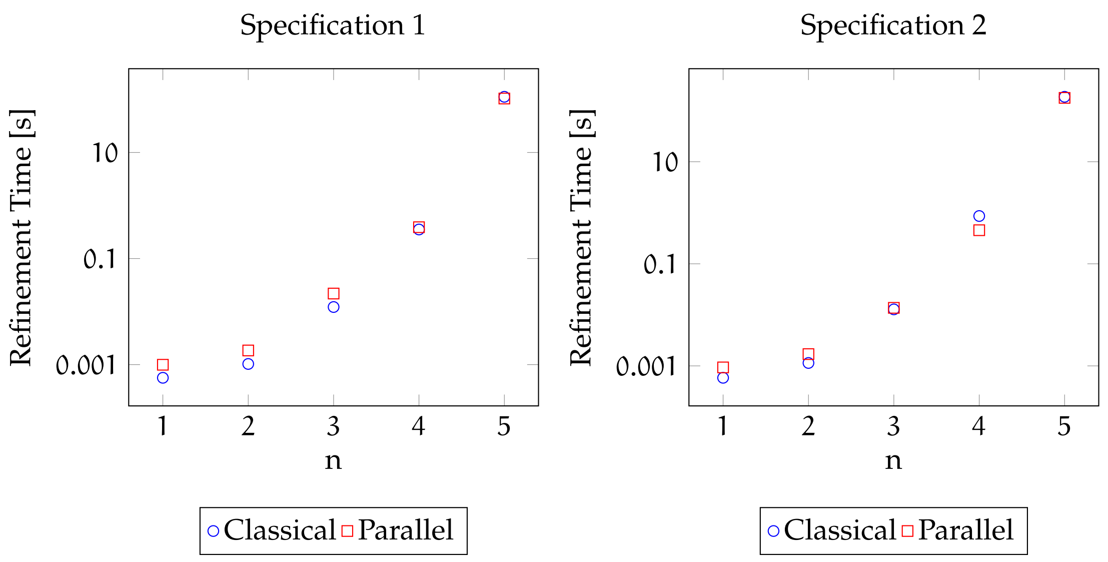
→Sec. 5.1, p. 32

In this section, the results of benchmarking the specifications from Sect. 5.1 are evaluated and discussed.

5.2.1 Parallelization

→Alg. 3, p. 21

The approach of parallelizing the check-spuriousness algorithm (Alg. 3) is only compared against the non-parallelized version. This is because the other approaches aim at generating customized assumptions, rather than on speeding up the check for spuriousness.



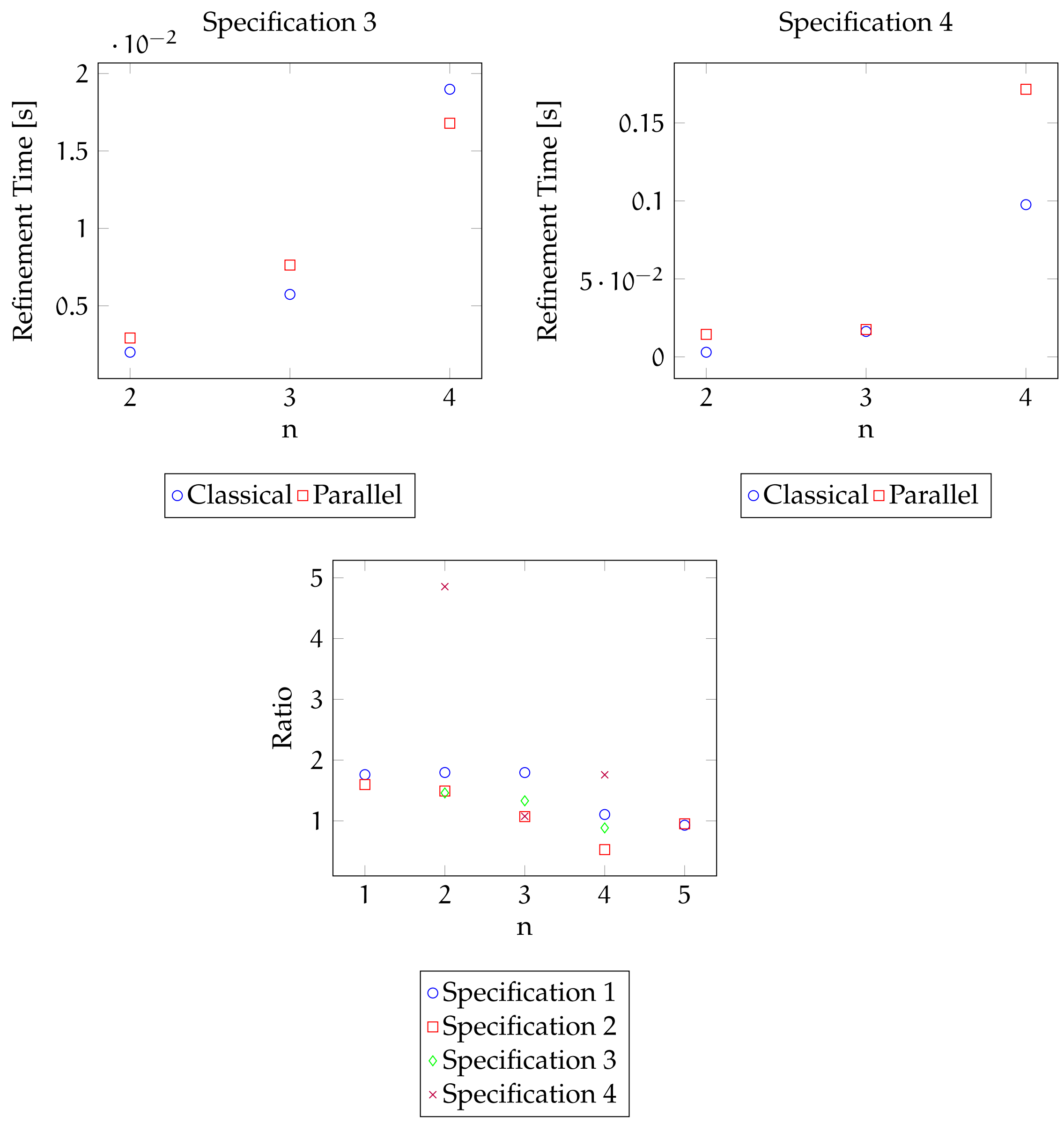


Figure 5.3: Refinement Time Ratio (Parallel / Classical)

For specifications where spuriousness can be detected early in the algorithm, the overhead generated by parallelization overweights the benefits. For specification 3 and specification 4 parallelization performed worse for every variant except $spec^3_4$, where parallelization ran 13% faster. For larger assumptions, the parallelization approach slightly outperformed the classical approach. For specification 5, which is the only unrealizable assumption in the test set, a complete exploration of all possible counter-

Classical	Parallel
35.2s	16.3s

Table 5.4: Specification 5: Check-Spuriousness Time

strategy paths is necessary to establish its spuriousness. In this case, the algorithm massively benefits from parallelization, as the runtime of the parallelized algorithm was only 46.3% of the classical version, as shown in Tbl. 5.4.

An overview of the comparative performance between the two approaches is given in Fig. 5.3.

5.2.2 Overall Runtime and Memory Usage

In this section, the overall wallclock time and memory usage for synthesis is compared between the implementations of Alg. 2, and its variants described in Sect. 3.3 and Sect. 3.4.

→ Alg. 2, p. 19
→ Sec. 3.3, p. 22
→ Sec. 3.4, p. 24

On specification 1 and 3 the process reached timeout after 1800 seconds for every approach for $n = 4$ and $n = 5$, also for specification 3 all benchmarkes timed out for $n = 5$, for specification 4 all benchmarks timed out for $n = 4$ and $n = 5$.

For specifications 1, 3, and 4, synthesis with strengthened initial assumptions ran fastest. For specification 2, this method was slightly slower than the classical approach. However, for this specification, there are no stronger initial assumptions given by the chosen procedure since every update contains functions. Synthesis with generalized assumptions performed roughly equivalent or worse to the classical approach concerning synthesis time, except for spec_2^1 and spec_3^1, where it was slightly faster. The largest difference between the classical and an alternative approach was for spec_2^3, spec_2^4, and spec_3^4 where stronger initial assumptions produced results in only 62% of the time needed by the classical strategy.

The refinement time though made up only a small portion of synthesis time, but as can be seen from the results, the strategy for assumption generation has a huge impact on the synthesis process.

For memory usage, the results were much more inconsistent. No strategy performed significantly better than another over a large portion of the test set. The overall memory usage between the approaches was generally in the same range, though there were some outliers.

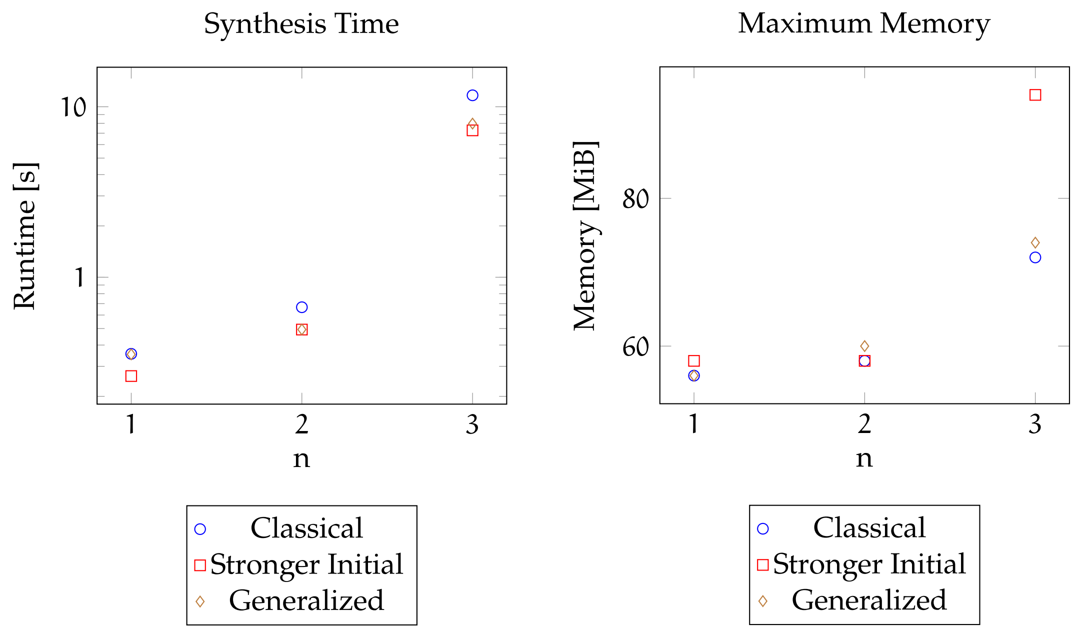


Figure 5.5: Specification 1

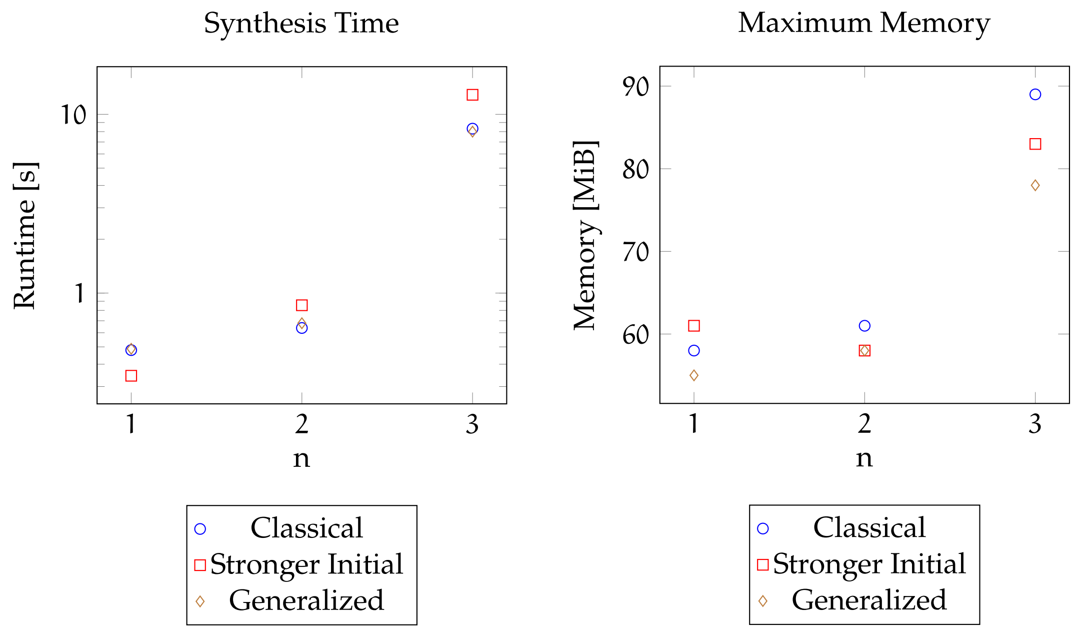


Figure 5.6: Specification 2

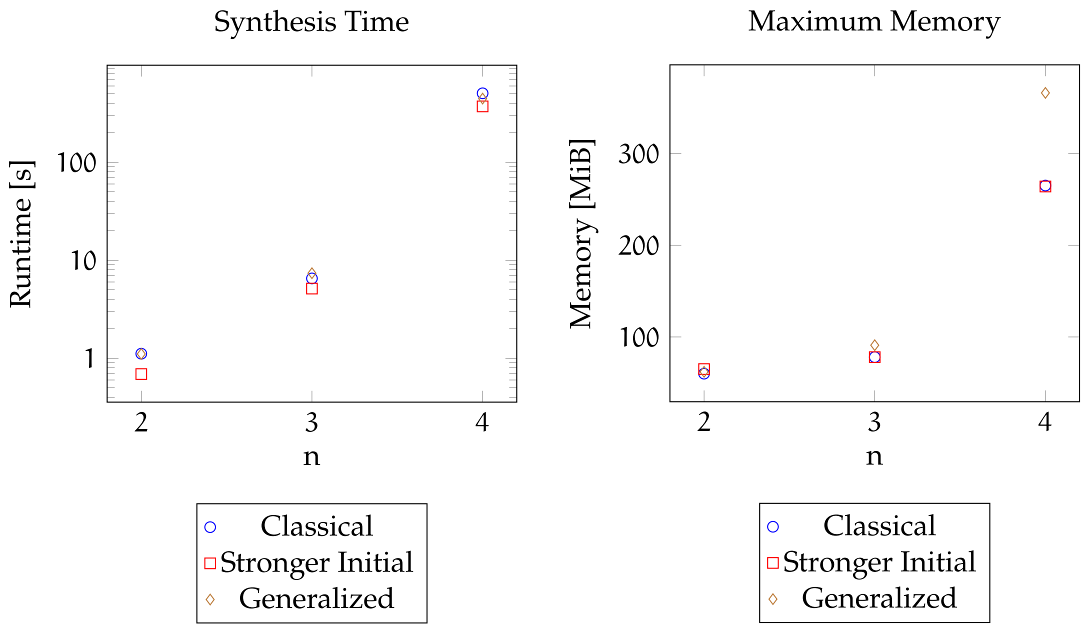


Figure 5.7: Specification 3

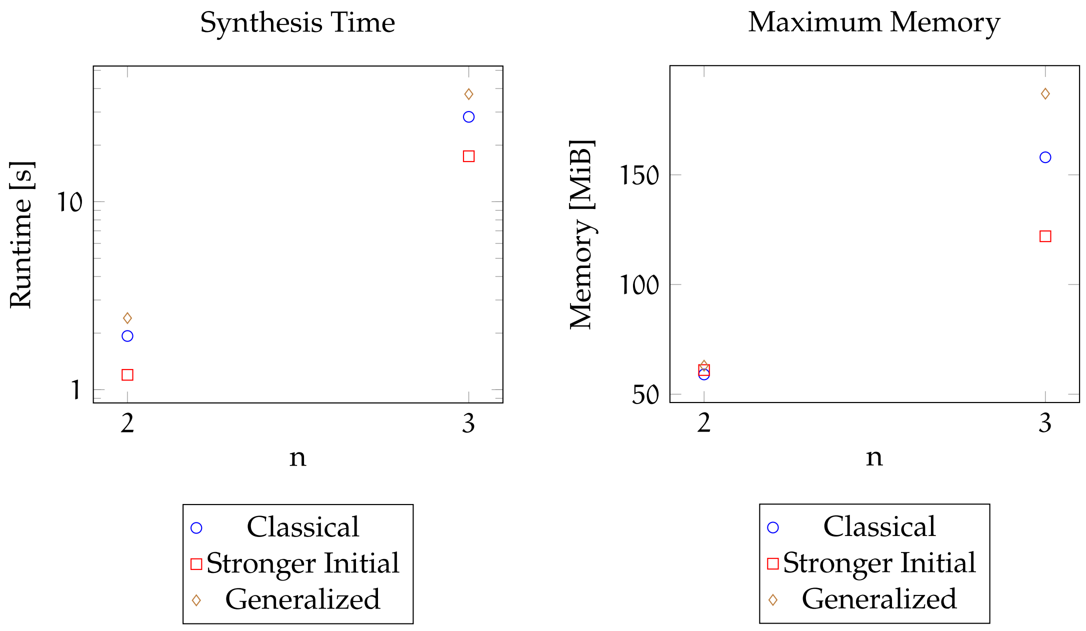


Figure 5.8: Specification 4

5.2.3 Refinement count

As shown in Tbl. 5.9, generalized assumptions did not decrease the number of refinement iterations needed. However, even though it was not the case for all specifications in the test set, stronger initial assumptions did significantly decrease the number of necessary refinements.

Note, that for $spec^1$ and $spec^2$ there was no difference in refinement count for each size-variant of a specification.

Specification	Classical	Stronger Initial	Generalized
$spec^1_n$	1	0	1
$spec^2_n$	1	1	1
$spec^3_2$	3	1	3
$spec^3_3$	5	2	5
$spec^3_4$	7	3	7
$spec^4_2$	3	1	3
$spec^4_3$	5	2	5
$spec^4_4$	7	3	7

Table 5.9: Refinement count by specification and algorithm

Chapter 6

Related Work

The synthesis problem was introduced in [12] by Alonzo Church and is therefore also known as Church's Problem. It asks for the construction of a reactive system, given a specification describing the desired behavior. Since its introduction, there have been many attempts to tackle the problem. Büchi and Landweber established the decidability of the problem for specifications in monadic second-order logic of one successor (S1S) and provided a solving algorithm [6]. However, the computational complexity of the translation from formulas S1S to Büchi automata is nonelementary [21].

By using Linear Temporal Logic (LTL) [18] instead of S1S, the complexity of the problem can be reduced to double exponential [9]. Nevertheless, for many practical applications, this is still computationally expensive. Other attempts to find a more efficient solution for the problem focus on specific fragments of LTL. An example of this is Generalized Reactivity (1) (GR(1)) [16], which further reduces the complexity to quadratic. These fragments, however, come at the cost of a less expressive specification language. Although it is possible to synthesize some specifications which can not be explicitly expressed in GR(1), they then require an expensive pre-processing step.

Another approach is bounded synthesis [20]. While the possible size of a system generated by a specification is high, many can be implemented with a comparatively small number of states. Bounded synthesis then reduces the synthesis problem to only find solutions up to a given number of states. This approach is suitable to construct minimal and therefore simply structured solutions. Furthermore, it is feasible to use it as a semi-decision procedure for the unbounded synthesis problem of distributed systems, which is in general undecidable.

For some systems that base on several reusable components, it is the case that synthesis leads to increasingly larger systems when compared to manual implementations [4]. Parameterized synthesis [14] can be used to counteract this problem for synthesizing the AMBA AHB protocol [4].

All these processes have in common that they perform poorly when handling data since it has to be encoded in the states of the system. A possible approach to efficient and scalable reactive synthesis is based on Temporal Stream Logic (TSL) [11]. TSL has similar syntax and semantics to LTL, however, it introduces useful abstractions on data handling. The synthesis technique is then based on the idea of synthesizing a controller for the data flow, without concretely specifying any data processing parts of the system. On the downside, the TSL synthesis problem is in general undecidable. A TSL specification is synthesized by under-approximating it in LTL with no environmental assumptions at all and then obtaining a simple solution by bounded synthesis. If the approximation is unrealizable due to increased freedom of the environment, counterexample-guided abstraction refinement (CEGAR) [7] is used to generate additional assumptions. Further research on synthesizing TSL specification focuses on extending TSL with first-order theories [10], which identified semi-decidable fragments of TSL for its satisfiability problem and introduced a corresponding algorithm. Moreover, there is a synthesis algorithm for TSL synthesis under theories [15].

TSL synthesis has already been used to successfully synthesize complex systems such as a music player app [11] as well as a small game on a handheld device [13]. Both tasks require an intense amount of data handling, which would not have been possible to synthesize using other approaches.

Chapter 7

Conclusion

In this thesis, we optimized the TSL check-spuriousness algorithm Alg. 1 for implementation. By nesting the loops in a specific order, it is possible to reduce the number of iterations over computations Furthermore, by taking advantage of partial symmetry in the algorithm and the generated assumption, the iterations can be cut down further. Another step to reducing iterations is possible by partitioning the predicate terms by their predicate, and only iterating over the respective set. Additionally, by exploring a counter-strategy only once for a given strategy computational effort can be reduced. Besides, by fixing the order of timestep pairs, it can be ensured that the number of necessary refinements is minimal.

By taking advantage of the fact that iterations over computations are independent of each other we introduced a parallelized version of the optimized algorithm. Moreover, we proposed a possible way of strengthening initial assumptions by adding assumptions that ensure equivalent predicate evaluation over values that are not the results of functions. Furthermore, we introduced an alternative, more generalized way of generating additional assumptions that does not rely on complete paths implying the necessary behavior.

Finally, we benchmarked the suggested variants of the algorithms on a set of test specifications. Its results suggest, that the strengthening of initial assumptions is a promising way to reduce synthesis time for TSL synthesis. Moreover, parallelization improves the runtime of spuriousness checks at least for large or unrealizable specifications.

Anyway, the results also showed that the actual time needed for checking the spuriousness of counter-strategies is minuscule compared to the overall synthesis time. Consequently, the step of refining specifications has little room for optimizations, however, TSL synthesis would massively benefit from further improvements in bounded synthesis.

Further Work For further work, it would be possible to make at least some assumptions on the evaluation of predicates and functions. This could easily be done by adjusting the evaluation function that is used for checking the spuriousness of a path. Consequently, this would make it possible to synthesize specifications that would be unrealizable under the theory of uninterpreted functions and predicates. Furthermore, it could be investigated if this opens up new ways of strengthening the initial assumptions or generating generalized assumptions. Additionally, there may be other ways of improving initial assumptions or generating generalized assumptions that have not been considered in this thesis.

Bibliography

[1] *AIGER Format Description*. http://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf. Accessed: 2022-09-05.

[2] *ANTLR Parser Generator*. https://www.antlr.org/. Accessed: 2022-09-05.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 026202649X. URL: http://www.amazon.com/Principles-Model-Checking-Christel-Baier/dp/026202649X%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D026202649X.

[4] Roderick Bloem, Swen Jacobs, and Ayrat Khalimov. "Parameterized Synthesis Case Study: AMBA AHB". In: *Electronic Proceedings in Theoretical Computer Science* 157 (July 2014), pp. 68–83. ISSN: 2075-2180. DOI: 10.4204/eptcs.157.9. URL: http://dx.doi.org/10.4204/EPTCS.157.9.

[5] *BoSy on Github*. https://github.com/reactive-systems/bosy. Accessed: 2022-09-05.

[6] J. Richard Buchi and Lawrence H. Landweber. "Solving Sequential Conditions by Finite-State Strategies". In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311. ISSN: 00029947. URL: http://www.jstor.org/stable/1994916.

[7] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". In: *J. ACM* 50.5 (Sept. 2003), pp. 752–794. ISSN: 0004-5411. DOI: 10.1145/876638.876643. URL: https://doi.org/10.1145/876638.876643.

[8] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. "BoSy: An Experimentation Framework for Bounded Synthesis". In: *Proceedings of CAV*. Vol. 10427. LNCS. Springer, 2017, pp. 325–332. DOI: 10.1007/978-3-319-63390-9_17.

[9] Bernd Finkbeiner. “Synthesis of Reactive Systems”. In: *Dependable Software Systems Engineering*. Ed. by Javier Esparza, Orna Grumberg, and Salomon Sickert. Vol. 45. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2016, pp. 72–98. ISBN: 978-1-61499-626-2.

[10] Bernd Finkbeiner, Philippe Heim, and Noemi Passing. “Temporal Stream Logic modulo Theories”. In: *Foundations of Software Science and Computation Structures*. Ed. by Patricia Bouyer and Lutz Schröder. Cham: Springer International Publishing, 2022, pp. 325–346. ISBN: 978-3-030-99253-8.

[11] Bernd Finkbeiner et al. “Temporal Stream Logic: Synthesis Beyond the Bools”. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 609–629. ISBN: 978-3-030-25540-4.

[12] Joyce Friedman. “Church Alonzo. Application of Recursive Arithmetic to the Problem of Circuit Synthesis Summaries of Talks Presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd Edn., Communications Research Division, Institute for Defense Analyses, Princeton, N. J., 1960, Pp. 3?50. 3a-45a”. In: *Journal of Symbolic Logic* 28.4 (1963), pp. 289–290. DOI: `10.2307/2271310`.

[13] Gideon Geier et al. “Syntroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications”. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. 2019, pp. 138–146. DOI: `10.23919/FMCAD.2019.8894261`.

[14] Swen Jacobs and Roderick Bloem. “Parameterized Synthesis”. In: *Logical Methods in Computer Science* Volume 10, Issue 1 (Feb. 2014). DOI: `10.2168/LMCS-10(1:12)2014`. URL: `https://lmcs.episciences.org/736`.

[15] Benedikt Maderbacher and Roderick Bloem. “Reactive Synthesis Modulo Theories Using Abstraction Refinement”. In: *arXiv e-prints*, arXiv:2108.00090 (July 2021), arXiv:2108.00090. arXiv: `2108.00090 [cs.LO]`.

[16] Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. “Synthesis of Reactive(1) Designs”. In: *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Vol. 3855. Lecture Notes in Computer Science. Springer, 2006, pp. 364–380. DOI: `10.1007/11609773\_24`. URL: `https://doi.org/10.1007/11609773%5C_24`.

[17] A. Pnueli and R. Rosner. “On the Synthesis of a Reactive Module”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 179–190. ISBN: 0897912942. DOI: `10.1145/75277.75293`. URL: `https://doi.org/10.1145/75277.75293`.

[18] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[19] *runsolver*. `http://www.cril.univ-artois.fr/~roussel/rech.php`. Accessed: 2022-09-05.

[20] Sven Schewe and Bernd Finkbeiner. "Bounded Synthesis". In: *Proc. ATVA*. Springer-Verlag, 2007, pp. 474–488.

[21] Larry J. Stockmeyer. "The Complexity of Decision Problems in Automata Theory and Logic". In: 1974.

[22] *Swift Programming Language*. `https://swift.org`. Accessed: 2022-09-05.

[23] Wolfgang Thomas. "Solution of Church's Problem: A tutorial". In: *New Perspectives on Games and Interaction* 4 (Jan. 2008).