Masaryk University
Faculty of Informatics

Parallel parameter synthesis from hybrid logic $HUCTL_p$ formulas

Master's Thesis

Samuel Pastva

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Parallel parameter synthesis from hybrid logic $HUCTL_p$ formulas

Master's Thesis

Samuel Pastva

Brno, Spring 2017

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Samuel Pastva

Advisor: prof. RNDr. Luboš Brim, CSc.

Acknowledgement

This work was supported by the Masaryk University as part of the MUNI/C/1017/2015 project: "Automatizovaná syntéza parametrov z temporálnych špecifikácií".

Personally, I would like to thank all members of the Systems Biology Laboratory for their insights, support and work we all invested in Pithya, HUCTL$_P$ and other related tools and algorithms. Namely, I would like to thank Luboš Brim, David Šafránek, Nikola Beneš, Martin Demko and Matej Hajnal for their time and dedication.

I also have to thank my family and friends, who supported me throughout the last five years of my studies and without whom I would not be the person I am today.

Abstract

In order to study complex phenomena arising in nature, one often uses various modelling frameworks which employ parameters to describe the uncertainty of the physical world. Related to these modelling techniques is the problem of parameter synthesis, that is to compute for a desired property the parameter valuations under which the property is satisfied in the model.

In this work, we present a novel parallel algorithm for solving the parameter synthesis problem for properties defined using the hybrid computation tree logic with past (HUCTL_{P}). The algorithm is based on a semi-symbolic approach, where the state space is represented explicitly while the parameter space is represented symbolically using formulas of a first order logic. The decisions about the parameter space are then delegated to an appropriate solver. The algorithm is implemented as a part of the parameter synthesis tool Pithya.

We show the scalability and applicability of our approach on a set of biochemical models based on ordinary differential equations.

Keywords

parameter synthesis, model checking, temporal logic, $HUCTL_P$, dynamical system, ordinary differential equation, parallel algorithm

Contents

1 Introduction

Before we dive into the technical details of this work, let us briefly describe the context and motivation behind it.

Dynamical systems

As countless examples from biology, physics and economy suggest, naturally occurring phenomena are often extremely hard, even impossible, to study *computationally*. This is often due to a huge amount of information involved. In order to study such phenomena, science often resorts to using models, which omit unnecessary details of the physical reality and focus only on the minimal representation needed to encode the interesting dynamics.

One of such modelling techniques are *dynamical systems* [1, 2], which employ the framework of *ordinary differential equations* to describe the dynamics of physical phenomena. In order to study such models exactly, one usually relies on techniques from the field of mathematical analysis. However, these are often intractable due to the sheer complexity of the differential equations involved. In such cases, one often needs to resort to simulation or various types of visualisations.

Parameter synthesis

When dealing with models, one often has to consider a significant amount of *uncertainty*, usually represented using parameters which influence the systems dynamics. The uncertainty can arise under different circumstances, due to the nature of the system (e.g. properties which are hard to measure experimentally) or due to the design of the system (e.g. initial conditions which are controlled by the scientist). However, no matter what is the reason for the uncertainty, it always complicates the study of the model. Even simulation and visualisation can be intractable when high amount of uncertainty is involved.

In such systems, we talk about a *parameter synthesis* problem. That is, given a desired property, determine all parameter valuations under which the system satisfies the property. For example, given a model of a cell with an ambient temperature as a parameter, determine under which circumstances is the cell able to reproduce.

Temporal logic

Before solving the parameter synthesis problem, one needs to first provide a formal description of the desired property. Creating such specification can be often an error prone task. To make this process more intuitive, one usually specifies the properties using some suitable *temporal logic*.

Commonly used temporal logics include linear time logic (LTL) and computation tree logic (CTL), where LTL uses the notion of linear time, whereas CTL uses branching time. Various extensions and modifications of these temporal logics exist [3, 4, 5]. In this work, we introduce and employ a hybrid extension of the UCTL logic [4], the hybrid computation tree logic with past (HUCTL_P) [6], as our framework for specifying model properties.

Model checking

To solve the parameter synthesis problem for properties specified using HUCTL_P logic, we introduce an algorithm based on the well known *model checking* technique [7]. Model checking is well studied exhaustive method often used for software and hardware verification, but which can be also applied to the parameter synthesis problem [8, 9, 10, 11].

To cope with the parameter uncertainty, we use the coloured approach [12, 13], which enables us to consider multiple parameter valuations with equivalent local behaviour at the same time, thus reducing the average computation time.

Overview

In the Preliminaries (Chapter 2), we formally define HUCTL_P and its semantics over direction transition systems. In Chapter 3, we describe the parameter synthesis procedure, which is based on a fixed point assumption function. Then, in Chapter 4, we discuss the implementation of the algorithm, provided as part of the Pithya tool. Finally, in Chapter 5, we present an evaluation of Pithya in terms of performance and a case study showing the applicability of our method.

2 Preliminaries

In this chapter, we introduce the basic notions used throughout the thesis together with their more intuitive explanations.

2.1 Direction transition system

Definition 1. *A* direction transition system *(DTS) is a tuple* (S, Dir, T, AP, L)*, where:*

- S *is a non-empty set of* states*;*
- Dir *is a finite non-empty set of* directions*;*
- $T \subseteq S \times Dir \times S$ *is the* transition relation *satisfying the following conditions:*
  - T *is* total*, that is, for each* s *there is some* s' *and* d *such that* $(s, d, s') \in T$*;*
  - T *is* past-total*, that is, for each* s *there is some* s' *and* d *such that* $(s', d, s) \in T$*;*
  - *for each* $s \neq s'$ *there is at most one* d *such that* $(s, d, s') \in T$*;*
  - *for each* s *there is either no* d *such that* $(s, d, s) \in T$ *or for all* $d \in Dir : (s, d, s) \in T$*;*
- AP *is a set of atomic propositions;*
- $L : S \to 2^{AP}$ *is a labelling function that associates a subset of* AP *to each state.*

A DTS is an extension of the standard notion of *transition systems* which adds to each transition also a direction label. Furthermore, aside from the standard requirement of totality, transition relation also has to be past-total, which allows us to reason about both future and past runs of such system.

Note that the self-loops in a DTS are not distinguishable from the direction standpoint, meaning there is no $loop$ direction. Instead, the self-loop is labelled with all directions, which intuitively means that

the system can move in any direction, but the perturbation is not strong enough to modify the overall state of the system.

We use the notation $\mathcal{D}(s,t)$ to denote the set of all directions between two states: $\mathcal{D}(s,t) = \{d \in Dir \mid (s,d,t) \in T\}$ (Note that this is either a singleton set, or the whole Dir). We will also use $s \xrightarrow{d} s'$ to denote $(s,d,s') \in T$ and $s \to s'$ if there exists $d \in Dir$ such that $s \xrightarrow{d} s'$.

2.2 Time flow and runs in DTS

In this work, we consider two possible semantics of a DTS: past and future. Collectively, we refer to these as time flow. We use a ${}^{\triangleright}$ prefix to denote a context where we consider the future semantics and ${}^{\triangleleft}$ to denote a context where the past semantics is considered. We also assume that the time flow can be negated, meaning ${}^{\neg\triangleright} \equiv {}^{\triangleleft}$ and ${}^{\neg\triangleleft} \equiv {}^{\triangleright}$.

Let $M = (S, Dir, T, AP, L)$ be a DTS. In accordance with the above specified notation, we define the *future transition relation* as ${}^{\triangleright}T = T$ and *past transition relation* as ${}^{\triangleleft}T = \{(s',d,s) \mid (s,d,s') \in {}^{\triangleright}T\}$. Note that since T is both total and past-total, both ${}^{\triangleright}T$ and ${}^{\triangleleft}T$ are total.

Let t be one of ${}^{\triangleright}$ and ${}^{\triangleleft}$. Then:

- A *run* ${}^{t}\pi$ is an infinite sequence $s_0, d_0, s_1, d_1, s_2, \ldots$ such that $(s_i, d_i, s_{i+1}) \in {}^{t}T$ for all i. If the time flow of the run is clear from context, we can omit the ^{t} prefix;
- ${}^{t}\pi_S(i)$ denotes the i-th state s_i and ${}^{t}\pi_{Dir}(i)$ denotes the i-th direction d_i of the run ${}^{t}\pi$;
- ${}^{t}\Pi_M$ denotes the set of all t-runs of the DTS M;
- Function ${}^{t}runs_M : S \to \mathcal{P}({}^{t}\Pi_M)$ computes all runs originating in the given state: ${}^{t}runs_M(s) = \{\pi \in {}^{t}\Pi_M \mid \pi_S(0) = s\}$;
- Function ${}^{t}succ_M : S \to \mathcal{P}(Dir \times S)$ computes the successors of the given state (with respect to time flow t): ${}^{t}succ_M(s) = \{(d,s') \in Dir \times S \mid (s,d,s') \in {}^{t}T\}$;
- Similarly, function ${}^{t}pred_M : S \to \mathcal{P}(Dir \times S)$ computes the predecessors of the given state: ${}^{t}pred_M(s) = {}^{\neg t}succ_M(s)$;

Whenever the DTS M is clear from context, we can omit the subscript M.

Compared to the standard transition systems, we incorporate the notion of direction into a run by representing it as an alternating sequence of states and directions. Notice that each run always starts in a state. Everything is then parametrised by the notion of time flows.

2.3 Direction formulae

To reason about a direction of a specific transition, we define the language of direction formulae.

Definition 2. *Let Dir be a set of directions. The language of direction formulae is then defined as follows:*

$$\chi ::= \mathit{true} \mid d \mid \neg\chi \mid \chi \wedge \chi$$

For a direction $\hat{d}$ is the satisfaction relation $\models$ then defined as follows:

$$\begin{aligned}
&\hat{d} \models \mathit{true} \\
&\hat{d} \models d && \iff \hat{d} = d \\
&\hat{d} \models \neg\chi && \iff \hat{d} \not\models \chi \\
&\hat{d} \models \chi_1 \wedge \chi_2 && \iff \hat{d} \models \chi_1 \text{ and } \hat{d} \models \chi_2
\end{aligned}$$

Intuitively, a direction formula is just a standard Boolean formula with directions as propositions.

2.4 Hybrid computation tree logic with past

To reason about a DTS, we define the following HUCTL$_{\mathrm{P}}$ logic.

2.4.1 Syntax

Definition 3. *Let p be an atomic proposition from the AP set, d a direction formula over Dir, t one of the time flows ($^{\triangleright}$ or $^{\triangleleft}$), and x a state variable. The language of HUCTL$_{\mathrm{P}}$ formulae is then defined as follows:*

$$\varphi ::= true \mid p \mid x \mid \neg\varphi \mid \downarrow x : \varphi \mid @x : \varphi \mid \exists x : \varphi \mid \varphi \wedge \varphi \mid {}^{t}\mathbf{E}\,\psi \mid {}^{t}\mathbf{A}\,\psi$$
$$\psi ::= \mathbf{X}_{\chi}\varphi \mid \varphi_{\chi}\,\mathbf{U}\,\varphi \mid \varphi_{\chi}\,\mathbf{U}_{\chi}\,\varphi \mid \varphi_{\chi}\,\mathbf{W}\,\varphi \mid \varphi_{\chi}\,\mathbf{W}_{\chi}\,\varphi$$

We call all φ formulae *state formulae* and all ψ formulae *path formulae*. We write $cl(\varphi)$ to denote the set of all sub-formulae of φ and $sub(\varphi)$ to denote the set of all direct sub-formulae.

The temporal operators follow the common naming scheme based on CTL (**X** - next, **U** - until, **W** - weak until), but we also introduce new operators, namely $@x : \varphi$, pronounced "at", $\downarrow x : \varphi$, pronounced "bind" and $\exists x : \varphi$, pronounced "exists".

Note that in situations where the aspect of time flow is irrelevant (i.e. when the statement holds for both past and future), we can omit the time flow prefix.

2.4.2 Semantics

In order to describe semantics of the HUCTL$_{\mathrm{P}}$ as a whole, we define the semantics of the state and path formulae separately. The model of a state formula over DTS M is a state s while the model of a path formula is a run π. Furthermore, each model is extended with a partial function $h : Var \to S$, which represents the valuation of the state variables.

We write h_0 to denote an empty valuation and $h[x \mapsto s]$ to denote a valuation which maps variable x to a state s but is otherwise defined as valuation h, formally:

$$h[x \mapsto s](x') = \begin{cases} s & x' = x \\ h(x') & otherwise \end{cases}$$

The satisfaction relation for states of a DTS M with respect to a HUCTL$_{\mathrm{P}}$ state formula is defined in Figure 2.1

Compared to CTL, we can see that the path operators **A** and **E** are extended with the notion of time flow. Also, the semantics of $\exists x : \phi$ follow directly from first-order logic.

The most interesting are the remaining operators: *bind* and *at*. Intuitively, *bind* provides a more "specialised" alternative to *exists*, while *at*

$$
\begin{aligned}
&(M,h,s) \models \mathit{true} \\
&(M,h,s) \models p && \iff p \in L(s) \\
&(M,h,s) \models x && \iff h(x) = s \\
&(M,h,s) \models \neg\varphi && \iff (M,h,s) \not\models \varphi \\
&(M,h,s) \models \downarrow x : \varphi && \iff (M,h[x \mapsto s],s) \models \varphi \\
&(M,h,s) \models @x : \varphi && \iff (M,h,h(x)) \models \varphi \\
&(M,h,s) \models \exists x : \varphi && \iff \exists s' \in S : (M,h[x \mapsto s'],s) \models \varphi \\
&(M,h,s) \models \varphi_1 \wedge \varphi_2 && \iff (M,h,s) \models \varphi_1 \text{ and } (M,h,s) \models \varphi_2 \\
&(M,h,s) \models {}^{t}\mathbf{E}\,\psi && \iff \exists \pi \in {}^{t}\mathit{runs} : (M,h,\pi) \models \psi \\
&(M,h,s) \models {}^{t}\mathbf{A}\,\psi && \iff \forall \pi \in {}^{t}\mathit{runs} : (M,h,\pi) \models \psi
\end{aligned}
$$

Figure 2.1: Semantics of HUCTL$_{\mathrm{P}}$ state formulae. Let $M = (S, Dir, T, AP, L)$ be a DTS and $h : Var \to S$ a valuation of state variables.

allows effect similar to memory access, allowing us to check property in a previously saved state (in some parent formula).

The satisfaction relation for runs of M with respect to a HUCTL$_{\mathrm{P}}$ path formula is defined in Figure 2.2.

As we can see, the semantics of path formulae are very similar to the standard CTL. However, the operators are extended with the notion of directions, effectively restricting the set of runs considered when deciding the satisfaction of the property.

2.4.3 Other operators

Apart from the basic set of operators defined by the HUCTL$_{\mathrm{P}}$ syntax, we will also use the following abbreviations to extend the logic with other common operators.

First, we define directed extensions of the standard CTL **F** and **G** operators:

$$
\begin{aligned}
(M,h,\pi) \models \mathbf{X}_{\chi}\varphi \quad &\iff \pi_{Dir}(0) \models \chi \text{ and } (M,h,\pi_S(1)) \models \varphi \\
(M,h,\pi) \models \varphi_1 {}_{\chi}\mathbf{U}\, \varphi_2 \quad &\iff \exists i : \pi_S(i) \models \varphi_2 \text{ and} \\
&\quad\ \forall j < i : \pi_S(j) \models \varphi_1 \wedge \pi_{Dir}(j) \models \chi \\
(M,h,\pi) \models \varphi_1 {}_{\chi}\mathbf{U}_{\xi}\, \varphi_2 \quad &\iff \exists i > 0 : \pi_S(i) \models \varphi_2 \text{ and} \\
&\quad\ \pi_S(i-1) \models \varphi_1 \wedge \pi_{Dir}(i-1) \models \xi \text{ and} \\
&\quad\ \forall j < i-1 : \pi_S(i) \models \varphi_1 \wedge \pi_{Dir}(j) \models \chi \\
(M,h,\pi) \models \varphi_1 {}_{\chi}\mathbf{W}\, \varphi_2 \quad &\iff (M,h,\pi) \models \varphi_1 {}_{\chi}\mathbf{U}\, \varphi_2 \text{ or} \\
&\quad\ \forall i : \pi_S(i) \models \varphi_1 \wedge \pi_{Dir}(i) \models \chi \\
(M,h,\pi) \models \varphi_1 {}_{\chi}\mathbf{W}_{\xi}\, \varphi_2 \quad &\iff (M,h,\pi) \models \varphi_1 {}_{\chi}\mathbf{U}_{\xi}\, \varphi_2 \text{ or} \\
&\quad\ \forall i : \pi_S(i) \models \varphi_1 \wedge \pi_{Dir}(i) \models \chi
\end{aligned}
$$

Figure 2.2: Semantics of HUCTL$_{\mathrm{P}}$ path formulae. Let $M = (S, Dir, T, AP, L)$ be a DTS and $h : Var \to S$ a valuation of state variables.

$$
\begin{aligned}
{}_{\chi}\mathbf{F}\, \varphi &\equiv true {}_{\chi}\mathbf{U}\, \varphi \\
{}_{\chi}\mathbf{G}\, \varphi &\equiv \varphi {}_{\chi}\mathbf{W}\, false
\end{aligned}
$$

Further discussion of these operators and their relationship with their CTL counterparts is provided in Subsection 2.4.5.

Second, we define other operators commonly used in the first order logic:

$$
\begin{aligned}
\forall x : \varphi &\equiv \neg\exists x : \neg\varphi \\
\exists x \in \varphi_1 : \varphi_2 &\equiv \exists x : ((@x : \varphi_1) \wedge \varphi_2) \\
\forall x \in \varphi_1 : \varphi_2 &\equiv \forall x : ((@x : \varphi_1) \implies \varphi_2)
\end{aligned}
$$

The first operator is the standard first-order universal quantifier, whereas the second and third operator are based on the standard

first-order quantifiers, but they are extended with the ability to restrict the space of the variable x to a validity region of a specific formula. Such modification can be very useful when optimizing the execution time of the formula, since validity for a significant amount of states can be decided without actually considering formula φ_2.

2.4.4 Relationship with CTL

Since HUCTL$_P$ is an extension of CTL, its operators can be used to define the standard CTL. To do so, we can use the following equivalences:

$$\begin{aligned}
\mathbf{E}\,\mathbf{X}\,\varphi &\equiv {}^{\triangleright}\,\mathbf{E}\,\mathbf{X}_{\mathit{true}}\varphi & \mathbf{A}\,\mathbf{X}\,\varphi &\equiv {}^{\triangleright}\,\mathbf{A}\,\mathbf{X}_{\mathit{true}}\varphi \\
\mathbf{E}\,\mathbf{F}\,\varphi &\equiv {}^{\triangleright}\,\mathbf{E}_{\mathit{true}}\,\mathbf{F}\,\varphi & \mathbf{A}\,\mathbf{F}\,\varphi &\equiv {}^{\triangleright}\,\mathbf{A}_{\mathit{true}}\,\mathbf{F}\,\varphi \\
\mathbf{E}\,\mathbf{G}\,\varphi &\equiv {}^{\triangleright}\,\mathbf{E}_{\mathit{true}}\,\mathbf{G}\,\varphi & \mathbf{A}\,\mathbf{G}\,\varphi &\equiv {}^{\triangleright}\,\mathbf{A}_{\mathit{true}}\,\mathbf{G}\,\varphi \\
\mathbf{E}[\varphi_1\,\mathbf{U}\,\varphi_2] &\equiv {}^{\triangleright}\,\mathbf{E}[\varphi_{1\mathit{true}}\,\mathbf{U}\,\varphi_2] & \mathbf{A}[\varphi_1\,\mathbf{U}\,\varphi_2] &\equiv {}^{\triangleright}\,\mathbf{A}[\varphi_{1\mathit{true}}\,\mathbf{U}\,\varphi_2] \\
\mathbf{E}[\varphi_1\,\mathbf{W}\,\varphi_2] &\equiv {}^{\triangleright}\,\mathbf{E}[\varphi_{1\mathit{true}}\,\mathbf{W}\,\varphi_2] & \mathbf{A}[\varphi_1\,\mathbf{W}\,\varphi_2] &\equiv {}^{\triangleright}\,\mathbf{A}[\varphi_{1\mathit{true}}\,\mathbf{W}\,\varphi_2]
\end{aligned}$$

As one can easily observe, by using *true* as a direction formula, the direction restrictions imposed by HUCTL$_P$ can be ignored and the pure CTL operators are obtained. For a further discussion on the effects of direction restrictions on the CTL semantics, please see section 2.4.5.

2.4.5 Weak operators

When understanding HUCTL$_P$ formulas, one has to keep in mind an important distinction between classic CTL and HUCTL$_P$. In CTL, the following equivalences hold universally:

$$\begin{aligned}
\neg\,\mathbf{A}\,\mathbf{X}\,\neg\varphi &\equiv \mathbf{E}\,\mathbf{X}\,\varphi & \neg\,\mathbf{A}\,\mathbf{G}\,\neg\varphi &\equiv \mathbf{E}\,\mathbf{F}\,\varphi \\
\neg\,\mathbf{E}\,\mathbf{X}\,\neg\varphi &\equiv \mathbf{A}\,\mathbf{X}\,\varphi & \neg\,\mathbf{E}\,\mathbf{G}\,\neg\varphi &\equiv \mathbf{A}\,\mathbf{F}\,\varphi
\end{aligned}$$

However, in HUCTL$_P$, these equivalences are only valid when the direction restriction on the operators are *true*, which reduces them to their classic CTL counterparts.

To understand why these equivalences do not hold, let us explore in detail the case of $\mathbf{A}\,\mathbf{X}$ and $\mathbf{E}\,\mathbf{X}$:

$$
\begin{aligned}
(M,h,s) \models & \neg {}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\neg\varphi \\
& \iff \neg[\forall \pi \in {}^{t}runs : \pi_{Dir}(0) \models \chi \wedge (M,h,\pi_S(1)) \models \neg\varphi] \\
& \iff \exists \pi \in {}^{t}runs : \pi_{Dir}(0) \not\models \chi \vee (M,h,\pi_S(1)) \not\models \neg\varphi \\
& \iff \exists \pi \in {}^{t}runs : \pi_{Dir}(0) \models \chi \implies (M,h,\pi_S(1)) \models \varphi \\
(M,h,s) \models & {}^{t}\mathbf{E}\,\mathbf{X}_{\chi}\varphi \\
& \iff \exists \pi \in {}^{t}runs : \pi_{Dir}(0) \models \chi \wedge (M,h,\pi_S(1)) \models \varphi
\end{aligned}
$$

First, let us observe that if $\chi = true$, both definitions are obviously equivalent. However, as we can see, the original ${}^{t}\mathbf{E}\,\mathbf{X}_{\chi}$ operator intuitively states, that there *exists a t-run where first direction models χ and the next state models φ*, hence there really must exists such run. On the other hand, the expression $\neg {}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\neg\varphi$ translates to a slightly different statement. Intuitively, $\neg {}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\neg\varphi$ states that there *exists a t-run where if the first direction models χ, the next state models φ*. Therefore this formula is satisfied not only when ${}^{t}\mathbf{E}\,\mathbf{X}_{\chi}$ is satisfied, but also when there is a run which does not start with a direction satisfying χ.

Similarly, in case of $\neg {}^{t}\mathbf{E}\,\mathbf{X}_{\chi}\neg\varphi$, we get an intuitive meaning *if the first direction of each t-run models χ, the next state models φ*. Compare this to the meaning of ${}^{t}\mathbf{A}\,\mathbf{X}_{\chi}$, which states that *the first direction of each t-run models χ and the next state models φ*.

As we can see, not only are the semantics of these expressions different, but both semantics can be potentially useful. To leverage this fact, we define a new set of operators:

$$
\begin{aligned}
{}^{t}\mathbf{E}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi &\equiv \neg {}^{t}\mathbf{A}_{\chi}\,\mathbf{G}\,\neg\varphi & \qquad {}^{t}\mathbf{E}\,\widetilde{\mathbf{X}}_{\chi}\varphi &\equiv \neg {}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\neg\varphi \\
{}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi &\equiv \neg {}^{t}\mathbf{E}_{\chi}\,\mathbf{G}\,\neg\varphi & \qquad {}^{t}\mathbf{A}\,\widetilde{\mathbf{X}}_{\chi}\varphi &\equiv \neg {}^{t}\mathbf{E}\,\mathbf{X}_{\chi}\neg\varphi
\end{aligned}
$$

We call these operators *weak*, because in the definition of each of these operators, the original strict direction requirement is replaced with a less restrictive implication. Indeed, similar to the $\mathbf{X}_\chi$ example, when understanding strict and weak ${}_\chi\mathbf{F}$ operator variants, one can follow these intuitive definitions:

- ${}^t\mathbf{E}_\chi\,\mathbf{F}\,\varphi$ - *exists a t-run such that at some point, φ is satisfied and prior to this point χ always holds*;
- ${}^t\mathbf{E}_\chi\,\widetilde{\mathbf{F}}\,\varphi$ - *exists a t-run such that at some point, χ is not satisfied or φ is satisfied (and prior to this point, χ always holds)*;
- ${}^t\mathbf{A}_\chi\,\mathbf{F}\,\varphi$ - *all t-runs contain a point where φ is satisfied and prior to this point, χ always holds*;
- ${}^t\mathbf{A}_\chi\,\widetilde{\mathbf{F}}\,\varphi$ - *all t-runs contain a point where χ is not satisfied or φ is satisfied (and prior to this point, χ always holds)*;

Finally, when using the weak operators, one has to keep in mind that even though the transition relation of each DTS is both total and past-total, it is not necessarily total or past-total with respect to a specific direction formula χ. Hence not only can one encounter runs that satisfy a weak formula due to a transition with does not satisfy χ, one can even encounter complete *direction deadlocks* — that is states, which have no transition satisfying given χ. In such states, all universal weak formulae are automatically satisfied. This fact on itself is not necessarily a disadvantage, however, one has to consider it when interpreting weak formulas.

2.4.6 Other observations

In this final subsection, we define a list of other useful equivalences that allow us to reduce the minimal set of operators needed to describe any HUCTL_P formula:

$$
\begin{aligned}
{}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}_{\xi}\,\varphi_2] &\equiv {}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}(\varphi_1 \wedge {}^{t}\mathbf{E}\,\mathbf{X}_{\xi}\varphi_2)]\\
{}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}_{\xi}\,\varphi_2] &\equiv {}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}(\varphi_1 \wedge {}^{t}\mathbf{A}\,\mathbf{X}_{\xi}\varphi_2)]\\
{}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{W}_{\xi}\,\varphi_2] &\equiv {}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{W}(\varphi_1 \wedge {}^{t}\mathbf{E}\,\mathbf{X}_{\xi}\varphi_2)]\\
{}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{W}_{\xi}\,\varphi_2] &\equiv {}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{W}(\varphi_1 \wedge {}^{t}\mathbf{A}\,\mathbf{X}_{\xi}\varphi_2)]\\
{}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{W}\,\varphi_2] &\equiv \neg{}^{t}\mathbf{E}[\neg\varphi_2\,\mathbf{U}(\neg\varphi_2 \wedge (\neg\varphi_1 \vee {}^{t}\mathbf{E}\,\mathbf{X}_{\neg\chi}\mathit{true}))]\\
{}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{W}\,\varphi_2] &\equiv {}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2] \vee \neg{}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\neg\varphi_1
\end{aligned}
$$

Based on these and previous observations in this section, we can see that one of the minimal operator sets needed to describe all $\mathrm{HUCTL_P}$ formulae is:

$$
\begin{aligned}
{}^{t}\mathbf{E}\,\mathbf{X}_{\chi}\varphi, {}^{t}\mathbf{E}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, {}^{t}\mathbf{E}_{\chi}\,\mathbf{U}\,\varphi,\\
{}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\varphi, {}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, {}^{t}\mathbf{A}_{\chi}\,\mathbf{U}\,\varphi,\\
@x:\varphi, \downarrow x:\varphi, \exists x:\varphi
\end{aligned}
$$

Naturally, this is not the only minimal operator set, however, it is the one we will consider from now on in this work.

2.5 Parametrised direction transition system

In order to reason about systems with parameters, we extend the definition of DTS with the notion of parameters. A PDTS is essentially a family of DTSs that share the same state space, but differ in terms of transition relations. Alternatively, one can view PDTS as a DTS where each transition is labelled not only with direction, but also with a parameter set.

2.5.1 Definition

Parametrised direction transition system is represented by a tuple $\mathcal{K} = (\mathcal{P}, S, \mathit{Dir}, \hat{T}, \mathit{AP}, L)$, where $\mathcal{P}$ is a finite set of parameter valuations and $\hat{T}$ is a parametrised transition relation $\hat{T} \subseteq S \times \mathit{Dir} \times \mathcal{P} \times S$. We use the

notation $\hat{T}_p$ to denote the parametrised transition relation restricted to a specific parameter valuation p, i.e. $\hat{T}_p = \{(s, d, s') \in S \times Dir \times S \mid (s, d, p, s') \in \hat{T}\}$. We then write $\mathcal{K}_p$ to denote a specific parametrisation of the original $\mathcal{K}$—a DTS such that $\mathcal{K}_p = (S, Dir, T_p, AP, L)$.

We write $\mathcal{P}(s, t)$ to denote the set of all parameter valuations for which the transition from s to t is allowed: $\mathcal{P}(s, t) = \{p \in \mathcal{P} \mid \exists d \in Dir : (s, d, p, t) \in \hat{T}\}$. The notion of time flows also naturally extends to PDTSs with ${}^{\triangleright}\hat{T} = \hat{T}$ and ${}^{\triangleleft}\hat{T} = \{(s, d, p, s') \mid (s', d, p, s) \in \hat{T}\}$.

2.5.2 Parameter representation

In this work, we assume that the parameters of the PDTS $\mathcal{K}$ are represented symbolically. We thus assume that we are given a (first-order) theory that is interpreted over the parameter valuations. Every $\mathcal{P}(s, t)$ is then represented using a formula Φ such that $p \models \Phi \iff p \in \mathcal{P}(s, t)$. We call such formula a parameter constrain. We use the notation `ff` and `tt` to denote the contradiction and tautology in terms of parameter formulae and the standard logical operators $\neg, \vee, \wedge$ to combine the parameter formulae. In general, we use upper-case Greek letters $\Phi, \Psi, \ldots$ to denote the parameter formulae and lower-case Greek letters $\varphi, \psi, \ldots$ to denote the HUCTL$_{\text{P}}$ properties.

We assume the following expressions can be used to reason about the parameter formulae describing a specific PDTS:

- ${}^{t}trans(s, t) = \Phi$ such that $p \models \Phi \iff \exists d \in Dir : (s, d, p, t) \in {}^{t}\hat{T}$; that is a set of all parameters for which the transition is available regardless of the direction (preserving given time flow).
- ${}^{t}dir(s, \chi, t) = \Phi$ such that $p \models \Phi \iff \exists d \in Dir : (s, d, p, t) \in {}^{t}\hat{T} \wedge d \models \chi$; which represents all parameters for which the transition is enabled and the given direction constraint is satisfied.

When reasoning about the parameter formulae, we usually use semantic equality (i.e. $\Phi_1 \equiv \Phi_2 \iff \forall p \in \mathcal{P} : p \models \Phi_1 \iff p \models \Phi_2$). In cases where syntactic equality is used instead of semantic equality, this fact should be explicitly stated. Notice that this allows us to reason about parameter formulae similarly to standard sets, however this needs to be taken into account when reasoning about complexity of various operations.

2.6 Parameter Synthesis

The parameter synthesis problem for a PDTS $\mathcal{K}$, an initial parameter constraint Φ_I and a HUCTL$_P$ formula φ is to compute a function $\mathcal{F}$ such that $\mathcal{F}(s) = \{p \in \mathcal{P} \mid (\mathcal{K}_p, h_0, s) \models \varphi \wedge p \models \Phi_I\}$. Naive, enumerative solution would be to run standard global model checking procedure for each state. In Chapter 3, we introduce an algorithm which uses symbolic representation introduced in 2.5.2 to provide a faster method for deciding parameter synthesis.

2.7 Partitioning and PDTS fragments

2.7.1 Fragments

In order to distribute the PDTS to several independent workers, we define the notion of *fragments*. Any PDTS can be divided into N fragments using an injective partition function $f : S \to \{1, \ldots, N\}$. Intuitively, the partition function divides the state space of a PDTS into N disjoint groups (some of which may be empty) which we then call fragments. Such partitions can be then distributed to several independent agents to reduce the amount of resources required by a singular agent during algorithm execution. However, this also usually introduces a communication overhead.

We say that an agent i *owns* a state s when the state is part of the agents fragment, that is $f(s) = i$. We also call these states *local* and denote the set of all local states of the fragment i as $S_i^\bullet$. Note that the sets of the local states of each fragment are pairwise disjoint and their union constitutes the full state space S.

All states that are not local, but can directly reach or be reached from a local state are called *border* states of fragment i and are denoted $S_i^\bigcirc$. Formally, state $s \in S \setminus S_i^\bullet$ is a border state if and only if there exists $t \in S_i^\bullet$ such that $s \to t$ or $t \to s$. Please observe that the border state sets for different fragments can intersect and if s is a border state in fragment i, then t is a border state in fragment $f(s)$.

We write $S_i^{\circledcirc} = S_i^\bullet \cup S_i^\bigcirc$ to denote the set of the *relevant* states. That is, the states which directly influence the value of local states.

Finally, all states which are not local nor border states are called remote. The set of all remote states is denoted $S_i^{\times}$. Furthermore, for each fragment i holds that $S = S_i^{\bullet} \cup S_i^{\bigcirc} \cup S_i^{\times}$ and $\emptyset = S_i^{\bullet} \cap S_i^{\bigcirc} \cap S_i^{\times}$.

2.7.2 Partitioning

In this section, we also define two key properties of a partition function that influence the overall effectiveness of resource distribution and the expected communication overhead:

- *The uniformity of the partitioning*. That is, to ensure that the sizes of $S_i^{\bullet}$ are always almost equal. Naturally, exact equality can't always be achieved, however, a good uniform partition function should ensure that the maximum size difference between fragments does not grow with the size of the system, but rather with the number of partitions.

- *Number of cross transitions*. A cross transition is a transition that leads between two states in different fragments of the system. Such transition is usually a source of communication overhead, since any related operation usually involves both fragments and therefore requires some communication. Hence the amount of cross transitions directly influences the overall communication overhead. Also observe that this number is very closely related to the size of the $S_i^{\bigcirc}$ sets, since the states at each end of the cross transitions will be considered as border states in one of the fragments.

As we can see, in order to achieve good performance, the partition function needs to satisfy both of these properties. However, this can't be easily achieved for all systems. Without any prior knowledge of the system, one could design a partition function that would be close to uniformity, but that might also introduce a significant amount of cross transitions. Similarly, if one were to focus on optimizing the amount of cross transitions, the resulting partitioning might not be uniform enough to be able to distribute to agents with limited resources.

3 Algorithm

In this chapter, we describe the distributed algorithm used for the parameter synthesis computation on PDTS and HUCTL$_P$ properties.

3.1 Assumption semantics

In order to represent the intermediate results during the computation and accommodate for the distributed nature of the PDTS fragments, we introduce the notion of assumptions and related assumption semantic function for the HUCTL$_P$ formulae.

3.1.1 Assumption function

We define an assumption function for a PDTS fragment i as $\mathcal{A}_i(\psi, h, s) = (\Phi^\top, \Phi^\bot)$, meaning that the HUCTL$_P$ property ψ is assumed to hold in state $s \in S$ and valuation $h : Var \to S$ under parameter valuation $p \in \mathcal{P}$ such that $p \models \Phi^\top$ and symmetrically, ψ is assumed *not* to hold for parameter valuations $p \models \Phi^\bot$. Collectively, assumption functions of all fragments represent the total knowledge about the system accumulated so far.

We will write $\mathcal{A}_i^\top(\psi, h, s)$ and $\mathcal{A}_i^\bot(\psi, h, s)$ to denote an assumption function which returns just the first parameter constrain $\Phi^\top$ or second parameter constrain $\Phi^\bot$ respectively. Finally, when the fragment identifier is clear from context, we can omit the subscript i.

We say that an assumption $\mathcal{A}(\varphi, s, h) = (\Phi^\top, \Phi^\bot)$ about the PDTS $\mathcal{K}$ is *valid* when for all p holds that:

$$(p \models \Phi^\top \Rightarrow (\mathcal{K}_p, s, h) \models \varphi) \wedge (p \models \Phi^\bot \Rightarrow (\mathcal{K}_p, s, h) \not\models \varphi)$$

From now on, we will only consider valid assumption functions unless stated otherwise. Furthermore, from this definition, we can conclude several important properties of valid assumption functions:

- Assumption implies satisfaction, but satisfaction does not necessarily imply assumption.

- Parameter constrain $\mathcal{A}^{\top}(\psi, s, h) \wedge \mathcal{A}^{\bot}(\psi, s, h)$ can never be satisfiable - i.e. ψ cannot be assumed to be valid and invalid at the same time.
- Parameter constrain $\neg\mathcal{A}^{\top}(\psi, s, h) \wedge \neg\mathcal{A}^{\bot}(\psi, s, h)$ can be satisfiable. This situation signifies that for some parameter valuations the result is yet *unknown*. We will denote this parameter constrain as $\mathcal{A}^{?}(\psi, s, h)$.
- These intuitive equalities hold only when the set of unknown parameter valuations is empty:

$$\mathcal{A}^{\top}(\psi, s, h) \not\equiv \neg\mathcal{A}^{\bot}(\psi, s, h)$$
$$\mathcal{A}^{\top}(\psi, s, h) \not\equiv \neg\mathcal{A}^{\top}(\neg\psi, s, h)$$

  The expression $\neg\mathcal{A}^{\bot}(\psi, s, h)$ represents not only parameters for which ψ is satisfied, but also parameters for which the result is unknown. Similarly, in the second case, $\neg\mathcal{A}^{\top}(\neg\psi, s, h)$ denotes not only parameters for which the $\neg\psi$ is not satisfied (hence ψ is satisfied) but also unknown parameter valuations.

We use $\mathcal{A}_0$ to denote an assumption function which has all results initialised to $(\texttt{ff}, \texttt{ff})$, hence the set of the valid and invalid parameter valuations is empty and the set of the unknown parameter valuations is the whole parameter universe.

Finally, we assume a partial ordering on the set of all possible assumption functions over PDTS fragment $\mathcal{K}_i$ based on the standard set inclusion relation. Formally:

$$\mathcal{A}_1 \leq \mathcal{A}_2 \iff \forall\psi, s, h : \mathcal{A}_2^{?}(\psi, s, h) \subseteq \mathcal{A}_1^{?}(\psi, s, h)$$

Under such ordering, the set of all assumption functions forms a complete lattice. This ordering captures the intuitive understanding of computation progress, because an assumption function $\mathcal{A}_2$ is bigger than $\mathcal{A}_1$ only if the complete knowledge about the system is higher

in $\mathcal{A}_2$. Also observe that the highest element of every chain is an assumption function with no unknown parameters while the lowest element is the $\mathcal{A}_0$.

3.1.2 Semantic function

In order to compute the maximal valid assumption function for a PDTS fragment $\mathcal{K}_i = (\mathcal{P}, S, Dir, \hat{T}, AP, L)$, we define a semantic function $\mathcal{C}_\mathcal{K}(\mathcal{A}_{in}) = \mathcal{A}_{out}$ which (if possible) increases the given assumption (with respect to the ordering) while preserving assumption validity.

In this definition, we do not use the full operator set of HUCTL$_\mathrm{P}$, but rather the simplified set of operators defined in section 2.4.6. The definition uses a special assignment operator $\mathcal{A}(\psi, s, h) \Leftarrow \Phi$ which means that the parameter constrain Φ is added to the current set of assumptions, formally:

$$\mathcal{A}(\psi, s, h) \Leftarrow \Phi \equiv \mathcal{A}(\psi, s, h) \leftarrow \mathcal{A}(\psi, s, h) \vee \Phi$$

Also observe that this operation is monotonic with respect to the assumption function ordering, that is, the set of unknown parameter valuations can only decrease by applying $\Leftarrow$.

The definition of the semantic function itself is divided into several parts, based on the HUCTL$_\mathrm{P}$ operators it deals with. Each part declaratively defines the new assumption function $\mathcal{A}_{out}$ based on the given $\mathcal{A}_{in}$.

First part, presented in Figure 3.1, describes the initial assumptions of the system. Please observe that in all cases, no unknown parameters are present. This naturally follows from the fact that the validity of the atomic formulae depends solely on the properties of the system and the variable valuation, and hence can be resolved unambiguously.

In the second part, shown in Figure 3.2, we provide the semantics for the temporal operators. As we can see, the differences between operators and their positive and negative semantics are often very subtle, but crucial.

First of all, let us observe that due to the negation inequality, we can't simply handle the negative assumptions as negations of the positive ones. Second, we see that the observations about the nature of weak operators made in the section 2.4.5 are still valid—that is, one

$$\mathcal{A}_{out}(true, s, h) \Leftarrow (\texttt{tt}, \texttt{ff})$$

$$\mathcal{A}_{out}(p, s, h) \Leftarrow \begin{cases} (\texttt{tt}, \texttt{ff}) & p \in L(s) \\ (\texttt{ff}, \texttt{tt}) & \texttt{otherwise} \end{cases}$$

$$\mathcal{A}_{out}(x, s, h[x \mapsto s']) \Leftarrow \begin{cases} (\texttt{tt}, \texttt{ff}) & s = s' \\ (\texttt{ff}, \texttt{tt}) & \texttt{otherwise} \end{cases}$$

Figure 3.1: Definition of semantic function $\mathcal{C}$ for atomic propositions, $p \in AP$ is an atomic proposition and $x \in Var$ is a state variable.

can clearly observe the strict ($\wedge$) and weak($\Rightarrow$) direction requirement in all definitions.

Finally, the semantics of the remaining boolean and hybrid operators are described in Figure 3.3.

Notice that except for the *at* operator, the semantic function uses only the *relevant* states of the PDTS fragment. Indeed, the *at* operator is fundamentally designed to be able to access any state in the system and hence can't be subject to such requirement.

3.1.3 Semantic function fixed point

As we can see, the semantic function computed according to the rules presented in this section is clearly monotonic (thanks to the $\Leftarrow$ operator) with respect to the complete lattice formed by the assumption functions of $\mathcal{K}$. Therefore according to the Knaster-Tarski theorem [14], $\mathcal{C}$ has a least fixed point which can be computed by repeated application of $\mathcal{C}$.

3.1.4 Semantic function validity

Even though the monotonicity of the semantic function is rather obvious, the validity of the resulting assumption function does not have to be entirely clear. Especially for the temporal operators, which are now

$$
\begin{aligned}
\mathcal{A}^{\top}_{out}({}^{t}\mathbf{E}\,\mathbf{X}_{\chi}\varphi, s, h) &\Leftarrow \bigvee_{s' \in S^{\odot}} {}^{t}trans(s, s') \wedge [{}^{t}dir(s, \chi, s') \wedge \mathcal{A}^{\top}_{in}(\varphi, s', h)] \\
\mathcal{A}^{\bot}_{out}({}^{t}\mathbf{E}\,\mathbf{X}_{\chi}\varphi, s, h) &\Leftarrow \bigwedge_{s' \in S^{\odot}} \neg {}^{t}trans(s, s') \vee [{}^{t}dir(s, \chi, s') \Rightarrow \mathcal{A}^{\bot}_{in}(\varphi, s', h)] \\
\mathcal{A}^{\top}_{out}({}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\varphi, s, h) &\Leftarrow \bigwedge_{s' \in S^{\odot}} \neg {}^{t}trans(s, s') \vee [{}^{t}dir(s, \chi, s') \wedge \mathcal{A}^{\top}_{in}(\varphi, s', h)] \\
\mathcal{A}^{\bot}_{out}({}^{t}\mathbf{A}\,\mathbf{X}_{\chi}\varphi, s, h) &\Leftarrow \bigvee_{s' \in S^{\odot}} {}^{t}trans(s, s') \wedge [{}^{t}dir(s, \chi, s') \Rightarrow \mathcal{A}^{\bot}_{in}(\varphi, s', h)] \\
\mathcal{A}^{\top}_{out}({}^{t}\mathbf{E}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s, h) &\Leftarrow \mathcal{A}^{\top}_{in}(\varphi, s, h) \vee \\
&\qquad \bigvee_{s' \in S^{\odot}} {}^{t}trans(s, s') \wedge [{}^{t}dir(s, \chi, s') \Rightarrow \mathcal{A}^{\top}_{in}({}^{t}\mathbf{E}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s', h)] \\
\mathcal{A}^{\bot}_{out}({}^{t}\mathbf{E}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s, h) &\Leftarrow \mathcal{A}^{\bot}_{in}(\varphi, s, h) \wedge \\
&\qquad \bigwedge_{s' \in S^{\odot}} \neg {}^{t}trans(s, s') \vee [{}^{t}dir(s, \chi, s') \wedge \mathcal{A}^{\bot}_{in}({}^{t}\mathbf{E}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s', h)] \\
\mathcal{A}^{\top}_{out}({}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s, h) &\Leftarrow \mathcal{A}^{\top}_{in}(\varphi, s, h) \vee \\
&\qquad \bigwedge_{s' \in S^{\odot}} \neg {}^{t}trans(s, s') \vee [{}^{t}dir(s, \chi, s') \Rightarrow \mathcal{A}^{\top}_{in}({}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s', h)] \\
\mathcal{A}^{\bot}_{out}({}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s, h) &\Leftarrow \mathcal{A}^{\bot}_{in}(\varphi, s, h) \wedge \\
&\qquad \bigvee_{s' \in S^{\odot}} {}^{t}trans(s, s') \wedge [{}^{t}dir(s, \chi, s') \wedge \mathcal{A}^{\bot}_{in}({}^{t}\mathbf{A}_{\chi}\,\widetilde{\mathbf{F}}\,\varphi, s', h)] \\
\mathcal{A}^{\top}_{out}({}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s, h) &\Leftarrow \mathcal{A}^{\top}_{in}(\varphi_2, s, h) \vee [\mathcal{A}^{\top}_{in}(\varphi_1, s, h) \wedge \\
&\qquad \bigvee_{s' \in S^{\odot}} {}^{t}trans(s, s') \wedge [{}^{t}dir(s, \chi, s') \wedge \mathcal{A}^{\top}_{in}({}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s', h)]] \\
\mathcal{A}^{\bot}_{out}({}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s, h) &\Leftarrow \mathcal{A}^{\bot}_{in}(\varphi_2, s, h) \wedge [\mathcal{A}^{\bot}_{in}(\varphi_1, s, h) \vee \\
&\qquad \bigwedge_{s' \in S^{\odot}} \neg {}^{t}trans(s, s') \vee [{}^{t}dir(s, \chi, s') \Rightarrow \mathcal{A}^{\bot}_{in}({}^{t}\mathbf{E}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s', h)]] \\
\mathcal{A}^{\top}_{out}({}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s, h) &\Leftarrow \mathcal{A}^{\top}_{in}(\varphi_2, s, h) \vee [\mathcal{A}^{\top}_{in}(\varphi_1, s, h) \wedge \\
&\qquad \bigwedge_{s' \in S^{\odot}} \neg {}^{t}trans(s, s') \vee [{}^{t}dir(s, \chi, s') \wedge \mathcal{A}^{\top}_{in}({}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s', h)]] \\
\mathcal{A}^{\bot}_{out}({}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s, h) &\Leftarrow \mathcal{A}^{\bot}_{in}(\varphi_2, s, h) \wedge [\mathcal{A}^{\bot}_{in}(\varphi_1, s, h) \vee \\
&\qquad \bigvee_{s' \in S^{\odot}} {}^{t}trans(s, s') \wedge [{}^{t}dir(s, \chi, s') \Rightarrow \mathcal{A}^{\bot}_{in}({}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2], s', h)]]
\end{aligned}
$$

Figure 3.2: Definition of semantic function $\mathcal{C}$ for temporal operators.

$$
\begin{aligned}
\mathcal{A}^{\top}_{out}(\varphi_1 \wedge \varphi_2, s, h) &\Leftarrow \mathcal{A}^{\top}_{in}(\varphi_1, s, h) \wedge \mathcal{A}^{\top}_{in}(\varphi_2, s, h) \\
\mathcal{A}^{\bot}_{out}(\varphi_1 \wedge \varphi_2, s, h) &\Leftarrow \mathcal{A}^{\bot}_{in}(\varphi_1, s, h) \vee \mathcal{A}^{\bot}_{in}(\varphi_2, s, h) \\
\mathcal{A}^{\top}_{out}(\exists x : \varphi, s, h) &\Leftarrow \bigvee_{s' \in S} \mathcal{A}^{\top}_{in}(\varphi, s, h[x \mapsto s']) \\
\mathcal{A}^{\bot}_{out}(\exists x : \varphi, s, h) &\Leftarrow \bigwedge_{s' \in S} \mathcal{A}^{\bot}_{in}(\varphi, s, h[x \mapsto s']) \\
\mathcal{A}_{out}(\neg\varphi, s, h) &\Leftarrow (\mathcal{A}^{\bot}_{in}(\varphi, s, h), \mathcal{A}^{\top}_{in}(\varphi, s, h)) \\
\mathcal{A}_{out}(\downarrow x : \varphi, s, h) &\Leftarrow \mathcal{A}_{in}(\varphi, s, h[x \mapsto s]) \\
\mathcal{A}_{out}(@x : \varphi, s, h[x \mapsto s']) &\Leftarrow \mathcal{A}_{in}(\varphi, s', h[x \mapsto s'])
\end{aligned}
$$

Figure 3.3: Definition of semantic function $\mathcal{C}$ for hybrid and boolean operators.

defined recursively based on the PDTS transitions rather than using the infinite runs.

Lemma 1. *Given a valid assignment $\mathcal{A}_{in}$, the semantic function $\mathcal{C}(\mathcal{A}_{in})$ produces a valid assignment.*

Proof First, we consider the validity of the atomic proposition assumptions and the hybrid operator assumptions as obvious, since they can be almost effortlessly translated to their direct definitions.

Next, we show a full validity proof for the positive assumption of the ${}^{t}\mathbf{A}_{\chi}\,\mathbf{U}$ operator—one of the most complex temporal operators. Proofs for other operators (and their negative counterparts) are very similar and therefore we won't list them in detail.

- Assume all current $\mathcal{A}^{\top}_{in}$ and $\mathcal{A}^{\bot}_{out}$ are valid and we are considering $\psi = {}^{t}\mathbf{A}[\varphi_{1\chi}\,\mathbf{U}\,\varphi_2]$.
- For a contradiction, let us assume that after application of $\mathcal{C}$, there exists a parameter valuation $p \in \mathcal{P}$ such that $p \models \mathcal{A}^{\top}_{out}(\psi, s, h)$ and $(\mathcal{K}_p, s, h) \not\models \psi$ for some s and h.
- According to the definition of $\mathcal{C}$, this can happen in two cases:

- $p \models \mathcal{A}^{\top}_{in}(\varphi_2, s, h)$ - this is in direct contradiction with the operator definition, since for all runs π there exists $i = 0$ such that $(\mathcal{K}_p, \pi_S(i), h) \models \varphi_2$ because $\pi_S(0) = s$. Therefore $(\mathcal{K}_p, s, h) \models \psi$ (the direction requirement is trivially satisfied since there is no j smaller than 0).
- $p \models [\mathcal{A}^{\top}_{in}(\varphi_1, s, h) \wedge \bigwedge_{s' \in S} \neg {}^t trans(s, s') \vee [{}^t dir(s, \chi, s') \wedge \mathcal{A}^{\top}_{in}({}^t \mathbf{A}[\varphi_{1\chi} \mathbf{U} \varphi_2], s', h)]]$. Hence not only $(\mathcal{K}_p, s, h) \models \varphi_1$, but also for all s' either the transition from s is not present, or if it is present, it satisfies the direction requirement, and $p \models \mathcal{A}^{\top}_{in}({}^t \mathbf{A}[\varphi_{1\chi} \mathbf{U} \varphi_2], s', h)$. Thanks to this fact, we see that every infinite run starting in s has to use one of these transitions. Therefore the only case when $(\mathcal{K}_p, s, h) \not\models \psi$ is when either the assumption $\mathcal{A}^{\top}_{in}(\varphi_1, s, h)$ or some of the assumptions $\mathcal{A}^{\top}_{in}({}^t \mathbf{A}[\varphi_{1\chi} \mathbf{U} \varphi_2], s', h)$ are not valid, which is a contradiction with our original assumption.

□

3.2 Main algorithm

In this section, we present the main algorithm which relies on the previously defined assumption function and the assumption fixed point.

3.2.1 Environment and data structures

Before we describe the main algorithm, we have to introduce our assumptions about the environment the algorithm will be executed in.

The algorithm presented in this work assumes a distributed environment of N independent, reliable agents connected using reliable communication channels. In practice, such model is suitable to model multi core machines and small to medium computational clusters where fault tolerance is not strictly necessary.

We assume that each agent has local access to the following data structures:

- Total number of communicating agents N;

- A unique agent identifier $id \in \{1, \ldots, N\}$;
- A partition function $f : S \to \{1, \ldots, N\}$;
- A PDTS fragment $\mathcal{K}_i$;
- A HUCTL$_P$ formula φ;

Note that not all of these structures have to be represented explicitly. For example, the PDTS fragment and the partition function is usually computed on-the-fly.

3.2.2 Algorithm pseudocode

The Algorithm 1 starts with a fully unknown assignment and iteratively computes new fixed point assumptions. As soon as the fixed point is reached, a round of communication is performed to ensure all remaining fragments are notified about the updated assumptions. At this point, information from other fragments is also received. As soon as no new information can be gained using this procedure, the algorithm proceeds to mark unknown parameter valuations as unsatisfied, assuming all sub-formulae have been successfully computed.

Here, the TerminationDetection procedure will return true only if the assumptions have not changed since last termination attempt and no communication is taking place. We don't discuss a specific implementation for this procedure, since termination detection in distributed systems is a well studied problem [15] and an optimal implementation usually depends on the available communication primitives in our environment.

To simplify the description of the Communication procedure, we assume that each agent can directly update assumptions of other agents. This is of course an unrealistic expectation in a distributed system. In reality, this form of communication assumes a proper message is sent, received and handled by the receiving agent. However, we do not provide an explicit pseudocode for this, since it is, similar to the TerminationDetection, a problem which largely depends on the available environment (in a multi-core machine simulating the distributed environment, even a direct update can be possible) and has been efficiently solved before [16].

Algorithm 1 Main fixed point algorithm.

1: **function** ParameterSynthesis(fragment $\mathcal{K}_{id}$, property φ)
2: $\quad\mathcal{A}_{id} \leftarrow \mathcal{A}_0$
3: $\quad$**repeat**
4: $\quad\quad$**repeat**
5: $\quad\quad\quad$**repeat**
6: $\quad\quad\quad\quad\mathcal{T} \leftarrow \mathcal{A}_{id}$
7: $\quad\quad\quad\quad\mathcal{A}_{id} \leftarrow \mathcal{C}(\mathcal{T})$
8: $\quad\quad\quad$**until** $\mathcal{T} = \mathcal{A}_{id}$
9: $\quad\quad\quad$Communicate$(\mathcal{A}_{id})$
10: $\quad\quad$**until** TerminationDetection
11: $\quad\quad$IncreaseCycles$(\mathcal{A}_{id}, \varphi)$
12: $\quad$**until** TerminationDetection
13: $\quad\mathcal{F}_i(s) \leftarrow \mathcal{A}_{id}^{\top}(\varphi, s, h_0)$
14: **function** IncreaseCycles(assumption function $\mathcal{A}$, property φ)
15: $\quad$for $s \in S^{\bullet} : Done(h, s) \leftarrow \texttt{tt}$
16: $\quad$**for all** $\psi \in sub(\varphi)$ **do**
17: $\quad\quad Done(h, s) = Done(h, s) \wedge \neg$IncreaseCycles$(\mathcal{A}, \psi)(h, s)$
18: $\quad$for $s \in S^{\bullet} : \mathcal{A}^{\bot}(\varphi, h, s) \Leftarrow \mathcal{A}^{?}(\varphi, h, s) \wedge Done(h, s)$
19: $\quad$**return** $\mathcal{A}^{?}(\varphi)$
20: **function** Communicate(assumption function $\mathcal{A}$)
21: $\quad$**for all** $s \in S_{id}^{\bullet}$ **do**
22: $\quad\quad$**for all** $i < N$ such that $s \in S_i^{\bigcirc}$ **do**
23: $\quad\quad\quad$for all $\psi, h : \mathcal{A}_i(\psi, h, s) \Leftarrow \mathcal{A}_{id}(\psi, h, s)$
24: $\quad\quad$**for all** $i < N$ and $\psi : (@x : \psi) \in cl(\varphi)$ **do**
25: $\quad\quad\quad$for all $h, s : \mathcal{A}_i(\psi, h, s) \Leftarrow \mathcal{A}_{id}(\psi, h, s)$

Note that, as we also discussed in the previous section, the *at* operator requires special treatment, due to its ability to access arbitrary states.

Finally, let us observe that the complete solution to the parameter synthesis problem can be constructed from the partial solutions computed by each fragment as follows:

$$\mathcal{F}(s) = \bigvee_{i \in \{1,\ldots,N\}} \mathcal{F}_i(s)$$

3.2.3 Correctness

Termination

Theorem 1. *Given a PDTS fragment $\mathcal{K}_{id}$ and a HUCTL$_P$ property φ, the Algorithm 1 eventually terminates.*

Proof First, let us observe that since the least fixed point of $\mathcal{C}(\mathcal{A})$ always exists, the condition at line 8 will also always terminate as soon as this fixed point is reached. Furthermore, since the communication function is also monotonic with respect to the assumption ordering, the condition at line 10 will also eventually reach a fixed point as soon as no new assumptions can be computed either by applying the semantic function, or communicating with other processes.

Finally, the function IncreaseCycles is also monotonic, and therefore the same argument holds for the condition on line 12. Eventually, all agents will reach a global fixed point and at that point, the system will terminate. □

Partial Correctness

Before we get to the main theorem, let us prove several additional lemmas about the whole algorithm that will greatly simplify the correctness proof:

Lemma 2. *Procedure* Communicate *preserves validity of the assumption function.*

Proof This is almost obvious, since the Communicate procedure will only increase assumptions in border states of fragment i if current fragment id successfully computed said assumption. Therefore, assuming the information for the local states in this fragment is valid (given that the semantic funciton $\mathcal{C}$ preserves assumption validity), the newly assigned information for the border states is also valid.

The exception being the *at* operator, which is always delivered to all fragments, regardless of border states. However, this also adheres to the global *at* semantics. □

Lemma 3. *Assuming no value of $\mathcal{A}_i(\psi, h, s)$ (globally for all i) can be increased either by applying $\mathcal{C}$ or by performing* Communicate*, the procedure* IncreaseCycles *preserves validity of the assumption function.*

Proof It should be noted that the assumption of the lemma corresponds to the conditions under which is the IncreaseCycles called in the algorithm (the conditions are ensured by the repeat-until loop on lines 4-10).

Second, the procedure can increase only negative assumptions about the system. Therefore, for a contradiction, let us assume that after application of IncreaseCycles there is a fragment $\mathcal{K}_i$, a parameter valuation p, a property ψ, a variable assignment h and a state s, such that $p \models \mathcal{A}_i^{\perp}(\psi, h, s)$ but $(\mathcal{K}_p, h, s) \models \psi$.

This can only happen when the assumptions about p are known for all sub-formulae of ψ, since the only time when the unknown assumption can increase is when the parameter valuation is marked as *Done*, which means it is not unknown in any of the sub-formulae.

Because assumptions for all sub-formulae are known, the only operators for which this error can occur are the temporal operators. That is because all other operators depend directly on their sub-formulae, and therefore when the assumptions for the sub-formulae are known, the assumption for the formula should also be known when the fixed point is computed. This would imply that the fixed point was not computed properly, contradicting the original assumption of the lemma.

However, the assumption of the computation and communication fixed point also implies that the only possible case when assumption for such operator can be unknown is in case of a circular dependency between assumptions. Any other case would imply that either the communication has not been handled properly (and therefore we

have unknown parameter valuations that should have been resolved using communication) or the fixed point has not been reached yet (and therefore some parameter valuations can still increase due to the assumptions about their successors).

Finally, if a parametrisation is unknown due to a circular dependency between assumptions, it can be safely assumed to be negative. That is because such assumption cannot ever increase to a positive value, therefore also the property cannot hold in such state. □

Lemma 4. *When the algorithm exits the main cycle and enters line 13, all unknown assumptions for the local states $\mathcal{A}^?(\psi, h, s)$ are empty.*

Proof This is easy to show, because in each iteration, the procedure IncreaseCycles increases at least one unknown parameter valuation to the negative value for some state (assuming unknown parameter valuations exist).

Therefore, assuming there are some unknown assumptions, there must exists the smallest formula ψ such that some of its assumptions are unknown and yet all sub-formulae are already known. These unknown assumptions are then promoted to negative ones by the IncreaseCycles procedure. Hence after a finite number of repetitions, IncreaseCycles increases all unknown parameter valuations to negative ones. □

Theorem 2. *Algorithm 1 computes the parameter synthesis problem for a given PDTS $\mathcal{K}$ and a given HUCTL$_P$ property φ.*

Proof By Lemma 4, there are no unknown parameter valuations for the local states of each fragment. Therefore an union across all fragments ensures that there are no unknown parameter valuations across the whole state space. Furthermore, since the assumption functions of each fragment are valid (by Lemma 1, 2 and 3), the resulting assumption function is also valid. □

3.2.4 Notes on the semantic function and complexity

One notable part of the algorithm that has been intentionally left out of the main pseudocode is how to compute the semantic function (and its fixed point). Similar to the TerminationDetection and the Communication procedures, there are several possible approaches

which depend on the nature of the model, the execution environment and other desired properties of the algorithm. Instead of providing the full pseudocode for this operation, we provide a discussion of several observations and approaches that can be taken when implementing the semantic function fixed point and we refer the reader to the actual implementation of the algorithm for one specific example.

- In the definition of the semantic function, assumptions about the temporal operators usually depend on all local and border states. This requirement can be easily reduced to just the actual successors (for some parameter valuation). In transition systems where a state has only a small number of potential successors, this means that the number of assumption dependencies is often not linear in the size of state space, but rather constant or at least logarithmic.

- Complexity can be further reduced by memorizing intermediate results of operations. When assumption in state s depends on its k successors, one can build a tree-like structure that can be then updated in logarithmic time when one of the dependencies changes.

- Since determining whether a value of an assumption changed or not requires a semantic equality check, which is often a very expensive operation, it can be beneficial to delay such check as long as possible (without introducing additional checks later). An example of such delaying would be to ensure that the check is performed only when no dependencies of the assumption under question are still being updated (or waiting to be updated). In other words, one can process the dependencies in a "distance-to-proposition" order, ensuring the values are updated only when necessary.

- It is crucial for the effectiveness of the algorithm to maintain a compact representations of the parameter formulae produced during computation. To this end, instead of using a simple formula solver, one should look for solvers that can also simplify the investigated formulae. However, such operations are usually even harder than standard formula solving. It might be therefore

beneficial to employ heuristics that can delay formula simplification when not needed or that can separate parts of the formula that cannot be further simplified.

4 Implementation

The algorithm presented in this work is available as an open source implementation. This implementation forms a key part in the Pithya [17] parameter synthesis tool for ODE based biochemical models. The source code and manual are also provided as digital appendices.

In this chapter, we discuss the architecture and characteristics of this implementation.

4.1 Pithya core overview

The Pithya tool has two main components: *graphical user interface* and the *core engine*. Here, we are concerned only about the core engine.

The core engine is implemented in an object-oriented manner using the Kotlin programming language (compiles to standard JVM bytecode). Furthermore, the engine can use the Microsoft Z3 SMT solver [18] for decisions about the parameter formulae.

The core engine itself is also divided into several modules:

- **Temporal Logic Module** This module is responsible for parsing the HUCTL$_P$ formulae and performing necessary transformations to ensure the formulae use only the supported set of operators. The input format of the HUCTL$_P$ formulae is specified as an ATLR4 [19] grammar.

- **Parameter Synthesis Module** The main module containing the algorithm itself with abstract definitions of the necessary data structures such as solver, state map or model.

- **ODE Model Module** Defines a parser for the `.bio` ODE model files and a set of solvers, successor generators and state maps that work with ODE models.

- **CLI Front-end** Provides a command line interface, combining the functionality of all modules into one executable.

In the following sections we will discuss these modules in detail.

4.2 Parameter Synthesis Module

4.2.1 States and parameter formulae representation

Before describing the components of the parameter synthesis module, we have to define the basic requirements it poses on anyone willing to use it:

- States of the PDTS all have unique (even across fragments) integer identifiers from a continuous range. This allows easier partitioning and provides room for interesting optimisations.
- On the other hand, the parameter formula representation is fully generic, allowing the user to choose whatever domain specific representation suits their needs. The only requirement is that the user provides a solver capable of performing basic operations required by the algorithm (discussed later in this section).

4.2.2 User-implemented interfaces

Now that we have described how the parameter synthesis module approaches states and parameters, we can describe basic interfaces that need to be implemented by potential users. Here, we provide the list of all of these interfaces with short descriptions of their functionality. However, the implementations have to adhere to a set of required invariants and synchronization rules in order for the algorithm to be valid, therefore we refer the reader to the source code documentation for more detailed information about each interface.

- `StateMap` State map is a simple map interface which provides a way to represent the state—parameter mapping used when computing the assumption function. However, as opposed to the assumption function, `StateMap` is a general purpose interface used throughout the code whenever a state—parameter collection is needed (successor/predecessor representation, communication, etc.). It is immutable by default, however there is a mutable variant which is used to represent incomplete results. For the list of available implementations, see code documentation.

- `Solver` The solver should be capable of providing basic constants (`tt`, `ff`) and performing standard operations such as: Conjunction, disjunction, complement (negation), test for emptiness (satisfiability) and formula simplification. These operations are then used to implement more complex, algorithm specific operations. User is also free to override these default implementations assuming a more efficient alternative is available. Finally, each solver should be able to serialize a parameter constraint into a byte buffer so that it can be safely transferred between fragments. Additionally, the module provides sample explicit solvers based on standard collections. These usually don't scale very well with increasing number of parameter valuations, but provide a good starting point for implementing and debugging more complex solvers (for full list, see code documentation).

- `Partition` Combining the PDTS fragment with its partition function, the `Partition` interface provides the total amount of fragments, current fragment identifier, methods for obtaining predecessors and successors of a specific state plus the ability to evaluate atomic propositions.

  Assuming the user does not want to provide his own partition function, they can implement a `Model` interface, which is a simplified version of the `Partition` which provides only the successor/predecessor generator and proposition evaluation. The `Model` can then be wrapped into one of the predefined partition functions:

  - `SingletonPartition` Partition function maps all states to a single fragment. Useful for debugging or working in single threaded environment.
  - `HashPartition` Partition which assigns states to a predefined number of fragments using an integer modulus as a hash function. It provides good levels of uniformity and concurrency, however, usually also requires a lot of communication.
  - `UniformPartition` A uniform partition divides the states into equally sized intervals and assigns each fragment one

interval. It provides good uniformity and assuming the identifiers of state neighbours are also numerically close to the identifier of the original state, it should provide low communication overhead. However, in cases when the communication cost is low, the better concurrency of the `HashPartition` can result in faster computation.

- `BlockPartition` A block partition is a hybrid between the `UniformPartition` and the `HashPartition`. The partition function will divide the state space into equally sized intervals, while each fragment is assigned a predefined number of intervals.

Apart from the predefined partitions, the parameter synthesis module also provides a very basic explicit model implementation, which can be useful for debugging, testing and creating toy examples (it is used as a model implementation for the validity testing).

- `Channel` Responsible for communication between fragments, functionality of `Channel` maps almost directly to the Communicate procedure in the algorithm pseudocode. The communication relies on serialization into byte buffers.

  The module provides two basic implementations:

  - `SingletonChannel` Channel used for single threaded workloads with no ability to communicate.
  - `SharedMemChannel` Channel which directly passes the byte buffers between fragments managed by the same virtual machine.

  As you can see, no truly distributed channel is provided directly by the module. Users should provide their own channel based on the distributed environment where the algorithm is running.

4.2.3 Module work flow

With all necessary data structures in place, the parameter synthesis engine accepts a set of investigated HUCTL$_P$ properties and is ready

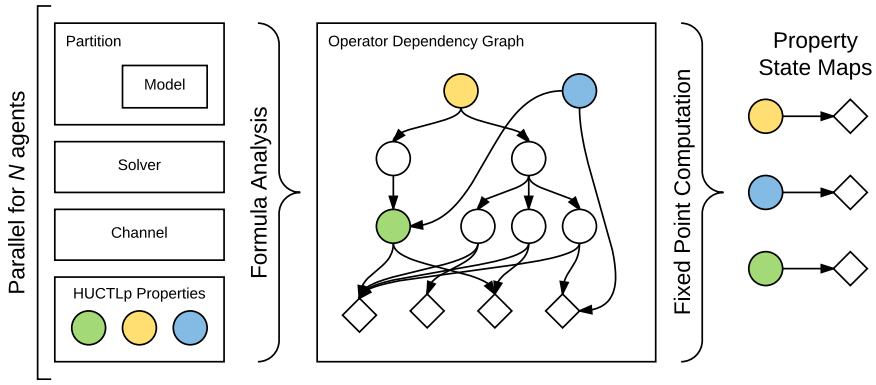


Figure 4.1: Work flow of the main parameter synthesis module. Circles represent HUCTL$_P$ operator objects, diamonds `StateMaps`.

to perform the main procedure. The work flow is depicted in figure 4.1.

The implementation starts by constructing a dependency graph based on the provided HUCTL$_P$ properties. Each node in the graph is represented by a special operator object which implements the logic of the semantic function $\mathcal{C}$ for one specific HUCTL$_P$ operator.

This construction ensures that whenever two properties share a common sub-formula, it is only computed once. Furthermore, this construction also unrolls the state variable valuation, so that for example a property $@x : \varphi$ is represented as $|S|$ distinct operators, depending on the value of x.

This allows us to ensure that unused valuations don't create unnecessary operators. A good example of such case is a formula $\downarrow x : \mathbf{A}\,\mathbf{F}(\neg x \wedge \mathbf{E}\,\mathbf{F}\,p)$ where p is some atomic proposition. In this case, $\mathbf{E}\,\mathbf{F}$ is independent on the valuation of x, and therefore can be represented by a single operator node in the dependency graph, while the $\mathbf{A}\,\mathbf{F}$ is unrolled into multiple operators depending on the value of x.

Another important optimisation this construction allows is canonisation of the state variable names. This means that if you consider a formula such as $[\downarrow x : \mathbf{A}\,\mathbf{F}\,x] \vee [\exists y : (\mathbf{A}\,\mathbf{F}\,y \wedge \neg y)]$, both $\mathbf{A}\,\mathbf{F}\,x$ and $\mathbf{A}\,\mathbf{F}\,y$ are resolved as the same operator object and are computed only once.

After the operator dependency graph is constructed, the fixed point algorithm iteratively processes operators in the graph, starting from

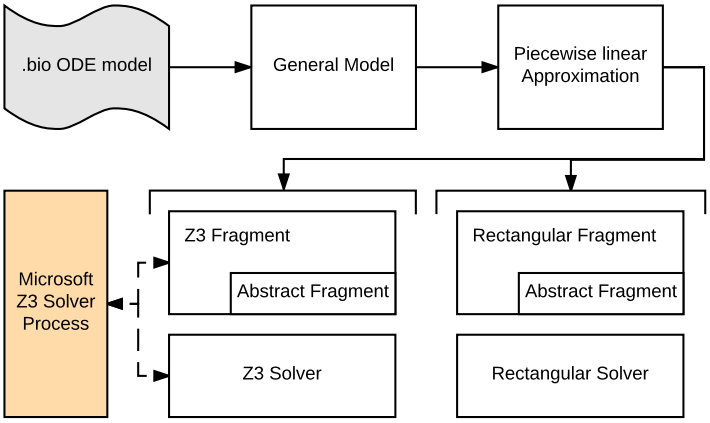


Figure 4.2: Work flow of the ode module module. The usage of a specific fragment implementation depends on the properties of the model.

the smallest (propositions). Multiple operators can be even processed concurrently, assuming their dependencies are already computed.

As a result of this operation, user obtains a set of `StateMaps` which represent the parameter synthesis result for the local states of the fragment for each requested property, as depicted in the third part of the work flow figure.

4.3 ODE Model Module

The ODE model module provides implementations for working with various types of models based on ordinary differential equations. It relies on the abstraction procedure described in [20, 21] when transforming the continuous equations into a discrete state space. The models are originally represented using a .bio model format (grammar provided in ANTLR4 syntax in the digital appendix) and this module is responsible for parsing the model file, computing the piecewise linear approximation and transforming the model into PDTS fragments that can be then used by the main parameter synthesis module. The module also provides appropriate solvers for the supported model types.

The work flow of this process, with all of it's main components is depicted in figure 4.2. After the piecewise linear approximation is constructed, the model is analysed for relationships between parameters and an appropriate parameter representation is chosen.

`Rectangular Model` and the corresponding `Rectangular Solver` are chosen when the parameters are fully independent (at most one parameter per equation). This type of parameter representation uses a set of hyper-rectangles to encode the parameter constrain and is fairly straightforward and efficient.

`Z3 Model` and `Z3 Solver` are then used for more complex models. In this case, the parameter constrains are represented directly as Microsoft Z3 compatible formulae and the SMT solver is directly responsible for deciding satisfiability and formula simplification. This is facilitated by directly interfacing with a Z3 process running in interactive mode. Z3 also provides a direct API for common programming languages. However, we choose not to use this, because it causes problems with parallelism on multi-core machines.

At the time of writing, Z3 was designed in a way that even for formulae which belong to completely separate contexts, some non-trivial synchronisation is performed. This synchronisation then significantly cripples the parallel execution to a point where adding more concurrent workers actually slows down the execution significantly. Having a set of independent (at the system level) solver processes then enables us to scale the execution on multi-core machines, even though it introduces more overhead and complexity to the whole program.

Finally, this module also provides an `Abstract Model` class which implements general logic for generating transitions in ODE based models. One can use this class to quickly implement new model variants. All that needs to be provided is a method producing a parameter constrain under which is a given equation at a given vertex positive or negative (corresponding to a positive or negative derivation). Other logic and optimisations (caching, etc.) is then handled solely by the abstract implementation. Naturally, an appropriate solver also has to be provided.

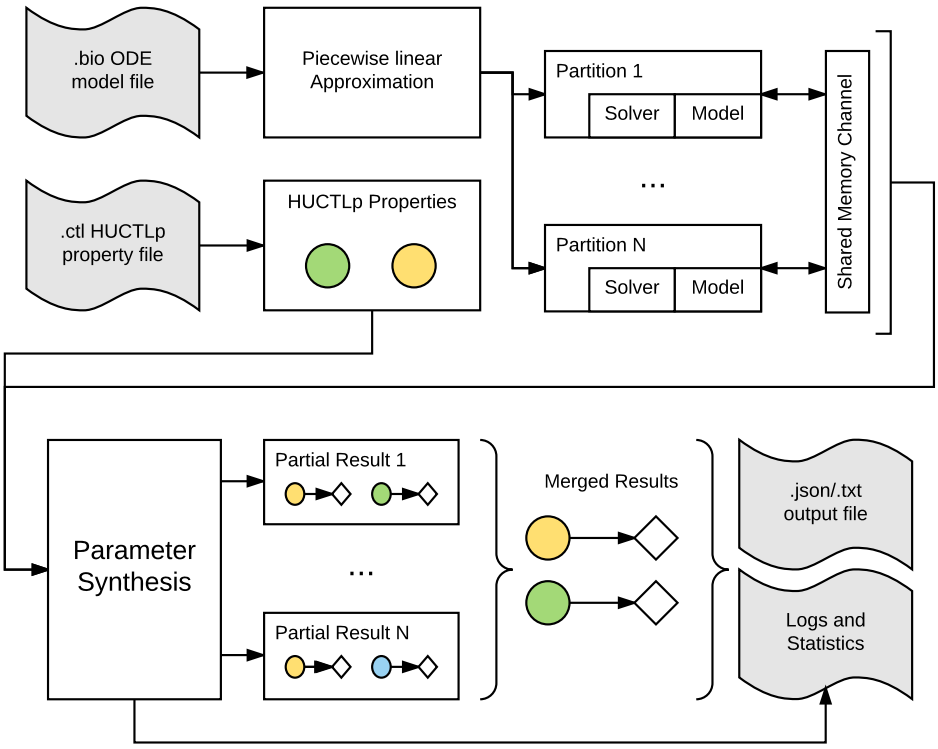


Figure 4.3: Work flow of the command line front-end.

4.4 Command line frontend

The command line front-end connects all these modules into one package with a well defined input and output. Schematic description of this module is given in Figure 4.3.

The process starts by parsing given input files and creating N independent partitions connected using a shared memory communication channel. These data structures are then passed into the parameter synthesis engine.

As soon as the parameter synthesis is finished, the front-end module obtains N partial results from the synthesis engine. These partial results are then merged into one general result set. Finally, the result set can be exported using two supported output formats. First format provides export into a machine readable .json file, suitable for further post-processing or visualisations. Second format provides a more human readable text format, which can be useful when debugging or working with simple models.

The actual content of both formats depends on the type of parameter representation used by the model. When working with hyper-rectangular parameter representation, a parameter constrain is represented using a suitable multi dimensional array interpreted directly as a set of hyper-rectangles. On the other hand, when working with the pure SMT approach, a parameter constrain is represented as a SMT-LIB 2 formula [22].

Finally, a set of diagnostic and benchmarking data is collected during the computation. These include the amount and frequency of data transfers between partitions (to monitor possible communication bottlenecks) and the average solver throughput (corresponds to the complexity of parameter constrains which were encountered during the computation). These data are generally printed directly to the standard output or can be easily redirected into a separate file.

For a detailed description of the command line arguments accepted by the front-end module, see the tool manual provided as a digital appendix.

5 Evaluation

In this chapter, we provide evidence to support the claim that our algorithm scales well with increasing amount of computational resources, and that the algorithm is useful, meaning it can provide interesting, non trivial information about the studied model.

5.1 Models

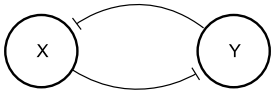
In order to evaluate the algorithm, we first need to define some non-trivial models which we will use to do so. These models, coming from the field of systems biology, are described in this section.

All of the models can be scaled in terms of state and parameter space cardinality by increasing the precision of the piecewise linear approximation.

5.1.1 Bi-stable repressilator

The first model to be considered is the smallest repressilator motif, studied in [23, 24]. It includes two nodes which inhibit each other (see Figure 5.1 (left)). In biology, this motif is very often present in gene regulatory networks, where X represents product of $geneX$ which inhibits production of $geneY$ and vice versa.

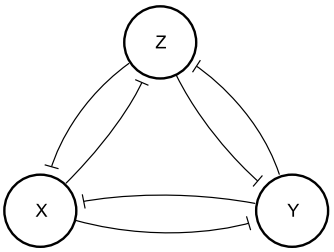
There are several ways to parametrise this model. In this work, we choose ϕ_X and ϕ_Y as our parameters, each in the $(0, 1)$ interval (unless explicitly stated otherwise).



$$\frac{d[X]}{dt} = k_1 \frac{K_1^{n_1}}{K_1^{n_1} + [Y]^{n_1}} - \phi_X[X]$$
$$\frac{d[Y]}{dt} = k_2 \frac{K_2^{n_2}}{K_2^{n_2} + [X]^{n_2}} - \phi_Y[Y]$$

$$k_1 = k_2 = 1, K_1 = K_2 = 5,$$
$$n_1 = n_2 = 5$$

Figure 5.1: Bi-stable repressilator regulatory network (left) and its ODE model taken from [23] (right).



$$\frac{d[X]}{dt} = k_1 \frac{K_{y_1}^{n_1}}{K_{y_1}^{n_1} + [Y]^{n_1}} \cdot \frac{K_{z_1}^{n_2}}{K_{z_1}^{n_2} + [Z]^{n_2}} - \phi_X[X]$$
$$\frac{d[Y]}{dt} = k_2 \frac{K_{x_2}^{n_3}}{K_{x_2}^{n_3} + [X]^{n_3}} \cdot \frac{K_{z_2}^{n_4}}{K_{z_2}^{n_4} + [Z]^{n_4}} - \phi_Y[Y]$$
$$\frac{d[Z]}{dt} = k_3 \frac{K_{x_3}^{n_5}}{K_{x_3}^{n_5} + [X]^{n_5}} \cdot \frac{K_{y_3}^{n_6}}{K_{y_3}^{n_6} + [Y]^{n_6}} - \phi_Z[Z]$$
$$\forall i, j; k_i = 1,$$
$$K_{x_i} = K_{y_i} = K_{z_i} = 5, n_j = 5$$

Figure 5.2: Tri-stable toggle switch regulatory network and its ODE model.

5.1.2 Tri-stable toggle switch

Tri-stable toggle switch is a model of 3-variable repressilator in which each node inhibits not only one but both of its neighbours (see Figure 5.2 (left)). Just one of the two ingoing inhibition is enough to repress any entity. Therefore the ODE model possesses multiplication of negative hill function in entity regulation (Fig. 5.2 (right)).

Similarly to the bi-stable repressilator, we choose ϕ_X, ϕ_Y and ϕ_Z as our parameters, each in the $(0.1, 0.2)$ interval (unless stated otherwise).

5.2 Applicability

To demonstrate the applicability of our approach, we provide an analysis of the stable states of our models, as discovered by the algorithm.

Properties

We study two types of stability motifs. First is a terminal strongly connected component, which can be represented using the HUCTL$_{\mathrm{P}}$ formula $tSCC = \downarrow x : {}^{\triangleright}\mathbf{A}\,\mathbf{G}\,{}^{\triangleright}\mathbf{E}\,\mathbf{F}\,x$. Intuitively, a terminal strongly connected component is a maximal set of states that are pairwise reachable (meaning that from any state, I can reach the whole component, but nothing else).

Second motif is a single terminal cycle, represented using the formula $tCycle = \downarrow x : {}^{\triangleright}\mathbf{A}\,\mathbf{X}\,{}^{\triangleright}\mathbf{A}\,\mathbf{F}$. Intuitively, a terminal cycle is a stronger requirement than a terminal component, since the component can contain multiple interleaving cycles, while the terminal cycle property explicitly specifies that there is exactly one cycle (otherwise the **A** requirement would be broken).

Notice that we would expect each model to have at least one terminal strongly connected component, however, no such requirement can be imposed to terminal cycles.

In order to show that there are at least two distinct instances of the studied patterns (either *tSCC* or *tCycle*) in the model, we use the following property: $biPattern = \exists a \in pattern : pattern \wedge \neg^{\triangleright}\mathbf{E}\,\mathbf{F}\,a$. The property holds in states where the *pattern* is satisfied and there exists other state (also satisfying *pattern*) not reachable from this state. This is implies presence of two pattern instances, since both our patterns are terminal. Also notice the use of the $exists - in$ operator, which guarantees only appropriate z are considered, thus simplifying the property description and computation performance.

Such formula can be further generalised to imply presence of three distinct instances: $triPattern = \exists a \in pattern : \exists b \in pattern : pattern \wedge \neg^{\triangleright}\mathbf{E}\,\mathbf{F}\,a \wedge {}^{\triangleright}\mathbf{E}\,\mathbf{F}\,b \wedge (@a : {}^{\triangleright}\mathbf{E}\,\mathbf{F}\,b)$.

Finally, we can be interested in presence of exactly one instance (or exactly two) of the studied pattern. To this end, we can simply use the property $single = pattern \wedge \neg biPattern \wedge \neg triPattern$.

Bi-stable repressilator

The results of our analysis of the bi-stable repressilator are presented in Figures 5.3 and 5.4. Each figure contains a state space plot and a parameter space plot, where green colour signifies the presence of exactly one pattern instance and the yellow colour signifies presence of exactly two pattern instances. Furthermore, in the state space plot, the mixture of green and yellow represents that either one or two instances of the studied pattern are present, depending on the parameter valuation.

The results of our terminal component analysis are presented in Figure 5.3. As expected, the model contains either one or two terminal components, depending on the parameter valuation. Furthermore,

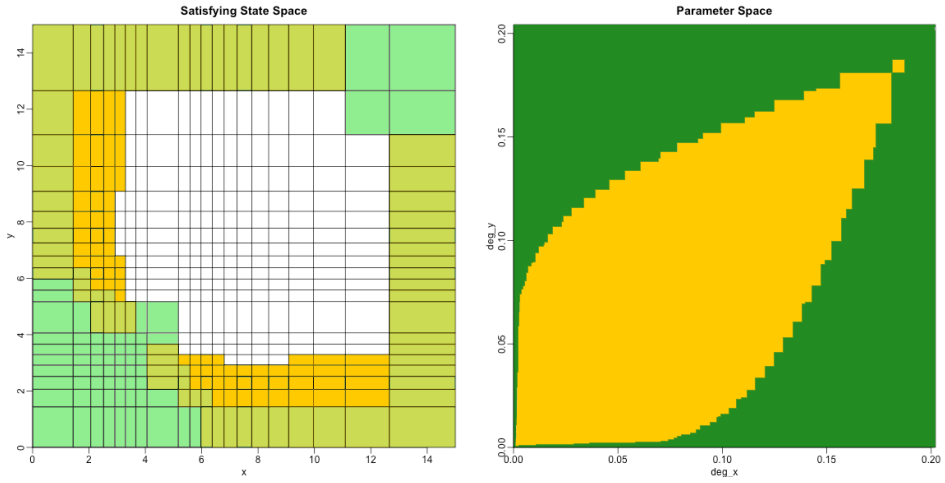


Figure 5.3: Presence of two terminal strongly connected components in the bi-stable repressilator model. The parameter space (right) has been zoomed to cover only the interesting area.

the location of these components is clearly visible in the state space plot (right).

The analysis of the terminal cycles is presented in Figure 5.4. As we can see, the model contains parameter valuations for which no terminal cycle is present. For these valuations a manual inspection revealed that multiple non-terminal cycles are present in the model.

Tri-stable toggle switch

The results of our analysis of the tri-stable toggle switch are presented in Figures 5.5 and 5.6. The presentation of these results is affected by the dimensionality of the model in question. Mainly, we present the dependence of just two variables (parameters) and project the remaining values into such plane. This makes the colour-mixing approach taken in the previous case infeasible, since there is no way to distinguish truly intersecting properties from the projected ones. Therefore we assume the reader is more interested in higher counts of pattern instances and prioritise those. All visualisations are produced directly using Pithya and the full results are available as a digital attachment

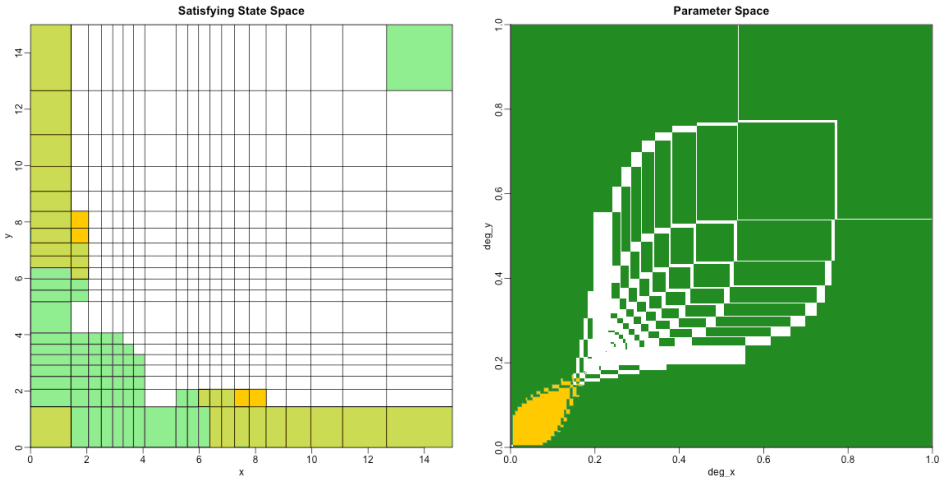


Figure 5.4: Presence of two terminal cycles in the bi-stable repressilator model.

to this work. The reader is therefore encouraged to inspect the data directly in case of any further questions.

As we can see in Figure 5.5, the model can contain either one, two, or three terminal strongly connected components, depending on the parameter valuations. The state space covered by the components appears to be continuous due to the dimensional projection. In fact, the three components are located in the corners of the projected triangle, while the triangle itself spans all three dimensions. Depending on the parameter valuation, the triangle can contract to produce two, or just one component.

Figure 5.6 then presents similar results, but as we can see, the stronger requirement of terminal cycles produces smaller result set. Interesting observation is that for the selected parametrisations, at least one terminal cycle is always present (This is not directly implied by the plots, since they are projections. It has been verified separately).

5.3 Scalability

We evaluate the scalability of the algorithm using two CTL and two HUCTL$_P$ properties:

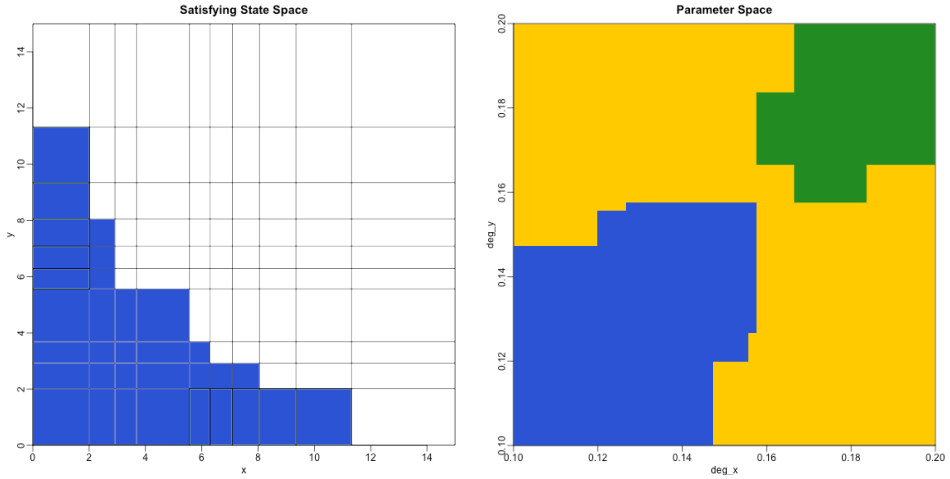


Figure 5.5: Presence of one (green), two (yellow) and three (blue) terminal strongly connected components in the tri-stable toggle switch model. Remaining dimensions have been projected into the plots.

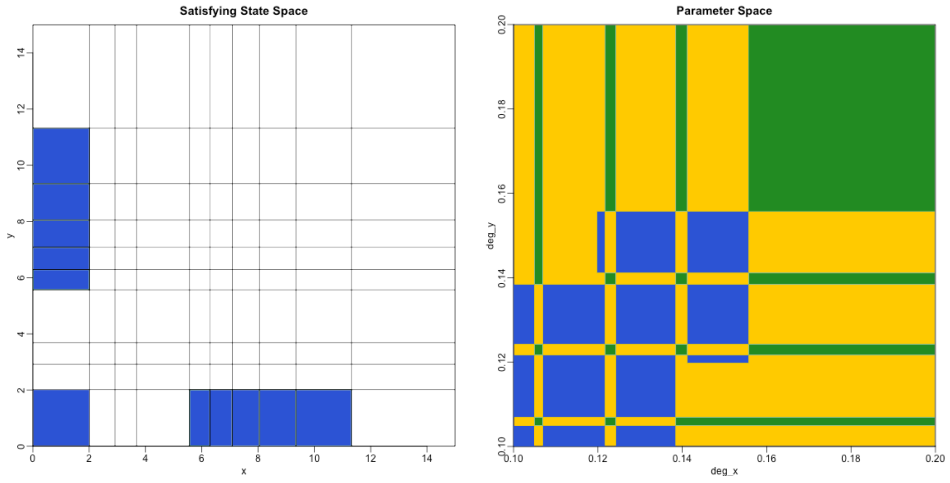


Figure 5.6: Presence of one (green), two (yellow) and three (blue) terminal cycles in the tri-stable toggle switch model. Remaining dimensions have been projected into the plots.

Table 5.1: Scalability results

	Bi-stable repressilator				Tri-stable toggle switch			
State count	$\sim 2.25e4$		~ 1000		$\sim 3.4e5$		~ 3000	
Property	φ_1	φ_2	φ_3	φ_4	φ_1	φ_2	φ_3	φ_4
1cpu/4gb	112s	35s	259s	89s	63s	62s	125s	30s
2cpu/8gb	76s	31s	168s	71s	45s	42s	86s	20s
4cpu/16gb	65s	26s	110s	44s	35s	36s	57s	17s
8cpu/32gb	38s	26s	65s	40s	30s	28s	39s	14s

$$\varphi_1 = {}^{\triangleright}\,\mathbf{E}\,\mathbf{F}\,\mathit{center}$$
$$\varphi_2 = {}^{\triangleright}\,\mathbf{A}\,\mathbf{F}\,\mathit{center}$$
$$\varphi_3 = \downarrow x : {}^{\triangleright}\,\mathbf{E}\,\mathbf{X}\,{}^{\triangleright}\,\mathbf{E}\,\mathbf{F}\,x$$
$$\varphi_4 = \downarrow x : {}^{\triangleright}\,\mathbf{A}\,\mathbf{X}\,{}^{\triangleright}\,\mathbf{A}\,\mathbf{F}\,x$$

Here, *center* specifies a proposition which is satisfied only in the very middle state of the model.

Each property has been tested on a model with an appropriate state space size (since HUCTL$_{\mathrm{P}}$ properties are usually much harder to verify). The results of this analysis are presented in Table 5.1. As we can see, the algorithm scales with increasing amount of computational resources, the only problem being the **A F** query on the two dimensional model. Further inspection revealed that the algorithm is not able to parallelise this query very well, because the valid state space does not provide much opportunities to branch the exploration into multiple parallel directions.

The evaluation has been performed on a 64-core server and is taken as an overage over five runs. However, the access to this server was not exclusive. The computation was restricted to the specified amount of processors and RAM during each experiment.

6 Conclusion

In this work, we presented an efficient distributed fixed point algorithm for solving the parameter synthesis problem for parametrised direction transition system (PDTS) with properties specified using the hybrid computation tree logic with past (HUCTL_P). The algorithm works in a semi-symbolic manner, with explicit state space and symbolic parameter space representations, relying on an appropriate solver for deciding and simplifying the parameter sets.

HUCTL_P is a more expressive extension of CTL, which allows specification of various interesting properties, such as strongly connected components, cycles or directed runs. We provide a detailed discussion of its semantics and its relationship with CTL.

We also provide an implementation of the above mentioned algorithm, which is optimised for multi-core usage. The implementation is freely available as part of the Pithya parameter synthesis tool. As a modelling framework, the implementation provides a module for working with ordinary differential equation based models. However, the core algorithm is completely model agnostic. We also provide a bridge to the Microsoft Z3 solver and various domain-specific optimised solvers.

For this implementation, we provide a case study which explores the terminal components and terminal cycles of two well know models from systems biology. We also provide a scalability analysis which shows that the algorithm is able to utilise provided computational resources.

As future work, we would like to extend the implementation with distributed computation capabilities, since the main framework is already prepared for this, only an appropriate `Communicator` is needed. Other possible research direction would be to design a more general, fixed point computation framework, which can then be used to implement other common algorithms, such as more efficient component detection. One can also consider a cloud oriented approach to the current fixed point algorithm, relying on stream processing. Finally, the implementation would greatly benefit from more domain specific solvers and on-the-fly compilation of models, which would speed up the parameter set related operations and state space generation.

Bibliography

1. SMALE, Steve. Differentiable Dynamical Systems. In: *The Mathematics of Time: Essays on Dynamical Systems, Economic Processes, and Related Topics*. New York, NY: Springer New York, 1980, pp. 1–82. ISBN 978-1-4613-8101-3. Available from DOI: 10.1007/978-1-4613-8101-3_1.
2. ABRAHAM, R.; MARSDEN, J.E. *Foundations of Mechanics*. AMS Chelsea Pub./American Mathematical Society, 1978. AMS Chelsea publishing. ISBN 9780821844380. Available also from: https://books.google.cz/books?id=4Y-ownk6ilsC.
3. ARECES, Carlos; CATE, Balder ten. 14 Hybrid logics. *Studies in Logic and Practical Reasoning*. 2007, vol. 3, pp. 821–868. ISSN 1570-2464. Available from DOI: http://dx.doi.org/10.1016/S1570-2464(07)80017-6.
4. BEEK, Maurice H. ter; FANTECHI, Alessandro; GNESI, Stefania; MAZZANTI, Franco. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*. 2011, vol. 76, no. 2, pp. 119–135. ISSN 0167-6423. Available from DOI: http://dx.doi.org/10.1016/j.scico.2010.07.002.
5. THIAGARAJAN, PS. A trace based extension of linear time temporal logic. In: *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on*. 1994, pp. 438–447.
6. BENEŠ, Nikola; BRIM, Luboš; DEMKO, Martin; PASTVA, Samuel; ŠAFRÁNEK, David. A Model Checking Approach to Discrete Bifurcation Analysis. In: FITZGERALD, John; HEITMEYER, Constance; GNESI, Stefania; PHILIPPOU, Anna (eds.). *FM 2016*. Springer, 2016, vol. 9995, pp. 85–101. LNCS. ISBN 978-3-319-48988-9. Available from DOI: 10.1007/978-3-319-48989-6.
7. CLARKE Jr., Edmund M.; GRUMBERG, Orna; PELED, Doron A. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.

8. BATT, G.; PAGE, M.; CANTONE, I.; GÖSSLER, G.; MONTEIRO, P.T.; JONG, H. de. Efficient parameter search for qualitative models of regulatory networks using symbolic model checking. *Bioinformatics*. 2010, vol. 26, no. 18, pp. 603–610.

9. DONALDSON, Robin; GILBERT, David. A Model Checking Approach to the Parameter Estimation of Biochemical Pathways. In: *CMSB*. Springer, 2008, vol. 5307, pp. 269–287. LNCS.

10. DONZÉ, Alexandre; CLERMONT, Gilles; LANGMEAD, Christopher J. Parameter synthesis in nonlinear dynamical systems: Application to systems biology. *Journal of Computational Biology*. 2010, vol. 17, no. 3, pp. 325–336.

11. JHA, Sumit Kumar; LANGMEAD, Christopher James. Synthesis and infeasibility analysis for stochastic models of biochemical systems using statistical model checking and abstraction refinement. *Theoretical Computer Science*. 2011, vol. 412, no. 21, pp. 2162–2187.

12. BARNAT, Jiri; BRIM, Lubos; KREJCI, Adam; STRECK, Adam; SAFRANEK, David; VEJNAR, Martin; VEJPUSTEK, Tomas. On Parameter Synthesis by Parallel Model Checking. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*. 2012, vol. 9, no. 3, pp. 693–705. ISSN 1545-5963. Available from DOI: 10.1109/TCBB.2011.110.

13. BENEŠ, Nikola; BRIM, Luboš; DEMKO, Martin; PASTVA, Samuel; ŠAFRÁNEK. Parallel SMT-Based Parameter Synthesis with Application to Piecewise Multi-Affine Systems. In: *ATVA'16*. Springer, 2016. LNCS. To appear.

14. TARSKI, Alfred. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 1955, vol. 5, no. 2, pp. 285–309. Available also from: http://projecteuclid.org/euclid.pjm/1103044538.

15. MATTERN, Friedemann. Algorithms for distributed termination detection. *Distributed Computing*. 1987, vol. 2, no. 3, pp. 161–175. ISSN 1432-0452. Available from DOI: 10.1007/BF01782776.

16. TANENBAUM, Andrew S; VAN STEEN, Maarten. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

17. SYBILA, Laboratory. *Pithya* [online]. 2017 [visited on 2017-05-17]. Available from: http://biodivine.fi.muni.cz/pithya.

18. MOURA, Leonardo de; BJØRNER, Nikolaj. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by RAMAKRISHNAN, C. R.; REHOF, Jakob. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN 978-3-540-78800-3. Available from DOI: 10.1007/978-3-540-78800-3_24.

19. PARR, Terence. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN 1934356999, 9781934356999.

20. RIZK, Aurélien; BATT, Gregory; FAGES, François; SOLIMAN, Sylvain. A general computational method for robustness analysis with applications to synthetic gene networks. *Bioinformatics*. 2009, vol. 25, no. 12, pp. i169. Available from DOI: 10.1093/bioinformatics/btp200.

21. COLLINS, Pieter J.; HABETS, Luc; SCHUPPEN, Jan H. van; ČERNÁ, Ivana; FABRIKOVÁ, Jana; ŠAFRÁNEK, David. Abstraction of Biochemical Reaction Systems on Polytopes. *IFAC Proceedings Volumes*. 2011, vol. 44, no. 1, pp. 14869–14875. ISSN 1474-6670. Available from DOI: http://dx.doi.org/10.3182/20110828-6-IT-1002.03317.

22. BARRETT, Clark; FONTAINE, Pascal; TINELLI, Cesare. *The Satisfiability Modulo Theories Library (SMT-LIB)* [www.SMT-LIB.org]. 2016.

23. BRIM, Luboš; DEMKO, Martin; PASTVA, Samuel; ŠAFRÁNEK, David. High-Performance Discrete Bifurcation Analysis for Piecewise-Affine Dynamical Systems. In: *Hybrid Systems Biology*. Springer, 2015, pp. 58–74.

24. DILÃO, Rui. The regulation of gene expression in eukaryotes: bistability and oscillations in repressilator models. *Journal of theoretical biology*. 2014, vol. 340, pp. 199–208.

A List of electronic attachments

- Current source code of the Pithya tool implementing the algorithm.
- Pithya manual with description of the input/output formats.
- Raw case study results.