MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Parameter Synthesis from Hypotheses Formulable in CTL Logic

BACHELOR THESIS

Samuel Pastva

Brno, Spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by 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

I would like to thank my supervisor . . .

Abstract

The main focus if this thesis is to provide a distributed algorithm for computation of the parameter synthesis problem for biochemical systems and CTL hypothesis and a usable implementation of said algorithm. The soundness and scalability of the algorithm is successfully demonstrated on biochemical models based on ordinary differential equations.

Keywords

Coloured Model Checking, CTL, Parameter Sythesis Problem, Systems Biology, Distribution

Contents

1 Introduction

2 Terms and Definitions

2.1 Kripke Structure

As an input of our algorithm, we expect a parametrized Kripke Structure as defined in [CITE]. Parametrized Kripke Structure is a tuple $\mathcal{K} = (\mathcal{P}, S, S_0, \rightarrow, L)$ where

- $\mathcal{P}$ is a finite set of parameters (all possible parameter valuations)
- S is a finite set of states
- $S_0 \subseteq S$ is a set of initial states
- $\rightarrow \subseteq S \times \mathcal{P} \times S$ is a total transition relation labeled by parameter valuations
- $L : S \rightarrow 2^{AP}$ is a labeling function from states to sets of atomic propositions which are true in such states

We write $s \xrightarrow{p} s'$ when $(s, p, s') \in \rightarrow$. We also write $s \rightarrow s'$ when $\exists p \in \mathcal{P}.(s, p, s') \in \rightarrow$. Note that fixing a valuation $p \in \mathcal{P}$ reduces the Parametrized Kripke Structure $\mathcal{K}$ to concrete, non-parametrized Kripke Structure $\mathcal{K}(p) = (S, S_0, \xrightarrow{p}, L)$.

2.2 CTL Logic

In order to correctly express various hypotheses in systems biology, the idea of branching time is need. Examples of such hypotheses are given in section [Case Study]. Therefore, this work uses the Computation Tree Logic(CTL) as means of hypotheses formulation.

CTL syntax is defined inductively upon finite set of atomic propositions:

$$\varphi ::= true \mid false \mid Q \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{EX}\varphi_1 \mid \mathrm{E}\varphi_1\mathrm{U}\varphi_2 \mid \mathrm{A}\varphi_1\mathrm{U}\varphi_2 \tag{2.1}$$

Sometimes, we will use parentheses to make bigger formulas easily readable, but they will in no way be used to modify the meaning of formula or priority of operators.

Note that there are also other temporal operators in standard CTL definition. We do not implement those directly in our algorithm. However, we use following equations to transform any general CTL formula prior to computation, so that it only uses operators supported in our algorithm. This way we can achieve a concise algorithm and also support whole CTL logic.

- $\mathrm{AX}\varphi = \neg\mathrm{EX}\neg\varphi$
- $\mathrm{EF}\varphi = \mathrm{E}(true)\mathrm{U}\varphi$
- $\mathrm{EG}\varphi = \neg\mathrm{A}(true)\mathrm{U}\neg\varphi$
- $\mathrm{AF}\varphi = \mathrm{A}(true)\mathrm{U}\varphi$
- $\mathrm{AG}\varphi = \neg\mathrm{E}(true)\mathrm{U}\neg\varphi$

Note that all of these transformations also preserve number of temporal operators in a formula.

Other boolean operators like implication or equivalence can also be derived using similar transformations.

Let φ be a CTL formula. We write $cl(\varphi)$ to denote the set of all sub-formulas of φ and $tcl(\varphi)$ to denote the set of all temporal sub-formulas of φ. By $|\varphi|$ we denote the size of formula φ.

We assume standard CTL semantics over non-parametrized Kripke structures as defined in [cite].

2.3 Parameter Synthesis Problem

Parameter synthesis problem is defined in following way. Suppose we are given a parametrised Kripke structure $\mathcal{K} = (\mathcal{P}, S, S_0, \rightarrow, L)$ and a CTL formula φ. For each state $s \in S$ let $P_s = \{p \in \mathcal{P} \mid s \models_{\mathcal{K}(p)} \varphi\}$,where $s \models_{\mathcal{K}(p)} \varphi$ denotes, that φ is satisfied in the state s of $\mathcal{K}(p)$. The parameter synthesis problem requires to compute the function $\mathcal{M}^{\mathcal{K}}_{\varphi} : S \rightarrow 2^{\mathcal{P}}$ such that $\mathcal{M}^{\mathcal{K}}_{\varphi}(s) = P_s$. Often we are especially interested in computing the set of all parameters for which the property holds in some initial states $\bigcap_{s \in S_0} \mathcal{M}^{\mathcal{K}}_{\varphi}(s)$. We will sometimes omit the φ and $\mathcal{K}$ when they are clear from the context.

2.4 CTL logic and model approximation

In model checking, some modeling approaches suffer from over or under approximation. We say that model is over-approximated when all feasible transitions are contained in the model, but it can also contain transitions that are not feasible in the situation the model is describing. Symmetrically, we say that model is under-approximated when all transitions in the model are feasible in the modeled situation, but not all feasible transitions has to be contained in the model.

It is important to discuss this relationship between CTL and approximated models, because it is much more complicated compared to linear-time logic since CTL allows for universal and existential quantification mixing.

We say that CTL formula is *universal* or that it belongs to ACTL when it only contains universal temporal operators and no negation. Symmetrically, we say that CTL formula is *existential* or that it belongs to ECTL when it only contains existential temporal operators and no negation.

Observe that the truth of ACTL properties is preserved in over-approximated models. In other words, if an ACTL property holds in an over-approximated model, it must also hold in the original model. However, their falsity cannot be guaranteed, because the false transitions may introduce paths that falsify the property in states where it would be normally true. Similarly, the falsity of ECTL properties in over-approximated models is preserved but the truth is not. In this case, the existence of false transitions can introduce states where ECTL property holds solely due to these false paths.

Symmetrically, for under-approximated models, the falsity of ACTL and the truth of ECTL is preserved. But due to similar arguments, we can't say anything about their counterparts.

If we allow full CTL, in general, we can't make any assumptions about results obtained from either under- or over-approximated systems. This is caused by mixing of existential and universal quantification which leads to results which may be spurious and incomplete at the same time. Therefore, no conclusions can be made without further investigation and validation of such results.

2.5 Kripke Fragments

Due to the state space explosion, given parametrised Kripke structure can be very large and therefore impossible to fit into memory of one computer. In order to solve parameter synthesis problem for such structures, we have to distribute the state space across several computational nodes. To this end, we introduce the notion of parametrised kripke fragments.

A parametrised Kripke structure $\mathcal{K}$ can be divided into several parametrised Kripke fragments $\mathcal{F}_1^{\mathcal{K}}, \mathcal{F}_2^{\mathcal{K}}, \cdots, \mathcal{F}_N^{\mathcal{K}}$ using a partition function f. Parametrised Kripke fragment with identifier i over Kripke structure $\mathcal{K} = (\mathcal{P}, S, S_0, \rightarrow, L)$ is then defined as a tuple $\mathcal{F}_i^{\mathcal{K}} = (f, \mathcal{P}, S_i, I_i, \rightarrow_i, L_i)$ where:

- $f : S \rightarrow \{1, \cdots, N\}$ is a partition function from all states to fragment identifiers
- $\mathcal{P}$ is a finite set of all parameters
- $S_i = \{s \in S \mid f(s) = i \vee \exists s' \in S.((s \rightarrow s' \vee s' \rightarrow s) \wedge f(s') = i\}$ is a subset of original state space which belongs to this fragment
- $I_i = \{s \in S_0 \mid s \in S_i\}$ is a set of all fragments initial states
- $\rightarrow_i = \{(s, p, s') \in \rightarrow \mid s \in S_i \wedge s' \in S_i \wedge (f(s) = i \vee f(s') = i)\}$ is a subset of original transition function reduced to only relevant states (not required to be total)
- $L_i = \{(s, l) \in L \mid f(s) = i\}$ is a labeling subset of original labeling function relevant to this fragment

In the following text, we will often omit the superscript $\mathcal{K}$ if it is clear from context.

Intuitively, Kripke fragment represents a subset of the original kripke structure defined by the partition function. It contains all nodes specified by the partition function with all their direct successors and predecessors and all corresponding transitions.

We define a set of border states $border(\mathcal{F}_i^{\mathcal{K}}) = \{s \in S_i \mid f(s) \neq i\}$. Intuitively, these states represent the remaining portion of the state space which is stored in memory of other processes and is not directly accessible. We say that state is an *internal state* if it is not a border state. We also define a set of cross edges as $cross(\mathcal{F}_i^{\mathcal{K}}) = \{(s, p, s') \in \rightarrow_i \mid s' \in border(\mathcal{F}_i^{\mathcal{K}}) \vee s \in border(\mathcal{F}_i^{\mathcal{K}})\}$. Intuitively, these are edges leading from border states to internal states or vice versa.

Note that for a constant partition function $f(s) = 1$ and any given kripke structure, the partitioning results in one fragment with an unchanged state space and an unchanged transition relation (The sets of border states and cross edges are empty for the resulting fragment). Similarly, for every Kripke structure, there exists a partition function for which $N = |S|$ and every resulting fragment contains only one internal state. Under such partitioning, all edges are cross edges.

In worst case (connected graphs), such partitioning results in fragments with $|S| - 1$ border states and one internal state (there is no way to achieve higher state count in fragment than in the original structure). Therefore we can assume that $\sum\limits_{i=1}^{N} |S_i| \leq |S|^2$, hence the number of introduced border states is at worst quadratic in terms of original state space. The number of internal states remains the same.

This increase seems rather high, however, it is important to note that this also requires one process for each original state. In real life scenarios, the number of states per process is usually much higher. Also note that the representation of border state in memory is usually much simpler (and smaller) than representation of internal state, so even a distribution with high border state count can be beneficial in terms of memory consumption.

In terms of edges, in the worst case, each edge has to be contained in two fragments (where either of the end states is an internal state), therefore the total number of edges is at worst doubled.

The number of border states and cross edges is also highly dependent on the partition function. It is usually best to design the partition function with specific model or modeling approach in mind in order to achieve optimal workload distribution. We will discuss different partition functions later in the text.

For each kripke fragment $\mathcal{F}_i$, we define a relation *successors* $\subseteq S_i \times 2^{\mathcal{P}} \times S_i$ to denote the set of colors for which there exist a transition from the first to the second state.

$$successors = \{(s, P, s') \mid P = \{p \in \mathcal{P} \mid (s, p, s') \in \to_i\}\}$$

2.6 Assumption Semantics

Classic interpretation of CTL formulas is not adequate for Kripke fragments. In order to accommodate for possible non-totality and distributed nature of the Kripke fragments over Kripke structure $\mathcal{K} = (\mathcal{P}, S, S_0, \to, L)$, we introduce the assumption function $\mathcal{A} : \mathcal{P} \times S \times cl(\varphi) \to Bool$. The values $\mathcal{A}(s, p, \varphi_1)$ are called assumptions. We use the notation $\mathcal{A}(p, s, \varphi_1) = \perp$ to say that the value of $\mathcal{A}(s, p, \varphi_1)$ is undefined. By $\mathcal{A}_\perp$ we denote assumption function which is undefined for all inputs.

Intuitively, $\mathcal{A}(s, p, \varphi_1) = \texttt{tt}$ when we can assume that φ_1 holds in state s for parameter valuation p, $\mathcal{A}(s, p, \varphi_1) = \texttt{ff}$ when we can assume that φ_1 does not hold in state s for parameter valuation p and $\mathcal{A}(s, p, \varphi_1) = \perp$ when we cannot assume anything about validity of φ_1 in state s for parameter valuation p.

Undefined values are important in CTL semantics over distributed fragments, since such values can be used in places where validity of formula cannot be computed because it depends on information stored in another fragment which has not been received yet. However, a correct model checking algorithm should always provide a definitive answer for all states and parameter valuations.

For a Kripke fragment $\mathcal{F}_i = (f, \mathcal{P}, S_i, I_i, \to_i, L_i)$ and a formula φ, the assumption function is defined inductively on the structure of the formula φ:

$$\mathcal{A}(p, s, Q) = \begin{cases} \texttt{tt} & Q \in L(s) \\ \texttt{ff} & \text{otherwise} \end{cases}$$

$$\mathcal{A}(p,s,\neg\varphi_1) = \begin{cases} \texttt{tt} & \mathcal{A}(p,s,\varphi_1) = \texttt{ff} \\ \texttt{ff} & \mathcal{A}(p,s\varphi_1) = \texttt{tt} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{A}(p,s,\varphi_1 \wedge \varphi_2) = \begin{cases} \texttt{tt} & \mathcal{A}(p,s,\varphi_1) = \texttt{tt} \text{ and } \mathcal{A}(p,s,\varphi_2) = \texttt{tt} \\ \texttt{ff} & \mathcal{A}(p,s\varphi_1) = \texttt{ff} \text{ or } \mathcal{A}(p,s,\varphi_2) = \texttt{ff} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{A}(p,s,\mathrm{EX}\varphi_1) = \begin{cases} \texttt{tt} & \exists s' \in S : s \xrightarrow{p} s' \wedge \mathcal{A}(p,s',\varphi_1) = \texttt{tt} \\ \texttt{ff} & \forall s' \in S : s \xrightarrow{p} s' \Rightarrow \mathcal{A}(p,s',\varphi_1) = \texttt{ff} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{A}(p,s,\mathrm{E}\varphi_1\mathrm{U}\varphi_2) = \begin{cases} \texttt{tt} & \text{exits a p-path } \pi = s_0s_1s_2\ldots \text{ with } s = s_0 \text{ such that } \exists x < |\pi| \text{ such that (either } \mathcal{A}(p,s_x,\varphi_2) = \texttt{tt} \text{ or } [s_x \in \mathit{border}(\mathcal{K}) \text{ and } \mathcal{A}(p,s_x,\mathrm{E}\varphi_1\mathrm{U}\varphi_2) = \texttt{tt}]\text{), and } \forall 0 \leq y < x : \mathcal{A}(p,s_y,\varphi_1) = \texttt{tt} \\ \texttt{ff} & \text{for all p-paths } \pi = s_0s_1s_2\ldots \text{ with } s = s_0 \text{ either } \exists x < |\pi| \text{ such that } (\mathcal{A}(p,s_x,\varphi_1) = \texttt{ff} \text{ and } \forall y \leq x : \mathcal{A}(p,s_y,\varphi_2) = \texttt{ff}) \text{ or } [\forall x < |\pi| : \mathcal{A}(p,s_x,\varphi_2) = \texttt{ff} \text{ and } (|\pi| = \infty \text{ or } (s_{|\pi|-1} \in \mathit{border}(\mathcal{K}) \text{ and } \mathcal{A}(p,s_{|\pi|-1},\mathrm{E}\varphi_1\mathrm{U}\varphi_2) = \texttt{ff}))] \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{A}(p,s,\mathrm{A}\varphi_1\mathrm{U}\varphi_2) = \begin{cases} \texttt{tt} & \text{for all p-path } \pi = s_0s_1s_2\ldots \text{ with } s = s_0 \text{ such that } \exists x < |\pi| \text{ such that [either } \mathcal{A}(p,s_x,\varphi_2) = \texttt{tt} \text{ or } (s_x \in \mathit{border}(\mathcal{K}) \text{ and } \mathcal{A}(p,s_x,\mathrm{A}\varphi_1\mathrm{U}\varphi_2) = \texttt{tt})]\text{, and } \forall 0 \leq y < x : \mathcal{A}(p,s_y,\varphi_1) = \texttt{tt} \\ \texttt{ff} & \text{exists a p-path } \pi = s_0s_1s_2\ldots \text{ with } s = s_0 \text{ and an index } x < |\pi| \text{ such that: } (\mathcal{A}(p,s_x,\varphi_1) = \texttt{ff} \text{ and } \forall y \leq x : \mathcal{A}(p,s_y,\varphi_2) = \texttt{ff}) \text{ or } [\forall x < |\pi| : \mathcal{A}(p,s_x,\varphi_2) = \texttt{ff} \text{ and } (|\pi| = \infty \text{ or } (s_{|\pi|-1} \in \mathit{border}(\mathcal{K}) \text{ and } \mathcal{A}(p,s_{|\pi|-1},\mathrm{E}\varphi_1\mathrm{U}\varphi_2) = \texttt{ff}))] \\ \bot & \text{otherwise} \end{cases}$$

2.7 Modelling techniques

To prove soundness and universlaity of our algorithm, we test it on two different modelling techniques.

First techinique is based on discretization of piece-wise multi-affine models defined using Ordinary Differential Equations. More about this techinique can be found in [cite]. ODE models have rectangular state space with low number of transitions per state and good locality. Since parameters valuations are continuous, they have to be symbolically represented in form of intervals. Unfortunately, ODE models suffer from over-approximation and therefore are not very well suited for verification of properties with mixed existential and universal quantification.

Second technique is based on Thomas Networks which are extension of Boolean Networks. Compared to ODE models, Thomas Networks are inherently discrete. They also do not suffer from over- or under-approximation. The downside of Thomas Networks are parameters, since even small models can contain a very large number of parameters and possible parameter valuations. Usually, it is also hard to find a compact and reasonably fast symbolic representation of said parameters. The influence of parameters on transition system also tends to by much more irregular compared to ODE models. This makes them less suitable for state space distribution.

2.8 State Space Partitioning

One of the most important aspects of efficient distributed model checking algorithms is a suitable state space partitioning. One that minimizes the communication overhead and provides an equalized workload distribution. In particular, the partitioning should provide a regular load-balancing, ensuring that each process is responsible for a proportional part of the state space. It should also provide a good locality, minimizing the number of cross transitions and therefore reducing the communication overhead.

The problem of computing the optimal partitioning can introduce a significant overhead into the computation. Therfore, various heuristics are considered to produce partitioning that is easy to compute and at the same time provides reasonable load-balancing and locality. One of such heuristics is a hash based partitioning which is usually used for computer and engineering systems. This partitioining does not require any prior knowledge about the structure of the state space, and therefore can be universally applied to almost any system.

It is based on a hash function mapping each state to a process. This approach usually results in in very good load balancing thanks to the uniformity of the hash function. However, hash based partitioning can't control the locality and therefore may introduce a high number of cross transitions into the system. This can significantly increase communication overhead.

In this work, we exploit the regular structure of the state space of biochemical models [cite] in order to produce partitioning that does not suffer from this kind of locality problem. We use structural properties of the rectangular abstraction of the given parametric piecewise multi-affine ODE model [cite]. The approximation is formed by an n-dimensional hyper-rectangular state space defined by n state variables and by a set of thresholds for each variable. The discretization of the state space also ensures that there are only transition between adjacent states with respect to the hyper-rectangular structure.

Our partitioning decomposes said state space into p hyper-rectangular subspaces such that all subspaces have similar state count (p is the number of processes). This heuristic usually proveds good load balancing, since the states are evenly distributed across all processes. However, compared to classic hash based partitioning, it can also provide better locality thanks to the fact that transitions only occur between adjacent states. This ensures that cross transitions can originate only in states on borders of these hyper-rectangular subspaces. Which in turn provides almost minimal number of cross transitions and therefore significantly reduces the communication overhead.

However, note that this is still only a heuristic and the results can be negatively influenced by the backward connectivity of the state space. Especially in systems with uneven distribution of transitions across states, better partitionings can be constructured that minimize the number of cross transitions while at the same time maintaining good load-balancing. On the other hand, our experiments demostrate that due to the fact, that we have to consider all possible parametrizations, the connectivity of the state space is significantly increased. Therefore, our heuristic can for majority of models create a partitioning that reduces the communication overhead significantly.

Similar heuristic can be deployed also for the Thomas networks, however, this modeling thechnique has a less predictable transition system and therefore the minimal number of cross transitions is not that easily achieved.

3 Algorithm

In this chapter, we describe the distributed algorithm that computes the assumption function $\mathcal{A}$.

3.1 Distributed Environment

In this section, we briefly describe the distributed environment assumed by our algorithm, in order to prevent any possible confusion.

We assume a distributed environment with fixed number of reliable processes connected by reliable, order-preserving channels (The order preservation can be relaxed to some extent). We also assume that each process has a fixed identifier and the set of all process identifiers is equal to the result set of the partition function. Each process can communicate directly (using the function SEND) with any other process (assuming it knows other process's identifier). We assume that each message can be transmitted in $O(1)$ time and all messages that can't be processed directly are stored in a queue until they can be processed.

Several parts of the algorithm do not have explicit termination (they terminate by reaching deadlock - no messages are exchanged between processes). In such cases, suitable termination detection algorithm is employed to detect this deadlock and terminate computation properly. Our implementation uses Safra's algorithm [CITATION] for this purpose, but the related code has been skipped for easier readability.

The algorithm is broken into two main parts. First describes the general outline of algorithm and is similar to classic CTL model checking. Second part describes how each of the supported temporal operators is processed. This part contains more detailed description of inter-process communication and operator specific data structures. To better reflect the distributed nature of the algorithm, description of each temporal operator routine is divided into three parts: Process variables, Initialization and Message handler. First section describes data structures stored in process memory available during whole computation. Initialization section is executed exactly once and no messages can be received until it's finished. Message handler defines what should happen when message is received.

3.2 Algorithm outline

The main idea of the algorithm is described in REFERENCE and resembles other CTL model checking algorithms.

```
1: procedure CHECKCTL($\phi, \mathcal{K} = (\mathcal{P}, S, S_0, \rightarrow, L)$)
2:     $\mathcal{A} \leftarrow \{(p, s, \alpha, \mathtt{tt}) \mid p \in \mathcal{P} \wedge \alpha \in L(s)\}$
3:     for all $i < |\phi|$ do
4:         for all $\psi$ in $cl(\phi)$ where $|\psi| = i$ do
5:             $\mathcal{A} \leftarrow$ CHECKFORMULA($\psi, \mathcal{K}, \mathcal{A}$)
6:         end for
7:     end for
8: end procedure
```

The algorithm starts by initializing the assumption function using the labeling function defined in kripke fragment. After that, it traverses the structure of formula, starting from smallest formulas and uses computed results to process more complex formulas. Function CHECKFORMULA computes all states and colors where formula ψ holds and returns assumption function updated accordingly. This is done using the local information contained in given kripke fragment, assumptions previously computed for smaller formulas and also by communicating with other processes. Note that only assumptions relevant for particular process are computed and returned (each process has information only about it's own state space).

3.3 Common operations

In this section, we define functions used to simplify the algorithm description. Let us fix a formula ϕ and a parametrised kripke fragment $\mathcal{K} = (\mathcal{P}, S, S_0, \rightarrow, L)$ as an input of the algorithm.

Intuitively, function $validStates : cl(\phi) \times AS^{\phi}_{\mathcal{K}} \rightarrow S \times 2^{\mathcal{P}}$ computes a set of states and parameters where truth of given formula is assumed. It is also responsible for handling of boolean operators, since these can be computed without any inter-process communication.

$$validStates(\phi_1 \wedge \phi_2, \mathcal{A}) = \{(s,p) \mid \mathcal{A}(s,p,\phi_2) = \texttt{tt} \wedge \mathcal{A}(s,p,\phi_2) = \texttt{tt}\}$$

$$validStates(\phi_1 \vee \phi_2, \mathcal{A}) = \{(s,p) \mid \mathcal{A}(s,p,\phi_2) = \texttt{tt} \vee \mathcal{A}(s,p,\phi_2) = \texttt{tt}\}$$

$$validStates(\neg\phi, \mathcal{A}) = \{(s,p) \mid \mathcal{A}(s,p,\phi) = \texttt{ff}\}$$

$$validStates(\texttt{tt}, \mathcal{A}) = \{(s,p) \mid s \in S \wedge p \in \mathcal{P}\}$$

$$validStates(\texttt{ff}, \mathcal{A}) = \emptyset$$

$$validStates(\phi, \mathcal{A}) = \{(s,p) \mid \mathcal{A}(s,p,\phi) = \texttt{tt}\}$$

Another useful function is $validColours : cl(\phi) \times S \times AS_{\mathcal{K}}^{\phi} \to 2^{\mathcal{P}}$ which returns a set of parameters for which given formula is assumed to be true in given state.

$$validColours(\phi, state, \mathcal{A}) = \{p | (state, p) \in validStates\}$$

We also define function $update : AS_{\mathcal{K}}^{\phi} \times S \times 2^{\mathcal{P}} \times tcl(\phi) \to AS_{\mathcal{K}}^{\phi}$ which takes current assumptions, a state, set of parameters and a formula and returns assumptions updated so that for all parameters of the given set, formula holds in given state.

```
$update(\mathcal{A}, state, colours, \phi)$ :
for all $p \in colours$ do
    Set $\mathcal{A}(state, p, \phi) = \texttt{tt}$
end for
```

3.4 Temporal Operators

In this section, we describe how the CHECKFORMULA is implemented for each of the temporal operators. Note that all of the following algorithms has implicit termination and therefore needs a proper termination detection algorithm to correctly terminate.

3.4.1 Exist Next Operator

```
1: **Process variables:**
2: $\mathcal{K} = (id, f, \mathcal{P}, S, I, \xrightarrow{p}, L)$        ▷ Kripke fragment
3: $\phi = \text{EX}\phi_1$        ▷ CTL formula
4: $\mathcal{A}$        ▷ Initial assumption function
5: procedure INIT
6:     for all $(state, colSet)$ in $validStates(\phi_1, \mathcal{A})$ do
7:         for all $(pred, tranCol)$ in $predecessors(state)$ do
8:             SEND$(f(state), (pred, colSet \cap tranCol))$
9:         end for
10:    end for
11: end procedure
12: procedure RECEIVE$(colSet, to)$
13:     $\mathcal{A} \leftarrow update(\mathcal{A}, to, colSet, \phi)$
14: end procedure
```

The simplest of temporal operators is the EX operator. During initialization, all states and colors where ϕ holds are computed. For each of such states, all predecessors are considered and appropriate message that will cause assumption update is sent.

If formula is marked as valid in state s for color p, it means a message containing such state and color has been received. This can only happen if said state has a successor under color p where φ_1 holds. Therefore no false positive results are produced. Also, for each state where φ_1 holds for color p, a message is sent for all predecessors of such state. Therefore all states where EXφ_1 holds are labeled accordingly. Since all correct states are labeled and no false positives are possible, the algorithm is correct.

In the worst case, algorithm has to send every color over every edge, therefore worst case message complexity is $card(\rightarrow_i)$. In practice, message count is usually much lower, since multiple colors can be packed into one message.

Assuming the validity of φ_1 is computed for all states, function $validStates(\varphi_1, \mathcal{A})$ can be computed in $O(|internal(S_i)| \cdot |\mathcal{P}|)$. The function *predecessors* can be pre-computed for all states in $O(card(\rightarrow_i))$ time. The procedure SEND is called at most $card(\rightarrow_i)$ times and the parameter set intersection is at worst linear in the size of parameter space.

This would result in $O(|\mathcal{P}| \cdot card(\rightarrow_i))$ complexity. However, this can be further reduced to $O(card(\rightarrow_i))$ since we can observe that for every predecessor where $|tranCol| > 1$, only one message is sent instead of $|tranCol|$ messages. The color set intersection can also be performed in $O(|tranCol|)$ time. This means that the price of set intersection is amortized by the reduced number of transmitted messages.

3.4.2 Exist Until Operator

```
1: Process variables:
2: $\mathcal{K} = (id, f, \mathcal{P}, S, I, \xrightarrow{p}, L)$          ▷ Kripke fragment
3: $\phi = \mathrm{E}\phi_1\mathrm{U}\phi_2$          ▷ CTL formula
4: $\mathcal{A}$          ▷ Initial assumption function
5: procedure INIT
6:     for all $(state, colSet)$ in $validStates(\phi_2, \mathcal{A})$ do
7:         $\mathcal{A} \leftarrow update(\mathcal{A}, state, colSet, \phi)$
8:         for all $(pred, tranCol)$ in $predecessors(state)$ do
9:             SEND$(f(state), (pred, colSet \cap tranCol))$
10:        end for
11:    end for
12: end procedure
13: procedure RECEIVE$(colSet, state)$
14:    $colSet \leftarrow colSet \cap valid(\phi_1, to, \mathcal{A})$
15:    if $colSet \neq \emptyset$ and $colSet \setminus valid(\phi, to, \mathcal{A}) \neq \emptyset$ then
16:        $\mathcal{A} \leftarrow update(\mathcal{A}, to, colSet, \phi)$
17:        for all $(pred, tranCol)$ in $predecessors(to)$ do
18:            SEND$(f(pred), (pred, colSat \cap tranCol))$
19:        end for
20:    end if
21: end procedure
```

The EU operator is a little more complex, but again fairly simple. The algorithm starts by computing all states and colours where ϕ_2 is true. Starting from these states, a backpropagation of parameter sets along the reversed transitions is performed. During the computation, the propagated parameter set is updated to reflect the validity of ϕ_1 and the validity of transitions used along the path. Note that backpropagation is stoped as soon as there is no new information computed (*colSat* is either empty or equal to already computed assumptions).

Algorithm's correctness can be shown by induction to the length of the path required to reach state where φ_2 holds using only states where φ_1 holds.

For $i = 0$, the algorithm is correct since it starts by finding all states and colours for which the φ_2 holds and markes all of these states with the formula φ.

Let us assume that algorithm found all states where paths of length i. This also means that a message with appropriate colours has been sent to all predecessors.

3.4.3 All Until Operator

```
Process variables:
$\mathcal{K} = (id, f, \mathcal{P}, S, I, \xrightarrow{p}, L)$                    ▷ Kripke fragment
$\phi = \mathrm{A}\phi_1\mathrm{U}\phi_2$                                      ▷ CTL formula
$T = \xrightarrow{p}$                                               ▷ Uncovered edges
$\mathcal{A}$                                                  ▷ Initial assumption function
procedure INIT
    for all $(state, colSet)$ in $validStates(\phi_2)$ do
        for all $(pred, tranCol)$ in $predecessors(state)$ do
            SEND$(f(state), (state, pred, \mathcal{P} \cap tranCol))$
        end for
    end for
end procedure
procedure RECEIVE$(colSet, from, to)$
    $T \leftarrow T \setminus \{(to, p, from) | p \in colSet\}$
    $colSet \leftarrow \{p | p \in colSet \wedge \forall s_2 \in S.(to, p, s_2) \notin T\}$
    $colSet \leftarrow colSet \cap valid(\phi_1, to, \mathcal{A})$
    if $colSet \neq \emptyset$ and $colSet \setminus valid(\phi, to, \mathcal{A}) \neq \emptyset$ then
        $\mathcal{A} \leftarrow update(\mathcal{A}, to, colSet, \phi)$
        for all $(pred, tranCol)$ in $predecessors(to)$ do
            SEND$(f(pred), (to, pred, colSet \cap tranCol))$
        end for
    end if
end procedure
```

The AU operator is the most complex one to handle. As opposed to EX, which requires at least one valid successor to be true, AU requires that all successors of the specific node are valid. In order to compute such information, we create a copy of transition relation and call it T.

During the computation, T is modified in such way, so that we can guarantee that if edge is not present in T, this edge leads to a state where either ϕ_2 or $A\phi_1 U\phi_2$ holds. This way, we can guarantee that only appropriate states and colors are marked as valid by our algorithm.

3.5 Merge message buffer

The main argument of coloured model checking efficiency is based on the assumption that operations on parameter sets are in practice less expensive than graph traversal. However, even a simple model structure can easily break the computation into many simultanious traversals. When two different colour sets are marked as valid in a state due to two distinct transitions, unfortunate timing can easily prevent these two colour sets from merging into one. This leads to two outgoing messages instead of one, which in the end results in two separate graph traversals instead of one.

In some cases, this simply cannot be avoided, since in order to know exactly when to wait for a merge and when to continue the traversal, we would basically have to solve the coloured model checking problem. However, we can take advantage of the fact, that many messages cannot be processed directly at the time of arrival, since message processing can be quite costly operation, especially when the transition system is being lazily calculated. Therefore we use a message buffer to store incomming messages that cannot be processed directly.

These bufferes can grow quite large during the computation and even outgrow the state space of the transition system itself. Of course, it would be easy to just slow down the creation of messages to prevent the buffers from growing. However, many messages in these buffers contain data that are either duplicate or can be merged into one message while maintaining correct semantics.

In order to reduce to the number of unnecessary graph traversals and reduce memory footprint of message buffers while maintaining good performance, we employ a merge message buffer as described in [REFERENCE].

In this text, we only provide the implementation for messages that contian destination node and colour set as used in EX and EU operators. However, the implementation for messages used by AU operator can be obtained trivially by replacing all occurences of destination node with pair of destination and source nodes.

```
**MergeBuffer**
$Q \leftarrow IterableHashMap$
**procedure** INSERT(node, colours)
    **if** $Q$ **hasKey** $node$ **then**
        $current \leftarrow Q.get(node)$
        $Q.replace(node, colours \cup current)$
    **else**
        $Q.insert(node, colours)$
    **end if**
**end procedure**
**procedure** EMPTY
    **return** $Q.isEmpty()$
**end procedure**
**procedure** TAKE
    $val \leftarrow Q.interator().first()$
    $Q.remove(val.key)$
    **return** $val$
**end procedure**
```

Key property of the buffer is that it is backed by an *IterableHashMap*. Compared to traditional *HashMap*, *IterableHashMap* provides also an iterator on all of its key-value pairs. This gives us the ability to take one (non-deterministic) element out of the map in constant time. Considering a reasonable hash function on the node set, we can guarantee that each operation on the underlying *IterableHashSet* can be done in constant time. Therefore the only interesting operation in terms of time complexity is the union of two colour sets, which can be done in $O(\mathcal{P})$ complexity.

It is important to note that the resulting buffer does not preserve the FIFO properties of a queue. However, this is not required by the algorithm (actually, performing a random search instead of classic DFS is considered a valid optimalization heuristic).

We do not provide exhaustive benchmark of this heuristic, since the main priority of this work is the model checking algorithm itself. However, a comparison with linked queue and circular buffer on models tested in the Scalability section showed that merge queue easily outperforms both of them especially in highly distributed environments. Linked queue was usually two times slower while circular buffer managed to provide approximately 60-80% of the merge buffer performance.

One reason for this is the fact that more distributed computations have generally more cross transitions. This increases the cost of traversal compared to parameter set operations which in turn makes the merge buffer more effective. Also, the properties of this buffer allow in some cases for super-linear scalablilty, since greater number of processes can produce higher number of merged traversals.

4 Implementation

Our algorithm is implemented in a proof-of-concept distributed CTL model checker available in the github repositiory of Sybila organization [cite]. In this section we briefly discuss the implementation details of this project.

Although most of the core model checking module is fully operational and stable, the project is still mainly in experimental phase, since new features, heuristics and modelling approaches are still being considered and reevaluated.

4.1 Architecture

In order to provde greater flexibility and ease of development, majority of the project is implemented in Java. However, several parts still require C++ code reused from similar previous projects.

The distributed environment is mainly provided by MPJ [cite], Java implementation of standard MPI interface. However, the model checking algorithm itself does not depend on any concrete communication library and can be easily adapted to any similar distributed communication tool.

The whole project is divided into several modules in order to maximize extensibility and modifiability during future development. This section provides a quick overview of each of these modules.

4.1.1 CTL Parser

In order to provide user with easy way to input CTL formulas, this module implements a parser for modified formula specification language used in BioDivIne[cite].

The original parser only supports LTL formulas, therefore the grammar has been modified to allow for CTL operators. However, user familiar with the original LTL syntax should have no problems adapting to the CTL syntax.

The grammar is written in Antlr parser generator and apart from the whole range of CTL and boolean operators supports also boolean and float propositions.

This module also handles the transformation of user provided formula into minimal set of operators supported by the main model checker.

4.1.2 Model Checker

This is the core module implementing the model checking algorithm described in this work. It defines the interfaces and contracts representing the kripke fragment, parameter set, partition function and inter-process communication.

It is completely independant on the implementation of said interfaces, and therefore provides great flexibility and extensibility. This module also provides partial implementations of some of these interfaces to ease the development of model related modules.

This module also implements the termination detection algorithm designed by Safra[cite]. The algorithm itself can't be easily overriden, however, the communication is again isolated into one, easily replacable class, so the support for different communcation libraries can be easily provided.

4.1.3 ODE Abstraction

This module is based on the ODE state space generator from the LTL model checking tool BioDiViNe. The code responsible for model parsing and abstraction calculation is reused directly and Java Native Interface(JNI) is used to extract resulting model into corresponding Java data structures.

The state space generator responsible for evaluation of atomic propositions and computation of successors/predecessors is also based on code from the BioDiViNe project. However, this section has been completely rewritten into Java and adapted to the CTL paradigm. This minimizes the number of slow JNI calls to existing C++ code. This new state space generator also features several bugfixes and speed optimalizations.

This module also implements the rectangular state space partition function described in this work. The parameter set is implemented using Google Guava Range Set[cite] which provides great flexibility and huge feature set while maintaining good performance.

4.1.4 Thomas Network Abstraction

This module is based on the Thomas Network state space generator from the LTL model checking tool Parsybone. Similarly to the ODE Abstraction module, most of the model parsing and preprocessing is done using the original code and extracted using JNI.

The parameter set is implemented using a EWAH Compressed Bitmap[cite], which provides decent performance even on large parameter sets and is very easy to use.

At the moment of writing, this module does not provide any good partitioning implementation, since much of this functionality is still in developement and most of the current implementation is subject to change. However, sequential computation on one processing node is fully supported.

4.1.5 Frontend

Frontend module ties together the functionality of all modules into several runnable utilities. However, the documentation and general outilne and output format of these tools have not yet been finalized and is subject to change. Therfore the code in this module should be taken more as an example of usage of different modules.

5 Experimental Evaluation

The aim of this chapter is to demonstrate the soundness of our algorithm on two biologically relevant models. First model is fairly simple and is used for benchmarking purposes, since it can be easily scaled and modified while maintaining similar properties and structure. Second model exhibits non trivial behaviour and is used to demonstrate the ability of our method to correctly detect such behaviour.

5.1 Scaleability

We evaluate the scalability of the algorithm on an ODE model of reversible catalytic reaction with varying number of intermediate enzyme-substrate complexes. Using this model, we can scale the number of intermediate products/variables (N), discretization thresholds (T) and unknown parameters. For simplicity, we assume that each variable is evaluated on the same amount of thresholds, which results in the total state space size of $(T-1)^N$.

The benchamarks were performed on a cluster of 12 computational nodes, each equipped with 16 GB of RAM memory and a quad-core Intel Xeon 2Ghz processor. In order to focus on the distribution and not multi-core performance, we utilize only one core on each machine. However, note that the algorithm can be easily execued on a multicore machine or a cluster of multicore machines using all cores. No other resource intensive software was running at these machines at the time of benchmarking.

The model itself is descirbed in the following scheme. The first line represents a simplified chemical equation of the model. The following lines describe differential equations for every species. The last two lines describe used parameters.

In the subsequent tables, we provide detailed information about the runtime of the distributed algorithm on various modifications of the model and the CTL formula $\mathrm{AG}(P \leq 30)$. The value N/A represents a situation when the algorithm ran out of memory. The results has been obtained as an arithmetic mean of several experiments.

They generally illustrate very good scaleability in terms of eval-

$$\begin{array}{c}
S + E \rightleftharpoons ES_1 \rightleftharpoons \ldots \rightleftharpoons ES_k \rightleftharpoons P + E \\
\hline
\dot{S} = 0.1 \cdot ES_1 - p_1 \cdot E \cdot S \\
\dot{E} = 0.1 \cdot ES_1 - p_2 \cdot E \cdot S + 0.1 \cdot ES_k - p_2 \cdot E \cdot P \\
\dot{ES_1} = 0.01 \cdot E \cdot S - p_3 \cdot ES_1 + 0.05 \cdot ES_2 \\
\vdots \\
\dot{ES_k} = 0.1 \cdot ES_{k-1} - p_k \cdot ES_k + 0.01 \cdot E \cdot P \\
\dot{P} = 0.1 \cdot ES_k - p_{k+1} \cdot E \cdot P - 0.1 \cdot P \\
\hline
p_1 = 0.01, p_2 = 0.01, p_3 = 0.2, \\
p_k = 0.15, p_{k+1} = 0.01
\end{array}$$

uated model properties. Note that the number of parameters can change the structure of the transition system, which may result in shorter run times, as demonstrated in table 5.1. Also note that in several occasions, especially when comparing single and dual machine experiments, the algorithm exhibits a super-linear speed-up. There are two main reasons of this behaviour.

First is the merge buffer as described in [reference], which can merge several traversals into one and therefore reduce expected overhead. Second is the garbage collector used by the Java Runtime Environment. Single machine experiments generally take up more memory, since the whole state space has to be stored on one computer. This leads to more aggressive garbage collecting and thus results in higher run times. On the other hand, when using high number of computational nodes, the garbage collection has only insignificant impact thanks to the high amount of available memory.

# of params / # of nodes	1	2	3	4	5	6
1	6456	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
2	2610	6179	5089	*N/A*	*N/A*	*N/A*
3	1696	4022	3661	3784	*N/A*	*N/A*
4	854	1759	2285	2454	*N/A*	*N/A*
5	683	1371	1365	1580	1736	*N/A*
6	499	1095	1019	1254	1609	1350
7	435	861	796	1023	1406	1340
8	292	642	650	853	1134	1118
9	258	439	630	752	983	962
10	232	418	557	637	839	822
11	198	347	516	553	784	679
12	177	329	420	562	759	660

Table 5.1: The runtime in seconds for the model with 6 variables, 13 thresholds and different number of unknown parameters.

# of variables / # of nodes	4	5	6	7
1	3.3	22	794	*N/A*
2	3.3	12	489	*N/A*
3	3.5	9.5	253	*N/A*
4	3.4	8.2	185	8571
5	2.7	8	112	6608
6	2.4	7.2	101	5291
7	2.7	7.3	77	3024
8	2.3	6.6	64	2630
9	2.3	6.3	55	2366
10	2.5	6.8	52	2081
11	2.7	6.2	47	1999
12	2.2	5.6	41	1828

Table 5.2: The runtime in seconds for the model with 1 unknown parameter, 11 thresholds per the variable and different number of variables.

# of thresholds	10	11	12	13
# of nodes				
1	274	794	1805	6456
2	196	489	1252	2610
3	84	253	570	1696
4	62	185	408	854
5	51	112	280	683
6	40	101	226	499
7	33	77	192	435
8	29	64	161	292
9	26	55	145	258
10	24	52	108	232
11	23	47	94	198
12	22	41	96	177

Table 5.3: The runtime in seconds for the model with 1 unknown parameter, 6 variables and different number of thresholds per variable.

5.2 Case study

To demonstrate the applicability of the algorithm, we investigate a well-known ODE model [cite] which represents a two-gene regulatory network describing interaction of the tumor suppressor protein pRB and the central transcription factor $E2F1$ (see Fig. 5.1 (left)). This network represents the crucial mechanism governing the transition from G_1 to S phase in the mammalian cell cycle. In the G_1-phase the cell makes an important decision. In high concentration levels, $E2F1$ activates the G_1/S transition mechanism. In low concentration of $E2F1$, committing to S-phase is refused and that way the cell avoids DNA replication.

This mechanism is and example of *bistable switch*. The system makes an irreversible decission to finally reach some of the two stable states. The model is represented by two differential equations as seen in Figure 5.1. In order to discretize this model, a piece-wise multi-affine approximation (PMA) has to be computed [cite]. In our experiments, we use 70 thresholds per each variable during this process. However, no significant change in results has been observed for gs1net.pdf

$$\frac{d[pRB]}{dt} = k_1 \frac{[E2F1]}{K_{m1}+[E2F1]} \frac{J_{11}}{J_{11}+[pRB]} - \phi_{pRB}[pRB]$$
$$\frac{d[E2F1]}{dt} = k_p + k_2 \frac{a^2+[E2F1]^2}{K_{m2}^2+[E2F1]^2} \frac{J_{12}}{J_{12}+[pRB]} - \phi_{E2F1}[E2F1]$$

$a = 0.04, k_1 = 1, k_2 = 1.6, k_p = 0.05, \phi_{E2F1} = 0.1$
$J_{11} = 0.5, J_{12} = 5, K_{m1} = 0.5, K_{m2} = 4$

Figure 5.1: G_1/S transition regulatory network and its ODE model taken from [**?**].

3d.png

Figure 5.2: Coloured model checking results. Red and blue parts correspond to the high and low stable regions, respectively. Yellow part displays the states where the overall *bistable switch* formula φ holds.

higher threshold counts.

On the resulting model, we can perform a coloured model checking of the formula $\varphi \equiv \text{EFAG}(\texttt{high}) \wedge \text{EFAG}(\texttt{low})$ considering the initial parameter space $\phi_{pRB} \in [0.001, 0.025]$. The atomic propositions `low` and `high` characterise the location of expected regions of $E2F1$ stability. Based on the results reported in [**?**] we define the stable regions as `high` $\equiv (E2F1 > 4 \wedge E2F1 < 7.5)$ and `low` $\equiv (E2F1 > 0.5 \wedge E2F1 < 2.5)$.

As seen in figure 5.2, the computation has successfully discovered both stable areas and corresponding parameters, as well as area and parameters from which both stable states can be reached. However, note that due to over-approximation, only the stable areas are guaranteed to be true.

6 Conclusion

The aim of this work was to design and implement a method that can efficiently solve parameter synthesis problem for biochemical models and CTL hypothesis.

We proposed an algorithm based on existing algorithms for distributed CTL model checking and coloured model checking of LTL formulas. The algorithm is given as pseudocode and we also provide a working implementation. The algorithm is not limited to the biochemical models and therefore has general applicability.

We also proposed an efficient state space partitioning heuristic for ODE models and a merge buffer heuristic for reduction of unnecessary state space traversals.

The provided implementation is oriented at biochemical models, provides suport for two distinct modelling techniques, full range of CTL operators and features easily extensible modular architecture.

We benchmarked the provided implementation and showed that the algorithm scales well with higher number of computation nodes and provides sufficient performance to investigate models with large state spaces, even though the exponential state space explosion still remains a problem. The soundness of the algorithm and applicability has been demonstrated on a biologically relevant model of bistable switch.

In the future, the implementation can be optimized to provide even better performance and scalability, especially on multicore architectures. Also support for different modelling techniques and partitioning heuristics can be implemented.

Another possible direction of development would be a coloured symbolic model checking, which should provide even better performance compared to our explicit method, especially in terms of memory usage.

Bibliography

[1] S. author, "Assumption-based distribution of CTL model checking," *STTT*, vol. 7, no. 1, pp. 61–73, 2005.