Information and Computation 242 (2015) 108–127

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Synthesizing structured reactive programs via deterministic tree automata

Benedikt Brütsch

RWTH Aachen University, Lehrstuhl für Informatik 7, Germany

ARTICLE INFO

Article history:
Received 21 October 2013
Available online 24 March 2015

ABSTRACT

Existing approaches to the synthesis of controllers in reactive systems typically involve the construction of transition systems such as Mealy automata. In 2011, Madhusudan proposed structured programs over a finite set of Boolean variables as a more succinct formalism to represent the controller. He provided an algorithm to construct such a program from a given ω-regular specification without synthesizing a transition system first. His procedure is based on two-way alternating ω-automata on finite trees that recognize the set of "correct" programs.

We present a more elementary and direct approach using only deterministic bottom-up tree automata and extend Madhusudan's results to the wider class of programs with bounded delay, which may read several input symbols before producing an output symbol (or vice versa). In addition, we show a lower bound for the size of these tree automata. Finally, we prove a lower bound for the number of Boolean variables that are required for a structured program to satisfy a given LTL specification, almost matching the known upper bound.

1. Introduction

Algorithmic synthesis is a rapidly developing field with many application areas such as reactive systems, planning and economics. Most approaches to the synthesis of controllers in reactive systems, for instance [1–4], involve synthesizing transition systems such as Mealy or Moore automata. Unfortunately, the resulting transition systems can be very large, which has motivated the development of techniques for the reduction of their state space (for example, [5]). Furthermore, the method of bounded synthesis [6,7] can be used to synthesize minimal transition systems by iteratively increasing the bound on the size of the resulting system until a solution is found. However, it is not always possible to obtain small transition systems. For example, for certain specifications in linear temporal logic (LTL), the size of the smallest transition system satisfying the specification is doubly exponential in the length of the formula [8].

The subject of the present paper is the synthesis of controllers represented in a more succinct way: We consider *structured reactive programs* over a finite set of Boolean variables, building on work of Madhusudan [9], refining his results and showing limitations of this approach. Structured reactive programs are non-terminating while-programs with input and output statements. They can be significantly smaller (regarding the length of the program code) than equivalent transition systems, and in contrast to transition systems, they are structured in the sense that they can be decomposed into subprograms.

E-mail address: bruetsch@automata.rwth-aachen.de.

http://dx.doi.org/10.1016/j.ic.2015.03.013
0890-5401/© 2015 Elsevier Inc. All rights reserved.

Madhusudan [9] proposes a procedure to synthesize such programs from ω-regular specifications without computing a transition system first, using the fact that structured programs can be represented by their syntax trees. Given a finite set of Boolean variables and a nondeterministic Büchi automaton (which we call the specification automaton) recognizing the complement of the specification, he constructs a two-way alternating ω-automaton on finite trees that recognizes the set of *all* programs over these variables that satisfy the specification. This automaton can be transformed into a nondeterministic tree automaton (NTA) to check for emptiness and extract a minimal program (regarding the height of the corresponding tree) from that set. In contrast to the transition systems constructed by classical synthesis algorithms, the synthesized program does not depend on the specific syntactic formulation of the specification, but only on its meaning.

In this paper, we present a direct construction of a deterministic bottom-up tree automaton (DTA) recognizing the set of correct programs, without a detour via more intricate types of automata. The DTA inductively computes a representation of the behavior of a given program in the form of so-called *co-execution signatures*. A co-execution is a pair consisting of a computation of the program and a corresponding run of the specification automaton, and a co-execution signature for a given program and a given specification automaton captures the essential information about their possible co-executions. A similar representation is used by Lustig and Vardi in their work on the synthesis of reactive systems from component libraries [10] to characterize the behavior of the components.

Our approach is not limited to programs that read input and write output in strict alternation, but extends Madhusudan's results to the more general class of programs with *bounded delay*: In general, a program may read multiple input symbols before writing the next output symbol, or vice versa, causing a delay between the input sequence and the output sequence. In a game-theoretic context, such a program corresponds to a strategy for a controller in a game against the environment where in each move the controller is allowed to either choose one or more output symbols or skip and wait for the next input (see [11]). This can be a suitable model for systems where the interaction between the controller and the environment is based on buffers for the input and output symbols. We consider programs that never cause a delay greater than a given bound $k \in \mathbb{N}$.

For a fixed k, the complexity of our construction matches that of Madhusudan's algorithm. In particular, the size of the resulting DTA is exponential in the size of the given specification automaton and doubly exponential in the number of program variables. In fact, we establish a lower bound, showing that the set of all programs over m Boolean variables that satisfy a given specification cannot even be recognized by an NTA with less than $2^{2^{m-1}}$ states, if any such program exists. However, note that a DTA (or NTA) accepting precisely these programs enables us to extract a minimal program for the given specification and the given set of program variables. Hence, while the tree automaton may be large, the synthesized program itself might be rather small.

To lay a foundation for our study of the synthesis of structured reactive programs, we define a formal semantics for such programs, which is only informally indicated by Madhusudan. To that end, we introduce the concept of *Input/Output/Internal machines (IOI machines)*, which are composable in the same way as structured programs. This allows for an inductive definition of the semantics.

The second half of this paper is dedicated to the question of how many (Boolean) variables are needed to satisfy a given specification. More specifically, we show that for certain specifications in LTL, at least $\Omega(2^{\sqrt{n}})$ Boolean variables are required, where n is the size of the respective LTL formula. This lower bound almost matches the exponential upper bound that can be derived from the doubly exponential upper bound for the size of transition systems for a given LTL specification [8]. In order to prove this lower bound, we draw on concepts from graph theory and exploit the fact that the so-called transition graphs of structured programs over a small number of variables have small tree-width. We show that for certain specifications, the tree-width of the transition graphs of the programs satisfying these specifications must be large, which allows us to deduce a lower bound for the number of variables used by these programs.

Related work Besides the concept of structured reactive programs, there are several other approaches to controller synthesis that may yield succinct implementations of the controller instead of classical transition systems. For instance, Aminof, Mogavero and Murano [12] provide a round-based algorithm to synthesize *hierarchical transition systems*, which can be exponentially more succinct than corresponding "flat" transition systems. The desired system is constructed in a bottom-up manner: In each round, a specification is provided and the algorithm constructs a corresponding hierarchical transition system from a given library of available components and the hierarchical transition systems created in previous rounds. Thus, in order to obtain a small system in the last round, the specifications in the previous rounds have to be chosen in an appropriate way.

Current techniques for the synthesis of (potentially) compact implementations in the form of circuits or programs typically proceed in an indirect way, by converting a transition system into such an implementation. For example, Bloem et al. [13] first construct a symbolic representation (a binary decision diagram) of an appropriate transition system and then extract a corresponding circuit. However, this indirect approach does not necessarily yield a succinct result.

Another succinct model for the representation of strategies in infinite games was introduced by Gelderie [14] in the form of *strategy machines*. They are equipped with a set of control states and a memory tape, providing a similar separation between control flow and memory as structured programs.

2. Syntax and semantics of structured programs

We consider structured programs as defined in [9]. Such programs read input symbols and write output symbols in the form of bitvectors of length N_{in} and N_{out}, respectively, for fixed numbers $N_{\text{in}}, N_{\text{out}} \geq 1$. *Expressions* and *programs* over a finite set B of Boolean variables are defined by the following grammar, where b stands for a variable in B and $\vec{b}$ for a vector of variables:

$$\begin{aligned}
\langle exp\rangle \quad &::= \quad \texttt{true} \mid \texttt{false} \mid b \mid \langle exp\rangle \wedge \langle exp\rangle \mid \langle exp\rangle \vee \langle exp\rangle \mid \neg\langle exp\rangle \\
\langle prog\rangle \quad &::= \quad b := \langle exp\rangle \mid \texttt{input}\,\vec{b} \mid \texttt{output}\,\vec{b} \mid \langle prog\rangle;\langle prog\rangle \\
&\qquad \texttt{if}\,\langle exp\rangle\,\texttt{then}\,\langle prog\rangle\,\texttt{else}\,\langle prog\rangle \mid \texttt{while}\,\langle exp\rangle\,\texttt{do}\,\langle prog\rangle
\end{aligned}$$

Intuitively, "`input` $\vec{b}$" reads an input symbol (i.e., a bitvector) and stores it in the variables $\vec{b}$. (Hence, $\vec{b}$ must consist of N_{in} variables.) Conversely, "`output` $\vec{b}$" writes the output symbol consisting of the current values of the variables $\vec{b}$. (Thus, $\vec{b}$ must consist of N_{out} variables in this case.) From the atomic programs, i.e., input statements, output statements and assignments, larger programs can be composed using concatenation, selection (`if/then/else`) and repetition (`while`). A formal semantics will be developed in the following sections.

2.1. IOI machines

We will define the semantics of the programming language defined above by mapping each program to a certain kind of transition system, which we call an *input/output/internal machine*, or *IOI machine* for short.

First, let us fix some notation: A *variable valuation* for a given set of Boolean variables B is a function $\sigma: B \to \{0,1\}$ that assigns to each variable its current value. We denote by Val_B the set of all possible variable valuations for B. We write $\sigma[b/c]$ to denote the variable valuation that is obtained from σ by replacing the value of b with $c \in \{0,1\}$. We also extend this notation to vectors $\vec{b}$ of variables and vectors $\vec{c}$ of Boolean values. We can now give the definition of an IOI machine:

Definition 2.1 *(IOI machine).* Let B be a finite set of Boolean variables. An IOI machine over B is a tuple $(Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \Delta, \mu, Q_{\text{entry}}, Q_{\text{exit}})$, where

- Q is a finite set of states,
- Σ_{in} is a finite alphabet of input symbols,
- Σ_{out} is a finite alphabet of output symbols,
- $\Delta = \Delta_{\text{int}} \cup \Delta_{\text{in}} \cup \Delta_{\text{out}}$ is a transition relation, consisting of
  input transitions $\Delta_{\text{in}} \subseteq Q \times \{(a,\varepsilon) \mid a \in \Sigma_{\text{in}}\} \times Q$,
  output transitions $\Delta_{\text{out}} \subseteq Q \times \{(\varepsilon,a) \mid a \in \Sigma_{\text{out}}\} \times Q$,
  internal transitions $\Delta_{\text{int}} \subseteq Q \times \{(\varepsilon,\varepsilon)\} \times Q$,
- $\mu: Q \to Val_B$ is a *memory content function* that determines the variable valuation for each state,
- $Q_{\text{entry}} \subseteq Q$ is a set of entry states with $\mu(q_1) \neq \mu(q_2)$ for all $q_1, q_2 \in Q_{\text{entry}}$,
- $Q_{\text{exit}} \subseteq Q$ is a set of exit states with $\mu(q_1) \neq \mu(q_2)$ for all $q_1, q_2 \in Q_{\text{exit}}$.

Let σ_0 be the initial variable valuation, where all variables have the value 0. We require that there exists a unique state $q_0 \in Q_{\text{entry}}$ with $\mu(q_0) = \sigma_0$ and call it the *initial state* of the IOI machine.

2.2. From programs to IOI machines

We can now provide an inductive definition of the semantics of structured programs. For each expression e, the function $[\![e]\!] : Val_B \to \{0,1\}$ determines the value of the expression e for a given variable valuation σ. It can be defined inductively in the usual way, so we omit the details. With each program p, we associate an IOI machine $ioi(p)$ in the following way. Note: For ease of notation, we let $\Sigma_{\text{in}} = \{0,1\}^{N_{\text{in}}}$ and $\Sigma_{\text{out}} = \{0,1\}^{N_{\text{out}}}$ from now on.

Atomic programs

- $ioi(\text{"}b := e\text{"}) := (Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \Delta, \mu, Q_{\text{entry}}, Q_{\text{exit}})$ with
  - $Q = \{entry, exit\} \times Val_B$,
  - $\Delta = \big\{ \big((entry,\sigma_1), (\varepsilon,\varepsilon), (exit,\sigma_2)\big) \mid \sigma_2 = \sigma_1[b/[\![e]\!](\sigma)] \big\}$,
  - $\mu\big((x,\sigma)\big) = \sigma$ (for $(x,\sigma) \in Q$),
  - $Q_{\text{entry}} = \{(entry,\sigma) \mid \sigma \in Val_B\}$,
  - $Q_{\text{exit}} = \{(exit,\sigma) \mid \sigma \in Val_B\}$.
- $ioi(\text{"}\texttt{input}\ \vec{b}\text{"}) := (Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \Delta, \mu, Q_{\text{entry}}, Q_{\text{exit}})$ with

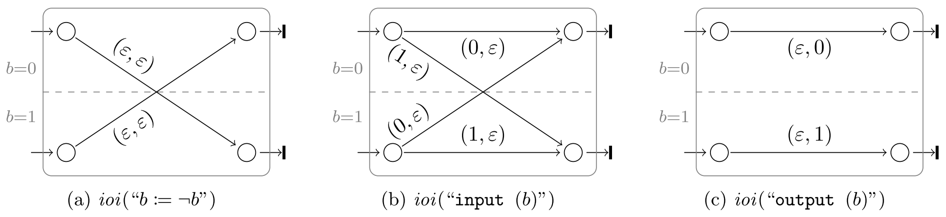

Fig. 1. Illustration of IOI machines for atomic programs over $B = \{b\}$. Entry states are indicated by a small incoming arrow $\rightarrow$, exit states by an outgoing arrow$\rightarrow$ **I**.

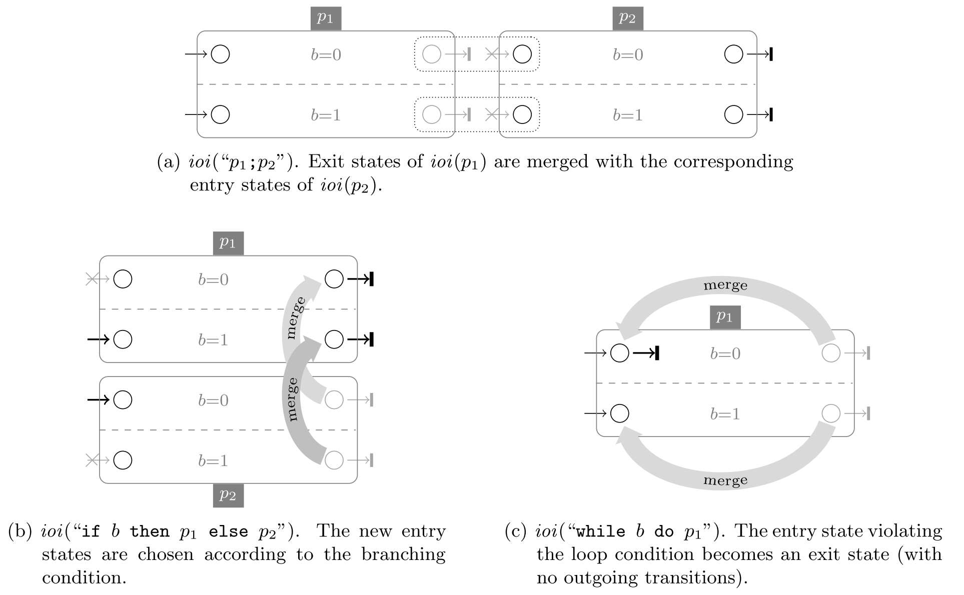
p_1
p_2
$b{=}0$
$b{=}1$
(a) ioi("p_1;p_2"). Exit states of $ioi(p_1)$ are merged with the corresponding entry states of $ioi(p_2)$.
merge
merge
(b) ioi("if b then p_1 else p_2"). The new entry states are chosen according to the branching condition.
merge
merge
(c) ioi("while b do p_1"). The entry state violating the loop condition becomes an exit state (with no outgoing transitions).

Fig. 2. Illustration of IOI machines for composite programs over $B = \{b\}$. States that get merged with other states are grayed out.

- Q, Q_{entry}, Q_{exit} and μ as above,
- $\Delta = \left\{ \big((entry, \sigma_1), (\vec{c}, \varepsilon), (exit, \sigma_2)\big) \mid \vec{c} \in \Sigma_{\text{in}} \text{ and } \sigma_2 = \sigma_1[\vec{b}/\vec{c}] \right\}$.

- ioi("output $\vec{b}$") $:= (Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \Delta, \mu, Q_{\text{entry}}, Q_{\text{exit}})$ with
  - Q, Q_{entry}, Q_{exit} and μ as above,
  - $\Delta = \left\{ \big((entry, \sigma), (\varepsilon, \vec{c}), (exit, \sigma)\big) \mid \vec{c} \in \Sigma_{\text{out}} \text{ and } \sigma(\vec{b}) = \vec{c} \right\}$.

Examples for IOI machines of atomic programs using only a single variable are shown in Fig. 1. In the examples, the input and output alphabets are simply $\{0, 1\}$.

Composite programs We will now describe how the IOI machines for composite programs can be constructed from those of the respective subprograms. This is illustrated in Fig. 2. Note that for every program, the corresponding IOI machine has precisely one entry state and precisely one exit state for each possible variable valuation.

Let p_1, p_2 be two programs and let $ioi(p_1)$ and $ioi(p_2)$ be their associated IOI machines, with memory content functions μ_1 and μ_2, respectively. We assume that $ioi(p_1)$ and $ioi(p_2)$ have disjoint sets of states, which can always be achieved by renaming.

The IOI machine of the program "p_1;p_2" is the series composition ("concatenation") of $ioi(p_1)$ and $ioi(p_2)$, which is obtained by merging every exit state q_1 of $ioi(p_1)$ with the corresponding entry state q_2 of $ioi(p_2)$ with the same variable valuation, i.e., with $\mu_1(q_1) = \mu_2(q_2)$. Note that such a state q_2 always exists because by construction, $ioi(p_2)$ has precisely one entry state for each possible variable valuation. The entry states of the new IOI machine are the same as in $ioi(p_1)$.

For the program "if e then p_1 else p_2", the corresponding IOI machine is the parallel composition of $ioi(p_1)$ and $ioi(p_2)$, which is constructed in the following way: The exit states of $ioi(p_2)$ are merged with the corresponding exit states (with the same variable valuation) of $ioi(p_1)$. Again, the existence of these corresponding states is guaranteed by construction of $ioi(p_2)$. Furthermore, we define a new set of entry states, which consists of those entry states q of $ioi(p_1)$ whose variable valuation $\mu_1(q)$ satisfies the condition e and those entry states q' of $ioi(p_2)$ whose variable valuation $\mu_2(q')$ violates e.

Finally, we obtain the IOI machine for "`while` e `do` p_1" by merging the exit states of $ioi(p_1)$ with the corresponding entry states with the same variable valuation. The set of entry states is the same as in $ioi(p_1)$, but the set of exit states now consists of those entry states q whose variable valuation $\mu_1(q)$ violates the loop condition e.

2.3. Computations and behavior of a program

A *computation* ϱ of a program p is a finite or infinite sequence of subsequent transitions of the corresponding IOI machine $ioi(p)$:

$$\varrho \;=\; q_1 \xrightarrow{(a_1,a_1')} q_2 \xrightarrow{(a_2,a_2')} q_3 \xrightarrow{(a_3,a_3')} \cdots$$

Note that (for $i \geq 1$) $a_i \in \Sigma_{\text{in}} \cup \{\varepsilon\}$ and $a_i' \in \Sigma_{\text{out}} \cup \{\varepsilon\}$. Such a computation yields a pair of (finite or infinite) words $(a_1a_2a_3\ldots,\ a_1'a_2'a_3'\ldots)$, which we call an *input/output sequence*, over the alphabets Σ_{in} and Σ_{out}, respectively.

An *initial computation* starts at the initial entry state q_0, where all variables have the value 0. The *infinite behavior* $\langle\!\langle p\rangle\!\rangle$ of a program p is the set of infinite input/output sequences $(\alpha,\beta) \in \Sigma_{\text{in}}^{\omega} \times \Sigma_{\text{out}}^{\omega}$ that can be produced by an initial computation of p. Furthermore, we call a program *reactive* if all its initial computations can be extended to infinite computations that yield an infinite input and output sequence.

At any given time during a computation ϱ as above, the length of the input sequence $a_1a_2\ldots a_i$ and the output sequence $a_1'a_2'\ldots a_i'$ might differ. The supremum of these length differences along a computation is called the *delay* of the computation:

$$delay(\varrho) = \sup\left\{\,\big||a_1a_2\ldots a_i| - |a_1'a_2'\ldots a_i'|\big| \;\mid\; i \geq 0\,\right\}$$

If the delay of a computation does not exceed a given bound $k \in \mathbb{N}$ then we call this computation *k-bounded*. A program is said to have *k-bounded delay* if all its initial computations are k-bounded. By restricting (for a given number k) the infinite behavior of a program p to input/output sequences of k-bounded initial computations, we obtain the *k-bounded infinite behavior* $\langle\!\langle p\rangle\!\rangle_k$ of p.

Later, we will consider the decomposition of computations of a program into computations of its subprograms. At the beginning of a subprogram computation, there might already be a length difference $d = |u| - |v|$, which we call *start difference*, between the input sequence u and the output sequence v that have been produced before. Similarly, the length difference at the end of the computation is called *end difference*. We define the delay of a computation ϱ for the start difference d as

$$delay(\varrho, d) = \sup\left\{\,\big|d + |a_1a_2\ldots a_i| - |a_1'a_2'\ldots a_i'|\big| \;\mid\; i \geq 0\,\right\}.$$

3. Solving the synthesis problem using deterministic tree automata

3.1. The synthesis problem

The *synthesis problem for structured reactive programs with bounded delay* can be formulated as follows:

Given: An ω-regular specification $R \subseteq (\Sigma_{\text{in}} \times \Sigma_{\text{out}})^{\omega}$ representing the permissible input/output sequences, a finite set of Boolean variables B and a delay bound $k \in \mathbb{N}$.

Task: Construct a structured reactive program p over B with k-bounded delay such that $\langle\!\langle p\rangle\!\rangle \subseteq R$, or detect that no such program exists.

We say that a program p *satisfies* the specification if $\langle\!\langle p\rangle\!\rangle \subseteq R$. In the following we assume that the specification R is provided in the form of a *nondeterministic Büchi automaton (NBA)* $\mathcal{A}_{\overline{R}}$ over the alphabet $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$ that recognizes the complement of the specification, i.e., $\mathcal{L}(\mathcal{A}_{\overline{R}}) = (\Sigma_{\text{in}} \times \Sigma_{\text{out}})^{\omega} \setminus R$, which is always possible for ω-regular specifications. An NBA $\mathcal{A}$ is a tuple $(S, \Sigma, s_0, \Delta, F)$, where S is a finite set of states, Σ is a finite alphabet, $s_0 \in S$ is the initial state, $\Delta \subseteq S \times \Sigma \times S$ is a transition relation, and $F \subseteq S$ is a set of final states. An (infinite) *run* of $\mathcal{A}$ on an ω-word $\alpha = a_1a_2a_3\ldots$ is a sequence of states $s_1s_2s_3\ldots$ such that $(s_i, a_i, s_{i+1}) \in \Delta$ for all $i \geq 1$. We will also consider (finite) runs on finite words, which are defined analogously. A run is *initial* if it starts with the initial state, i.e., if $s_1 = s_0$. An NBA $\mathcal{A}$ accepts a given ω-word α if there exists an initial run of $\mathcal{A}$ on α that visits a final state infinitely often. The ω-language *recognized* by $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \{\alpha \in \Sigma^{\omega} \mid \alpha \text{ is accepted by } \mathcal{A}\}$ (see [15]).

3.2. Programs as trees

Our synthesis procedure is based on the fact that programs can be viewed as trees over a ranked alphabet. A *ranked alphabet* is a finite alphabet Σ, partitioned into sets $\Sigma_0, \Sigma_1, \ldots, \Sigma_m$. A tree node that is labeled by a symbol a with rank i, i.e., $a \in \Sigma_i$, must have precisely i child nodes.

```
while true do {
     input (b1);
     b2 := b2 ∨ b1 ;
     output (b2)
}
```

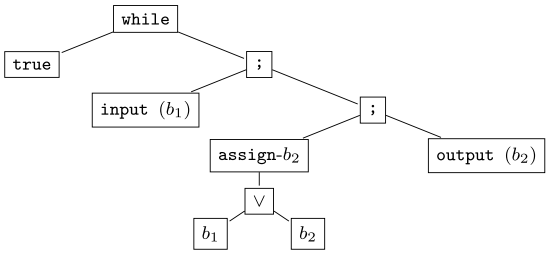

Fig. 3. Example: a program and its tree representation.

Definition 3.1 *(Tree).* Let $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_m$ be a ranked alphabet. The set T_Σ of finite trees over Σ is inductively defined as follows:

- Every symbol $a \in \Sigma_0$ is a tree, i.e., $a \in T_\Sigma$.
- If $t_1, \ldots, t_i \in T_\Sigma$ and $a \in \Sigma_i$ then $a(t_1, \ldots, t_i) \in T_\Sigma$.

Programs over a set B of Boolean variables can be interpreted as trees over the ranked alphabet $\Sigma^B_{\text{prog}} = \Sigma^B_0 \cup \Sigma^B_1 \cup \Sigma^B_2 \cup \Sigma^B_3$, where

- $\Sigma^B_0 = \{\texttt{true}, \texttt{false}\} \cup B \cup \left\{ \texttt{input}\ \vec{b} \mid \vec{b} \in B^{N_\text{in}} \right\} \cup \left\{ \texttt{output}\ \vec{b} \mid \vec{b} \in B^{N_\text{out}} \right\}$,
- $\Sigma^B_1 = \{\neg\} \cup \{\texttt{assign}{-}b \mid b \in B\}$,
- $\Sigma^B_2 = \{\vee, \wedge\} \cup \{\text{“;”}, \texttt{while}\}$, and
- $\Sigma^B_3 = \{\texttt{if}\}$.

The tree representation *tree*(p) of a program p, also known as the syntax tree of p, is defined as follows:

Tree representation of expressions:

- $\mathit{tree}(\texttt{true}) = \texttt{true}$
- $\mathit{tree}(\texttt{false}) = \texttt{false}$
- $\mathit{tree}(b) = b \quad (\text{for } b \in B)$
- $\mathit{tree}(\neg e_1) = \neg(\mathit{tree}(e_1))$
- $\mathit{tree}(e_1 \wedge e_2) = \wedge(\mathit{tree}(e_1), \mathit{tree}(e_2))$
- $\mathit{tree}(e_1 \vee e_2) = \vee(\mathit{tree}(e_1), \mathit{tree}(e_2))$

Tree representation of programs:

- $\mathit{tree}(\texttt{input}\ \vec{b}) = \texttt{input}\ \vec{b}$
- $\mathit{tree}(\texttt{output}\ \vec{b}) = \texttt{output}\ \vec{b}$
- $\mathit{tree}(b := e) = \texttt{assign}{-}b\,(\mathit{tree}(e))$
- $\mathit{tree}(p_1; p_2) = ;(\mathit{tree}(p_1), \mathit{tree}(p_2))$
- $\mathit{tree}(\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2) = \texttt{if}\,(\mathit{tree}(e), \mathit{tree}(p_1), \mathit{tree}(p_2))$
- $\mathit{tree}(\texttt{while}\ e\ \texttt{do}\ p_1) = \texttt{while}\,(\mathit{tree}(e), \mathit{tree}(p_1))$

We use programs and their tree representations interchangeably. In particular, we simply write p instead of *tree*(p) to refer to the tree representation of p. Fig. 3 shows an example for a tree representation of a program. To recognize sets of programs, we use deterministic bottom-up tree automata:

Definition 3.2 *(Deterministic tree automaton).* A deterministic bottom-up tree automaton or simply *deterministic tree automaton (DTA)* is a tuple $\mathcal{B} = (Q, \Sigma, \delta, F)$, where

- Q is a finite set of states,
- $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_m$ is a ranked alphabet,
- $\delta: \bigcup_{i=0}^m (Q^i \times \Sigma_i) \to Q$ is a transition function, and
- $F \subseteq Q$ is a set of final states.

The unique run of a DTA on a given tree evaluates the tree by assigning a state to each node in a bottom-up manner. This evaluation is captured by the function $\delta^*: T_\Sigma \to Q$ with $\delta^*(a) := \delta(a)$ for $a \in \Sigma_0$ and $\delta^*(a(t_1, \ldots, t_i)) := \delta(\delta^*(t_1), \ldots, \delta^*(t_i), a)$ for $a \in \Sigma_i$ and $t_1, \ldots, t_i \in T_\Sigma$. The DTA accepts a tree $t \in T_\Sigma$ if $\delta^*(t) \in F$.

For a given DTA $\mathcal{B} = (Q, \Sigma, \delta, F)$ over $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_m$, we can perform an *emptiness test* in polynomial time: We start by determining the set of states $Q_0 = \{\delta(a) \mid a \in \Sigma_0\}$ that are reachable via trees of height 0. In addition, we store for each of these states q a corresponding tree t_q as a witness for reachability. Formally, for $q \in Q_0$, we let $t_q = a$ for some $a \in \Sigma_0$ with $\delta(a) = q$.

Then, for $\ell = 1, 2, \ldots$, we compute the set $Q_\ell = \{\delta(q_1, \ldots, q_i, a) \mid i \in \{1, \ldots, m\},\ q_1, \ldots, q_i \in Q_{\ell-1},\ a \in \Sigma_i\}$ of states that are reachable via trees of height ℓ. Again, we store a corresponding tree (of height ℓ) along with each of the newly added states $q \in Q_\ell \setminus Q_{\ell-1}$, by letting $t_q = a(t_{q_1}, \ldots, t_{q_i})$ for some $a \in \Sigma_i$, $q_1, \ldots, q_i \in Q_{\ell-1}$ with $\delta(q_1, \ldots, q_i, a) = q$. Note that t_q can be represented using pointers to the previously constructed subtrees, preventing an exponential blow-up.

We stop as soon as we have either found a final state or we have $Q_\ell = Q_{\ell-1}$. In the latter case, we have computed all reachable states without finding a final state, so the language recognized by $\mathcal{B}$ is empty. In the former case, we have found a reachable state $q \in F$ and can present the tree t_q as a witness for nonemptiness. Since we consider trees with increasing height in each step, this tree t_q is minimal with respect to its height among all the trees accepted by $\mathcal{B}$.

Note that this algorithm will always terminate after at most $|Q|$ steps, since we add at least one state to the set of reachable states in every step. Furthermore, each step can be performed in time polynomial in $|Q|$.

3.3. Outline of the synthesis procedure

We will show the following theorem:

Theorem 1. *Let B be a finite set of Boolean variables, let $k \in \mathbb{N}$ and let $\mathcal{A}_{\overline{R}}$ be a nondeterministic Büchi automaton recognizing the complement of a specification $R \subseteq (\Sigma_{\text{in}} \times \Sigma_{\text{out}})^\omega$. We can construct a DTA that accepts a tree p iff p is a reactive program over B with k-bounded delay and $\langle\!\langle p \rangle\!\rangle \subseteq R$. Moreover, the size of this DTA is doubly exponential in $|B|$ and k and exponential in the size of $\mathcal{A}_{\overline{R}}$.*

An emptiness test of the resulting DTA yields a solution to the synthesis problem. In particular, we can extract a program of minimal size with respect to the height of the corresponding tree.

We obtain the desired tree automaton by intersecting three DTAs: The first DTA $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$ recognizes the set of programs over B whose k-bounded computations satisfy the specification R. That means, a program p is accepted iff $\langle\!\langle p \rangle\!\rangle_k \subseteq R$. The second DTA $\mathcal{B}_{\text{reactive}}(B)$ recognizes the reactive programs over B. Finally, we use a third DTA $\mathcal{B}_{\text{delay}}(B, k)$ to recognize the programs over B with k-bounded delay. In the following, we focus on the construction of $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$ and only give a brief description of the other two DTAs, which can be constructed in a similar way.

3.4. Co-execution signatures

The DTA $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$ evaluates a given program p in a bottom-up manner, thereby assigning one of its states to each node of the program tree. The state reached at the root node must provide enough information to decide whether $\langle\!\langle p \rangle\!\rangle_k \subseteq R$, or equivalently, whether $\langle\!\langle p \rangle\!\rangle_k \cap \mathcal{L}(\mathcal{A}_{\overline{R}}) = \emptyset$. To that end, we are interested in the possible runs of $\mathcal{A}_{\overline{R}}$ on the input/output sequences generated by the program. Thus, we consider pairs (ϱ, π) consisting of a (finite or infinite) program computation ϱ and a corresponding run π of $\mathcal{A}_{\overline{R}}$, which we call (finite or infinite) *co-executions*. Intuitively, $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$ inductively computes a representation of the possible co-executions of a given program and $\mathcal{A}_{\overline{R}}$. We define these representations, called *co-execution signatures*, in the following.

The beginning and end of a co-execution can be indicated by a valuation of the program variables and a state of $\mathcal{A}_{\overline{R}}$. However, we have to consider the following: The input sequence of a finite computation might be longer or shorter than its output sequence, but a finite run of $\mathcal{A}_{\overline{R}}$ only consumes input and output sequences of the same length. The suffix of the input/output sequence after the end of the shorter sequence, is hence still waiting to be consumed by $\mathcal{A}_{\overline{R}}$. We call this suffix the *overhanging sequence*. Thus, we indicate the start and end of a co-execution by tuples of the form $\gamma = (\sigma, s, u, v)$, called *co-configurations*, where σ is a variable valuation, s is a state of $\mathcal{A}_{\overline{R}}$ and $(u, v) \in \left(\Sigma_{\text{in}}^* \times \{\varepsilon\}\right) \cup \left(\{\varepsilon\} \times \Sigma_{\text{out}}^*\right)$ is an overhanging sequence. More precisely, since we are only interested in k-bounded computations, we only consider co-configurations with $|u| \le k$ and $|v| \le k$.[1] We denote the set of these co-configurations by C_k. Note that C_k depends on the set of variables B and the specification automaton $\mathcal{A}_{\overline{R}}$, but we omit them in our notation for the sake of readability.

We say that a co-execution (ϱ, π) starting with a co-configuration $\gamma = (\sigma, s, u, v)$ is k-bounded if the computation ϱ is k-bounded for the start difference $|u| - |v|$, i.e., if $\mathit{delay}(\varrho, |u| - |v|) \le k$. A finite co-execution is called *complete* if the program terminates at the end of the computation. The *finite co-execution signature* $\mathit{Sig}^{\text{fin}}(p, \mathcal{A}_{\overline{R}}, k) \subseteq C_k \times \{0, 1\} \times C_k$ of a program p (with respect to $\mathcal{A}_{\overline{R}}$) is defined as follows: We have $(\gamma, f, \gamma') \in \mathit{Sig}^{\text{fin}}(p, \mathcal{A}_{\overline{R}}, k)$ iff there exists a complete k-bounded co-execution (ϱ, π) that starts with the co-configuration γ and ends with γ' such that the run π of $\mathcal{A}_{\overline{R}}$ visits a final state iff $f = 1$. (Note: We say that π visits a final state if some state in π, *excluding the very first state*, is a final state.) The *infinite co-execution signature* $\mathit{Sig}^{\infty}(p, \mathcal{A}_{\overline{R}}, k) \subseteq C_k$ of p is a set of co-configurations with $\gamma \in \mathit{Sig}^{\infty}(p, \mathcal{A}_{\overline{R}}, k)$ iff there exists an infinite k-bounded co-execution starting with γ such that the run of $\mathcal{A}_{\overline{R}}$ visits a final state infinitely often.

[1] For programs that read input and write output in strict alternation (starting with an input statement) as considered in [9], we only need co-configurations with $|u| \le 1$ and $v = \varepsilon$.

3.5. Inductive computation of co-execution signatures

After these preparations, we can now construct the DTA $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\bar{R}}) = (Q_{\text{sat}}, \Sigma^B_{\text{prog}}, \delta_{\text{sat}}, F_{\text{sat}})$ in such a way that

- for any expression e over B: $\delta^*_{\text{sat}}(e) = \{\sigma \in \mathit{Val}_B \mid [\![e]\!](\sigma) = 1\}$,
- for any program p over B: $\delta^*_{\text{sat}}(p) = \left(\mathit{Sig}^{\text{fin}}(p, \mathcal{A}_{\bar{R}}, k), \mathit{Sig}^{\infty}(p, \mathcal{A}_{\bar{R}}, k)\right)$.

The set of states is therefore $Q_{\text{sat}} = Q_{\text{expr}} \cup Q_{\text{prog}} \cup \{\bot\}$, where

- $Q_{\text{expr}} = 2^{\mathit{Val}_B}$ and
- $Q_{\text{prog}} = 2^{C_k \times \{0,1\} \times C_k} \times 2^{C_k}$.

The state $\bot$ is used to indicate syntactically invalid programs. Note that the number of states is doubly exponential in the number of variables, doubly exponential in k and exponential in the size of $\mathcal{A}_{\bar{R}}$. For a fixed k, this matches the complexity of Madhusudan's construction [9].

In order to define the set of final states F_{sat}, let σ_0 be the initial variable valuation (where all variables have the value 0) and let s_0 be the initial state of $\mathcal{A}_{\bar{R}}$. Then $(\sigma_0, s_0, \varepsilon, \varepsilon) \in \mathit{Sig}^{\infty}(p, \mathcal{A}_{\bar{R}}, k)$ iff there is an initial k-bounded computation of p such that some initial run of $\mathcal{A}_{\bar{R}}$ on the resulting input/output sequence visits a final state infinitely often, i.e., $\langle\!\langle p \rangle\!\rangle_k \nsubseteq R$. Hence, we let

$$F_{\text{sat}} = \left\{ (Z^{\text{fin}}, Z^{\infty}) \in Q_{\text{sat}} \mid (\sigma_0, s_0, \varepsilon, \varepsilon) \notin Z^{\infty} \right\}.$$

We will now define the transition relation δ_{sat}. In the following, we assume that $V, V_1, V_2 \in Q_{\text{expr}}$ and $(Z^{\text{fin}}_1, Z^{\infty}_1)$, $(Z^{\text{fin}}_2, Z^{\infty}_2) \in Q_{\text{prog}}$.

■ *Transitions to evaluate expressions:*

- $\delta_{\text{sat}}(\mathtt{true}) = \mathit{Val}_B$
- $\delta_{\text{sat}}(\mathtt{false}) = \emptyset$
- $\delta_{\text{sat}}(b) = \{\sigma \in \mathit{Val}_B \mid \sigma(b) = 1\}$
- $\delta_{\text{sat}}(V_1, V_2, \wedge) = V_1 \cap V_2$
- $\delta_{\text{sat}}(V_1, V_2, \vee) = V_1 \cup V_2$
- $\delta_{\text{sat}}(V, \neg) = \mathit{Val}_B \setminus V$

To define the rest of the transition function, we have to show how the co-execution signatures of a program can be determined in an inductive way. Note that the infinite co-execution signature is empty for all atomic programs (because they terminate immediately).

■ *Transitions for assignments:* A computation of a program $p =$ "$b := e$" changes the variable valuation according to the value of the expression e. Any corresponding run of $\mathcal{A}_{\bar{R}}$ has length 0 (and hence does not visit a final state) since no input is read and no output is written by the program. Thus, the overhanging sequence at the beginning and end of the co-execution is the same. We obtain:

$$\begin{aligned} &\delta_{\text{sat}}(V, \mathtt{assign}{-}b) = (Z^{\text{fin}}, \emptyset) \\ &\text{with} \quad (\gamma, f, \gamma') \in Z^{\text{fin}} \quad \text{iff} \quad \gamma = (\sigma, s, u, v) \in C_k, \\ &\qquad\qquad\qquad\qquad\qquad\quad \gamma' = (\sigma[b/c], s, u, v) \text{ with } c = 1 \text{ iff } \sigma \in V, \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{and } f = 0. \end{aligned}$$

■ *Transitions for input statements:* A computation of a program $p =$ "$\mathtt{input}\ \vec{b}$" changes the variable valuation according to the input symbol. If there is an overhanging output symbol, which is yet to be consumed by $\mathcal{A}_{\bar{R}}$, a corresponding run of $\mathcal{A}_{\bar{R}}$ will proceed via the first overhanging output symbol and the current input symbol. Otherwise, the input symbol is appended to the overhanging sequence. For a formal definition, let S be the set of states and Δ be the transition relation of $\mathcal{A}_{\bar{R}}$:

$\delta_{\text{sat}}(\mathtt{input}\ \vec{b}) = (Z^{\text{fin}}, \emptyset)$ with $(\gamma, f, \gamma') \in Z^{\text{fin}}$ iff one of the following holds:

1. $\gamma = (\sigma, s, \varepsilon, \vec{c}_{\text{out}} \cdot w) \in C_k$ with $\vec{c}_{\text{out}} \in \Sigma_{\text{out}}$, $w \in \Sigma^{<k}_{\text{out}}$, and
   $\gamma' = (\sigma[\vec{b}/\vec{c}_{\text{in}}], s', \varepsilon, w)$ for some $\vec{c}_{\text{in}} \in \Sigma_{\text{in}}$ with $\left(s, (\vec{c}_{\text{in}}, \vec{c}_{\text{out}}), s'\right) \in \Delta$, and $f = 1$ iff $s' \in F$,
2. $\gamma = (\sigma, s, u, \varepsilon) \in C_k$ with $u \in \Sigma^{<k}_{\text{in}}$, and
   $\gamma' = (\sigma[\vec{b}/\vec{c}_{\text{in}}], s, u \cdot \vec{c}_{\text{in}}, \varepsilon)$ for some $\vec{c}_{\text{in}} \in \Sigma_{\text{in}}$, and $f = 0$.

■ *Transitions for output statements:* A computation of a program $p =$ "$\mathtt{output}\ \vec{b}$" does not change the variable valuation. Analogously to the case of an input statement, a corresponding run of $\mathcal{A}_{\bar{R}}$ will consume the first overhanging input symbol and the current output symbol. If no overhanging input symbol exists, the output symbol is appended to the overhanging sequence.

$\delta_{\mathrm{sat}}(\texttt{output}\ \vec{b}) = (Z^{\mathrm{fin}}, \emptyset)$ with $(\gamma, f, \gamma') \in Z^{\mathrm{fin}}$ iff one of the following holds:

1. $\gamma = (\sigma, s, \vec{c}_{\mathrm{in}} \cdot w, \varepsilon) \in C_k$ with $\vec{c}_{\mathrm{in}} \in \Sigma_{\mathrm{in}}$, $w \in \Sigma_{\mathrm{in}}^{<k}$, and $\gamma' = (\sigma, s', w, \varepsilon)$ with $\big(s, (\vec{c}_{\mathrm{in}}, \sigma(\vec{b})), s'\big) \in \Delta$, and $f = 1$ iff $s' \in F$,
2. $\gamma = (\sigma, s, \varepsilon, v) \in C_k$ with $v \in \Sigma_{\mathrm{out}}^{<k}$, and $\gamma' = (\sigma, s, \varepsilon, v \cdot \sigma(\vec{b}))$, and $f = 0$.

■ *Transitions for concatenation:* Co-executions for programs of the form "$p_1;p_2$" consist of a co-execution of p_1 and $\mathcal{A}_{\bar{R}}$ and a subsequent co-execution of p_2 and $\mathcal{A}_{\bar{R}}$. Therefore, we have

$\delta_{\mathrm{sat}}\big((Z_1^{\mathrm{fin}}, Z_1^{\infty}), (Z_2^{\mathrm{fin}}, Z_2^{\infty}), ;\big) = (Z^{\mathrm{fin}}, Z^{\infty})$ with

- $(\gamma, f, \gamma') \in Z^{\mathrm{fin}}$ iff there exist γ'', f_1, f_2 such that $(\gamma, f_1, \gamma'') \in Z_1^{\mathrm{fin}}$, $(\gamma'', f_2, \gamma') \in Z_2^{\mathrm{fin}}$ and $f = \max\{f_1, f_2\}$,
- $\gamma \in Z^{\infty}$ iff either $\gamma \in Z_1^{\infty}$, or there exist γ', f such that $(\gamma, f_1, \gamma') \in Z_1^{\mathrm{fin}}$ and $\gamma' \in Z_2^{\infty}$.

■ *Transitions for selection:* Consider a program "$\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2$". A co-execution of such a program is either a co-execution of p_1 and $\mathcal{A}_{\bar{R}}$ such that the variable valuation at the beginning of the co-execution satisfies the condition e, or a co-execution of p_2 and $\mathcal{A}_{\bar{R}}$ such that the variable valuation at the beginning violates the condition. Hence, we let

$\delta_{\mathrm{sat}}\big(V, (Z_1^{\mathrm{fin}}, Z_1^{\infty}), (Z_2^{\mathrm{fin}}, Z_2^{\infty}), \texttt{if}\big) = (Z^{\mathrm{fin}}, Z^{\infty})$ with

- $\big((\sigma, s, u, v), f, \gamma'\big) \in Z^{\mathrm{fin}}$ iff one of the following holds:
  1. $\sigma \in V$ and $\big((\sigma, s, u, v), f, \gamma'\big) \in Z_1^{\mathrm{fin}}$, or
  2. $\sigma \notin V$ and $\big((\sigma, s, u, v), f, \gamma'\big) \in Z_2^{\mathrm{fin}}$,
- $(\sigma, s, u, v) \in Z^{\infty}$ iff one of the following holds:
  1. $\sigma \in V$ and $(\sigma, s, u, v) \in Z_1^{\infty}$, or
  2. $\sigma \notin V$ and $(\sigma, s, u, v) \in Z_2^{\infty}$.

■ *Transitions for repetition:* The co-execution signatures for a program of the form $p =$ "$\texttt{while}\ e\ \texttt{do}\ p_1$" can be determined by the following reasoning:
A finite co-execution of p and $\mathcal{A}_{\bar{R}}$ can be decomposed into a finite sequence of co-executions of p_1. An infinite co-execution of p can either eventually stay inside a loop iteration forever or perform infinitely many iterations. It can therefore be decomposed either into a finite sequence of co-executions of p_1 followed by an infinite co-execution of p_1, or into a finite sequence of co-executions of p_1 followed by a cycle of co-executions of p_1, leading back to a previous co-configuration.
We can obtain a formal representation of all finite sequences of consecutive co-executions of p_1 that are compatible with the loop condition e in the following way: We consider only those tuples (γ, f, γ') in $\mathit{Sig}^{\mathrm{fin}}(p_1, \mathcal{A}_{\bar{R}}, k)$ where the variable valuation in γ satisfies the loop condition e, and we compute the reflexive transitive closure of the resulting relation. To that end, we define $\mathit{closure}(Z)$ for a relation $Z \subseteq C_k \times \{0,1\} \times C_k$ as the smallest relation $Y \subseteq C_k \times \{0,1\} \times C_k$ such that

- $(\gamma, 0, \gamma) \in Y$ for all $\gamma \in C_k$, and
- $(\gamma, f_1, \gamma') \in Y$, $(\gamma', f_2, \gamma'') \in Z$ implies $(\gamma, \max\{f_1, f_2\}, \gamma'') \in Y$.

Now we can define

$\delta_{\mathrm{sat}}\big(V, (Z_1^{\mathrm{fin}}, Z_1^{\infty}), \texttt{while}\big) = (Z^{\mathrm{fin}}, Z^{\infty})$ with

- $\big(\gamma, f, (\sigma', s', u', v')\big) \in Z^{\mathrm{fin}}$ iff $\big(\gamma, f, (\sigma', s', u', v')\big) \in \mathit{closure}(Z)$ and $\sigma' \notin V$,
- $\gamma \in Z^{\infty}$ iff at least one of the following holds:
  1. there exist $\gamma' = (\sigma', s', u', v') \in C_k$, $f \in \{0,1\}$ such that $(\gamma, f, \gamma') \in \mathit{closure}(Z)$, $\sigma' \in V$ and $\gamma' \in Z_1^{\infty}$,
  2. there exist $\gamma' = (\sigma', s', u', v') \in C_k$, $f \in \{0,1\}$ such that $(\gamma, f, \gamma') \in \mathit{closure}(Z)$, $\sigma' \in V$ and $(\gamma', 1, \gamma') \in \mathit{closure}(Z)$,

where $Z = \big\{ \big((\sigma, s, u, v), f, \gamma\big) \in Z_1^{\mathrm{fin}} \mid \sigma \in V \big\}$.

■ *Transitions for syntactically invalid programs:* In all other cases, the tree is not a syntactically correct program or expression, so we set $\delta_{\mathrm{sat}}(\ldots) = \bot$.

3.6. Reactivity and delay signatures

The desired DTA as described in Section 3.3 is constructed using $\mathcal{B}_{\mathrm{sat}}(B, k, \mathcal{A}_{\bar{R}})$ and the two DTAs $\mathcal{B}_{\mathrm{reactive}}(B)$ and $\mathcal{B}_{\mathrm{delay}}(B, k)$, which recognize the set of reactive programs and the set of programs with k-bounded delay over B, respectively. The construction of the latter two DTAs is similar to the construction of $\mathcal{B}_{\mathrm{sat}}(B, k, \mathcal{A}_{\bar{R}})$. However, instead of the co-execution signatures, these DTAs compute the *reactivity signatures* and *delay signatures* of a given program, capturing the relevant information about the possible computations of the program. These signatures can be computed inductively by a DTA, analogously to the case of co-execution signatures. Formal constructions are given in Appendices A and B.

The *finite reactivity signature* $RSig^{\text{fin}}(p)$ of a program p, representing the terminating computations of p, consists of tuples of the form $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma')$, where σ and σ' are the variable valuations at the beginning and end of a computation of p, and $f_{\text{in}}, f_{\text{out}} \in \{0,1\}$ indicate whether the computation involves any input and output statements, respectively. The *infinite reactivity signature* $RSig^{\infty}(p)$ of p consists of those variable valuations that allow for an infinite computation with only a finite number of input statements or output statements. Note that the program p is reactive iff its infinite reactivity signature does not contain the initial variable valuation σ_0 and its finite reactivity signature does not contain any tuples $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma')$ with $\sigma = \sigma_0$.

Delay signatures also come in two variations: The *admissible delay signature* $DSig^{\leq}(p,k)$ of a program p contains tuples of the form $\big((\sigma,d),(\sigma',d')\big)$ with $d, d' \in \{-k,\dots,k\}$, indicating that there exists a terminating computation ϱ that starts with the variable valuation σ and start difference d, ends with the variable valuation σ' and end difference d', and never exceeds the delay bound k, i.e., $delay(\varrho, d) \leq k$. The *inadmissible delay signature* $DSig^{>}(p,k)$ of p consists of tuples (σ,d), with $d \in \{-k,\dots,k\}$, indicating that there exists a (finite or infinite) computation ϱ that starts with the variable valuation σ and start difference d, and exceeds the delay bound, i.e., $delay(\varrho,d) > k$. The program p has k-bounded delay iff its inadmissible delay signature does not contain the tuple $(\sigma_0, 0)$.

4. A lower bound for the size of the tree automata

The size of the DTA constructed in Section 3 is doubly exponential in the given number of Boolean variables that the programs may use. We will now show that this cannot be avoided, even if we use nondeterministic tree automata (NTAs):

Theorem 2. *Let B be a set of m Boolean variables, let $k \in \mathbb{N}$ and let $R \subseteq (\Sigma_{\text{in}} \times \Sigma_{\text{out}})^{\omega}$ be a specification that is realizable by some program over B with k-bounded delay. Let $\mathcal{C}$ be an NTA that accepts a tree p iff p is a reactive program over B with k-bounded delay and $\langle\!\langle p \rangle\!\rangle \subseteq R$. Then $\mathcal{C}$ has at least $2^{2^{m-1}}$ states.*

Proof. Consider a set of Boolean variables $B = \{b_1,\dots,b_m\}$. There are $2^{2^{m-1}}$ functions g of the type $\{0,1\}^{m-1} \to \{0,1\}$. Each of these functions can be implemented by a program p_g that sets b_m to the value $g(\sigma(b_1), \sigma(b_2), \dots, \sigma(b_{m-1}))$, where $\sigma \in Val_B$ is the valuation of the variables before p_g is executed.

An NTA $\mathcal{C}$ as in Theorem 2 must be able to distinguish all of these programs and it needs at least $2^{2^{m-1}}$ states to do so in an adequate way. To see this, we first define for each of the functions g above another program p_g^{check} that checks whether b_m has the correct value according to the function g, and enters an infinite loop without any input statements otherwise. Now we can construct a program p_g^{verified} that executes $p_g; p_g^{\text{check}}$ for all valuations of the variables $b_1, b_2, \dots, b_{m-1}$ and at the end resets all variables to 0. Note that all checks will succeed, so p_g^{verified} will terminate.

Now assume, towards a contradiction, that $\mathcal{C}$ has less than $2^{2^{m-1}}$ states. Moreover, let p_{sat} be a program that satisfies the given specification. Then for every g, the program $p_g^{\text{verified}}; p_{\text{sat}}$ satisfies the specification as well and is therefore accepted by $\mathcal{C}$. Consider, for every function g, an accepting run of $\mathcal{C}$ on $p_g^{\text{verified}}; p_{\text{sat}}$, and in particular, consider the state that is visited at the root of the subtree p_g. For at least two distinct functions h, h', this state must be the same, because there are $2^{2^{m-1}}$ functions but $\mathcal{C}$ has less than $2^{2^{m-1}}$ states.

Therefore, we still obtain an accepting run if we replace the subprogram p_h by $p_{h'}$ in the program $p_h^{\text{verified}}; p_{\text{sat}}$. But the resulting program will run into an infinite loop, because at least one of the checks fails. This is a contradiction to the premise, so $\mathcal{C}$ must indeed have at least $2^{2^{m-1}}$ states. □

5. Bounds for the required number of program variables

5.1. Overview

The programs constructed by the synthesis procedure presented in Section 3 are restricted to a given set of Boolean variables B. However, there may be no program over B satisfying the specification if B is chosen too small. In this section, we investigate how many Boolean variables are needed to satisfy a given specification. We will only consider specifications given in linear temporal logic (LTL). LTL formulae over a set of atomic propositions are constructed according to the following grammar:

$$\varphi ::= true \mid ap \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2,$$

where *ap* stands for an atomic proposition. In our case, an atomic proposition is either an input symbol $a \in \Sigma_{\text{in}}$ or of the form "out $= a$" for some output symbol $a \in \Sigma_{\text{out}}$. For convenience, we also allow the Boolean connectives $\wedge$, $\to$ and $\leftrightarrow$ (defined in the usual way), and we let $F\varphi = true\, U\varphi$ and $G\varphi = \neg F \neg\varphi$. We interpret LTL formulae over infinite input/output sequences $(\alpha, \beta) = (a_1 a_2 a_3 \dots,\ a'_1 a'_2 a'_3 \dots)$ with $a_i \in \Sigma_{\text{in}}$, $a'_i \in \Sigma_{\text{out}}$ using the following satisfaction relation:

$$
\begin{array}{llll}
(\alpha,\beta) & \models \mathit{true} & & \text{for all } (\alpha,\beta),\\
(\alpha,\beta) & \models a & (\text{for } a \in \Sigma_{\text{in}}) & \text{if } a = a_1,\\
(\alpha,\beta) & \models \text{out} = a & (\text{for } a \in \Sigma_{\text{out}}) & \text{if } a = a'_1,\\
(\alpha,\beta) & \models \neg\varphi & & \text{if } (\alpha,\beta) \not\models \varphi,\\
(\alpha,\beta) & \models \varphi_1 \vee \varphi_2 & & \text{if } (\alpha,\beta) \models \varphi_1 \text{ or } (\alpha,\beta) \models \varphi_2,\\
(\alpha,\beta) & \models X\varphi & & \text{if } (a_2a_3\ldots,\ a'_2a'_3\ldots) \models \varphi,\\
(\alpha,\beta) & \models \varphi_1 U \varphi_2 & & \text{if for some } i \geq 1:\ (a_ia_{i+1}\ldots,\ a'_ia'_{i+1}\ldots) \models \varphi_2\\
& & & \text{and for all } 1 \leq j < i:\ (a_ja_{j+1}\ldots,\ a'_ja'_{j+1}\ldots) \models \varphi_1.
\end{array}
$$

It is well known that the size of the smallest Mealy automaton realizing a given LTL specification is at most doubly exponential in the size of the LTL formula [3]. For such a Mealy automaton, we can easily construct an equivalent structured program in the following way: The current state of the automaton is encoded using Boolean variables. The program performs an infinite loop that contains a case distinction (using nested `if/then/else` constructs) encoding the transition function of the automaton. Note that the number of variables needed to represent the current state of the Mealy automaton is at most exponential in the size of the LTL formula. Hence, we obtain the following result:

Theorem 3. *(Consequence of [3].) For any realizable LTL specification φ, there exists a structured program p, over some set of Boolean variables B, that satisfies φ, where $|B|$ is at most exponential in the size of the LTL formula.*

We will show an (almost) matching lower bound. Note that such a lower bound is not necessarily implied by the doubly exponential lower bound for the size of a Mealy automaton realizing an LTL specification (see [8]). This is because the current "state" of a program does not only depend on the valuation of its variables but also on the current location within the program (also known as the program counter).

Before we give an outline of our approach, let us fix some terminology. An (undirected) *graph* $G = (V(G), E(G))$ consists of a set $V(G)$ of vertices and a set $E(G) \subseteq \{\{x, x'\} \mid x, x' \in V(G)\}$ of undirected edges. A *walk* in G is a sequence $\pi = x_1x_2x_3\ldots$ of vertices of G such that $\{x_i, x_i + 1\} \in E(G)$ for $i \geq 1$. In particular, we are interested in the transition graphs of structured programs:

Definition 5.1 *(Transition graph).* Let p be a structured program with the associated IOI machine $\mathit{ioi}(p) = (Q, \Sigma_{\text{in}}, \Sigma_{\text{out}}, \Delta, \mu, Q_{\text{entry}}, Q_{\text{exit}})$. The *transition graph* of p is the graph G_p with

- $V(G_p) = Q$ and
- $E(G_p) = \{\{q, q'\} \mid (q, (a, a'), q') \in \Delta \text{ for some } a \in \Sigma_{\text{in}} \cup \{\varepsilon\},\ a' \in \Sigma_{\text{out}} \cup \{\varepsilon\}\}$.

By slight abuse of terminology, we refer to Q_{entry} and Q_{exit} as the sets of entry states and exit states of G_p.

Our proof for a lower bound for the number of variables is based on the notion of tree-width (see, for example, [16]), which is a measure for the similarity of a graph to a tree. We will construct LTL specifications that can only be satisfied by programs whose transition graphs have "large" tree-width. (More accurately, the required tree-width grows rapidly with the size of the LTL formula.) Since the structure of the control flow of a structured program is very "tree-like", the transition graph of a structured program can only have large tree-width if the program uses a large number of variables, which yields the desired lower bound. More precisely, we proceed in three steps:

1. We construct a family of LTL specifications $(\varphi_n)_{n\in\mathbb{N}}$ whose length is quadratic in n, such that the transition graph of any program satisfying φ_n (stutter-)simulates the hypercube of dimension $d = 2^n$.
2. We show that if a (labeled) graph G_1 is (stutter-)simulated by another graph G_2 then the tree-width of G_2 cannot be smaller than the tree-width of G_1.
3. We show that the tree-width of the transition graph of any program with m Boolean variables is at most $3 \cdot 2^m - 1$.

(A formal definition of the notions of tree-width and stutter simulation will be given later.)

Since the tree-width of the d-dimensional hypercube is exponential in d (see [17]), we can then deduce that the tree-width of the transition graph of any program satisfying φ_n is at least doubly exponential in n. Therefore, any program satisfying φ_n must use an exponential number of Boolean variables.

5.2. Preliminaries: labeled graphs and stutter simulation

First, we need to define what it means for a graph to stutter-simulate another graph. More precisely, we consider *labeled graphs* (G, ℓ), consisting of a graph G with a labeling function $\ell\colon V(G) \to L$ for some set L. We extend the labeling function to sets of vertices $X \subseteq V(G)$ and to walks $\pi = x_1x_2x_3\ldots$ in the following way: $\ell(X) = \{\ell(x) \mid x \in X\}$ and $\ell(\pi) = \{\ell(x_i) \mid i \geq 1\}$.

Definition 5.2 *(Stutter simulation).* A stutter simulation between two labeled graphs (G_1, ℓ_1) and (G_2, ℓ_2) is a relation $\mathfrak{S} \subseteq V(G_1) \times V(G_2)$ such that

1. for all $x \in V(G_1)$, there exists some $y \in V(G_2)$ with $(x, y) \in \mathfrak{S}$, and
2. for all $(x, y) \in \mathfrak{S}$:
   - $\ell_1(x) = \ell_2(y)$ and
   - for all $x' \in V(G_1)$ with $\{x, x'\} \in E(G_1)$, there exists a walk $yy_1y_2 \dots y_n y'$ in G_2 with $(x', y') \in \mathfrak{S}$ and $(x, y_i) \in \mathfrak{S}$ for all $i \in \{1, \dots, n\}$.

We say that a vertex y (stutter-)simulates a vertex x if $(x, y) \in \mathfrak{S}$.

Note that the first condition in the definition of stutter simulation requires that each vertex of the first graph is simulated by some vertex of the second graph. If there exists a stutter simulation $\mathfrak{S} \subseteq V(G_1) \times V(G_2)$ between (G_1, ℓ_1) and (G_2, ℓ_2), we say that (G_2, ℓ_2) (stutter-)simulates (G_1, ℓ_1).

The following proposition will be useful later:

Proposition 1. *Let $\mathfrak{S} \subseteq V(G_1) \times V(G_2)$ be a stutter simulation between two labeled graphs (G_1, ℓ_1) and (G_2, ℓ_2). Let $x \in V(G_1)$ and $y \in V(G_2)$ such that $(x, y) \in \mathfrak{S}$. Then for any x' that is reachable from x via some walk π_1, there exists a vertex $y' \in V(G_1)$ with $(x', y') \in \mathfrak{S}$ such that y' is reachable from y via some walk π_2 with $\ell_2(\pi_2) = \ell_1(\pi_1)$.*

Proof. We proceed by induction over the length of the walk π_1 from x to x'.

Induction base: If $x = x'$ then y is a correct choice for y'.

Induction step: Let $k \geq 1$ be the length of the walk π_1 from x to x'. Then there exists a vertex $x'' \in V(G_1)$ such that x'' is reachable from x via a walk π_1' of length $k - 1$ and $\{x'', x'\} \in E(G_1)$.
By the induction hypothesis, there exists a vertex $y'' \in V(G_2)$ with $(x'', y'') \in \mathfrak{S}$ such that y'' is reachable from y via some walk π_2' with $\ell_2(\pi_2') = \ell_1(\pi_1')$. Because $\mathfrak{S}$ is a stutter simulation, $\{x'', x'\} \in E(G_1)$ implies the existence of a vertex $y' \in V(G_2)$, with $(x', y') \in \mathfrak{S}$, that is reachable from y'' via some walk $y''y_1y_2 \dots y_n y'$ with $(x'', y_i) \in \mathfrak{S}$ for all $i \in \{1, \dots, n\}$. Note that $\ell_2(y''y_1y_2 \dots y_n y') = \{\ell_2(y''), \ell_1(x''), \ell_2(y')\}$. Hence, y' is reachable from y via the walk $\pi_2' y_1 y_2 \dots y_n y'$ with $\ell_2(\pi_2' y_1 y_2 \dots y_n y') = \ell_2(\pi_2') \cup \{\ell_1(x''), \ell_2(y') = \ell_1(x')\} = \ell_1(\pi_1') \cup \{\ell_1(x''), \ell_1(x')\} = \ell_1(\pi_1)$. □

We will be particularly interested in stutter simulations between a graph G_1 whose vertices are labeled by the identity function *id* on $V(G_1)$ and a graph G_2 whose vertices are labeled with vertices of G_1:

Notation. We write $G_1 \preccurlyeq G_2$ if there exists a labeling $\ell \colon V(G_2) \to V(G_1)$ such that (G_1, id) is stutter-simulated by (G_2, ℓ).

5.3. Step one: forcing the program to simulate a hypercube

5.3.1. A family $(\varphi_n)_{n \in \mathbb{N}}$ of specifications

We consider input sequences over the alphabet $\Sigma_{\text{in}} = \{0, 1, \boxplus, \boxminus, \boxed{?}\}$. (These symbols can be encoded using three input bits.) We interpret such an input sequence as a series of instructions to manage a set of words of length n, i.e., a set $S \subseteq \{0, 1\}^n$. An input sequence infix (i.e., a finite segment of an input sequence) of the form $\boxplus w$ with $w \in \{0, 1\}^n$ is interpreted as an instruction to add the word w to the set. Similarly, an infix of the form $\boxminus w$ with $w \in \{0, 1\}^n$ is viewed as an instruction to remove the word w from the set. Finally, an infix of the form $\boxed{?}w$ with $w \in \{0, 1\}^n$ is a query that asks for the membership of w in the set. We call an input sequence *queryless* if it does not contain the symbol $\boxed{?}$.

Each finite prefix v of such an input sequence uniquely determines a set, denoted by S_v, that is obtained from the empty set by adding and removing words according to the instructions in v. More precisely, a word w is in S_v iff there is an occurrence of the instruction $\boxplus w$ in v such that the suffix of v after that occurrence does not contain the instruction $\boxminus w$. We will call such sets S_v *word sets*.

Now we construct an LTL specification over the input alphabet Σ_{in} as defined above and the output alphabet $\Sigma_{\text{out}} = \{0, 1\}$ that requires the output to be 1 infinitely often iff the following two conditions are met:

1. The input sequence contains exactly one query (one occurrence of $\boxed{?}$).
2. The queried word w is in the word set S_v determined by the prefix v of the input sequence up to the query.

This can be expressed by the LTL formula

$$\varphi_n = (\text{“at most one query”} \wedge \text{“positive query”}) \leftrightarrow GF(out = 1)$$

with the following auxiliary formulae (where X^i is used as an abbreviation for $\overbrace{XX\ldots X}^{i \text{ times}}$):

$$\text{“at most one query”} = G\,(\boxed{?} \rightarrow XG(\neg\boxed{?}))$$

$$\text{“positive query”} = F\big(\underbrace{\boxplus \wedge \text{“queried word”}}_{\text{queried word added}} \wedge \underbrace{\neg F(\boxminus \wedge \text{“queried word”})}_{\text{not removed later}}\big)$$

$$\text{“queried word”} = F\,\boxed{?} \wedge \bigwedge_{1 \le i \le n} \Big((X^i 0 \wedge G(\boxed{?} \rightarrow X^i 0)) \vee (X^i 1 \wedge G(\boxed{?} \rightarrow X^i 1))\Big)$$

Note that the length of φ_n is only quadratic in n. Furthermore, φ_n is clearly realizable by a program that keeps track of the current word set using $\mathcal{O}(2^n)$ Boolean variables: The program outputs 0 until a query occurs and then starts outputting 1 if and only if the query was successful. If a second query is encountered, the program starts outputting 0 again.

5.3.2. The specification φ_n and the hypercube

If we view the possible word sets $S \subseteq \{0,1\}^n$ as the vertices of a graph and connect two sets S, S' by an edge iff S' can be obtained from S by following a $\boxplus$- or $\boxminus$-instruction (i.e., by either adding or removing a single word) then we obtain the hypercube of dimension 2^n:

Definition 5.3 *(Hypercube).* The *hypercube of dimension d*, denoted by H_d, is constructed in the following way: Let D be a set containing exactly d elements. The vertices of the hypercube are the possible subsets of D, i.e., $V(H_d) = 2^D$. Two vertices $S, S' \subseteq D$ are connected by an (undirected) edge iff S' can be obtained from S by either adding or removing a single element.

Intuitively, a program satisfying φ_n must keep track of the current word set, determined by the input sequence that has been read so far. This means that at each state of the transition graph of the program, the current word set is uniquely determined and hence we can label each state with such a set. The program must be able to emulate the instructions provided by the input sequence, which amount to movements along the edges of the above-mentioned hypercube of dimension 2^n. This leads us to the following lemma:

Lemma 5.1. *Let p be a structured reactive program that satisfies φ_n. Then $H_{2^n} \preccurlyeq G_p$.*

To prepare for a formal proof of this lemma, we first prove the following proposition:

Proposition 2. *Let p be a structured reactive program satisfying φ_n. Let q be a state of the associated IOI machine $ioi(p)$ such that q is reachable from the initial state q_0 via some queryless finite input sequence v. Then for all (finite) input sequences v' that lead from q_0 to q, we have $S_{v'} = S_v$.*

Proof. Assume that the proposition does not hold. Then there are two input sequences v, v' leading to q with $S_v \neq S_{v'}$.

Let w be an arbitrary word that is in S_v or in $S_{v'}$ but not in both sets. Now, if we append to v and v' a query $\boxed{?}w$ for w, followed by an arbitrary infinite queryless input sequence α, then for one of the resulting infinite input sequences $v\boxed{?}w\alpha$ and $v'\boxed{?}w\alpha$, the program will produce an incorrect output sequence.

More precisely, for one of the sequences, the query must yield a positive result and thus the program must output 1 infinitely often. For the other sequence, the query must yield a negative result, so the program must output 1 only finitely often. But in both cases the (IOI machine of the) program is in state q immediately before the query, so from that point on, the output will be the same for both input sequences and hence the specification will be violated for one of them. □

We can now prove Lemma 5.1:

Proof of Lemma 5.1. We have to show the existence of a labeling $\ell: V(G_p) \rightarrow V(H_{2^n})$ such that (H_{2^n}, id) is stutter-simulated by (G_p, ℓ). (W.l.o.g., we assume that H_{2^n} is constructed from the set $\{0,1\}^n$, i.e., $V(H_{2^n})$ is the powerset of $\{0,1\}^n$.) Recall that the vertices of G_p are the states of the IOI machine $ioi(p)$ and two vertices are connected by an edge iff they are connected by a transition of $ioi(p)$.

We choose the following labeling ℓ for G_p: If a state q is reachable (in $ioi(p)$) from the initial state q_0 via some queryless input sequence v, its label is $\ell(q) = S_v$. Due to Proposition 2, this label is uniquely determined. Otherwise, we choose an arbitrary word set $S \subseteq \{0,1\}^n$ and set $\ell(q) = S$.

We can now define a stutter simulation $\mathfrak{S}$ between H_{2^n} and G_p:

$$\mathfrak{S} = \Big\{ (S, q) \in V(H_{2^n}) \times V(G_p) \;\Big|\; \begin{array}{l}\ell(q) = S \text{ and } q \text{ is reachable from } q_0 \\ \text{via some queryless input sequence}\end{array} \Big\}$$

We will now show that $\mathfrak{S}$ is indeed a stutter simulation. For all word sets $S \in V(H_{2^n})$, there exists some state q such that $(S, q) \in \mathfrak{S}$, because for each word set S, we can construct a queryless finite input sequence v with $S_v = S$ and this finite input sequence leads to some state q with $\ell(q) = S_v$. Thus, condition 1 in the definition of a stutter simulation is satisfied.

Now consider any $(S, q) \in \mathfrak{S}$: We have $id(S) = \ell(q)$ by definition of $\mathfrak{S}$, as required in the definition of stutter simulation.

Furthermore, consider a set S' with $\{S, S'\} \in E(H_{2^n})$. S' can be obtained from S by adding or removing a single word $w \in \{0,1\}^n$. W.l.o.g., we assume $S' = S \cup \{w\}$. Note that q is reachable (in $ioi(p)$) via some queryless input sequence v and $\ell(q) = S_v = S$. Since p is reactive, there must be a computation of $ioi(p)$ from q via the input sequence $\boxplus w$. All the states that occur in this computation before the sequence $\boxplus w$ has been read completely have the same label as q, because for any strict prefix v' of $\boxplus w$, we have $S_{vv'} = S_v = \ell(q)$. However, the state q' that is reached immediately after $\boxplus w$ has been read is reachable via $v\boxplus w$ and hence $\ell(q') = \ell(q) \cup \{w\} = S'$. Thus, also condition 2 in the definition of stutter simulation is satisfied, so $\mathfrak{S}$ is a stutter simulation. Therefore, (H_{2^n}, id) is indeed stutter-simulated by (G_p, ℓ). $\square$

5.4. Interlude: tree-width and the cops and robber game

To prepare for the following steps, we have to define the notion of tree-width. To that end, we recapitulate the cops and robber game introduced in [16], which provides a characterization of the tree-width of a graph.

Definition 5.4 *(Cops-and-robber game).* In the cops-and-robber game, k cops in helicopters try to catch a robber on a graph G. Note that the cops and the robber can observe each other's moves. In the beginning, the cops choose a set $C_0 \in [V(G)]^{\le k}$ of starting vertices. (We write $[V(G)]^{\le k}$ to denote the set of subsets of $V(G)$ that contain at most k vertices.) Then the robber chooses his starting vertex $x_0 \in V(G)$.

The cops and the robber continue to move in alternation: At the beginning of the $(i+1)$th step of a play, we have a position (C_i, x_i), i.e., the cops are located at the vertices C_i and the robber is on vertex x_i. Now the players make their moves:

1. The cops can move around freely in their helicopters, choosing a set $C_{i+1} \in [V(G)]^{\le k}$.
2. The robber runs from x_i to some vertex x_{i+1} along a walk in G.

A cop that stays at his previous vertex, i.e., a cop on a vertex in $C_i \cap C_{i+1}$, lands at that vertex. (However, such a cop may still fly to a different vertex in the next step.) We say that the vertices in $C_i \cap C_{i+1}$ are *controlled* by a cop.

A play of the game is a sequence of positions of the following form:

$$(C_0, x_0)\ (C_1, x_1)\ (C_2, x_2)\ \ldots$$

The cops win if at least one of the following conditions is met:

- At some point, the robber runs to a vertex $x_{i+1} \in C_{i+1}$, which allows the cops to capture him in their next move.
- On his way from a vertex x_i to x_{i+1}, the robber crosses a vertex $x \in C_i \cap C_{i+1}$ controlled by a cop.

Otherwise, the robber wins because he can evade the cops forever.

We can use the cops-and-robber game to define the tree-width of a graph. While tree-width is usually defined using so-called tree decompositions of graphs, the following definition is equivalent, as was shown in [16].

Definition 5.5 *(Tree-width).* Let G be a graph and let k be the least number of cops such that the cops have a winning strategy in the cops-and-robber game on G. The *tree-width* of G is $tw(G) = k - 1$.

5.5. Step two: stutter simulation and tree-width

Lemma 5.2. *Let G_1, G_2 be two graphs with $G_1 \preccurlyeq G_2$. Then $tw(G_1) \le tw(G_2)$.*

Proof. We have to show the following: If the robber has a winning strategy against k cops on the graph G_1 (in the game presented in Definition 5.4) then the robber also has a winning strategy against k cops on the graph G_2.

We have $G_1 \preccurlyeq G_2$, so there exists a labeling $\ell: V(G_2) \to V(G_1)$ such that (G_1, id) is stutter-simulated by (G_2, ℓ), witnessed by some stutter simulation $\mathfrak{S}$. W.l.o.g., we assume that all vertices $y \in V(G_2)$ simulate some vertex $x \in V(G_1)$, i.e., $(x, y) \in \mathfrak{S}$. (Otherwise, we can simply remove all other vertices from G_2, since this will only limit the options of the robber.) Note that this implies $(\ell(y), y) \in \mathfrak{S}$ for all $y \in V(G_2)$, since G_1 is labeled by the identity function.

Assume that the robber has a winning strategy on G_1. We will now construct a winning strategy for the robber on G_2. Intuitively, the fact that G_1 is simulated by G_2 allows the robber to "simulate" his G_1-moves in G_2 and thus evade the cops. More precisely, the G_2-strategy for the robber is defined as follows.

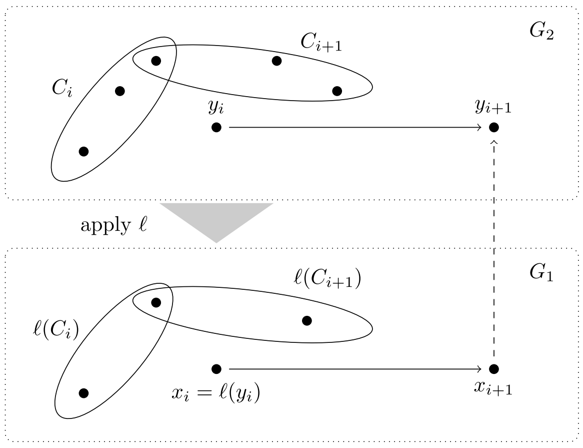

Fig. 4. Construction of the strategy for the robber on G_2 from his winning strategy on G_1.

Winning strategy for the robber on G_2 We start with the choice of the starting vertex for the robber. Let $C_0 \in [V(G_2)]^{\leq k}$ be the set of starting vertices chosen by the cops in G_2. Now let $x_0 \in V(G_1)$ be the starting vertex chosen by the robber in G_1 (according to his winning strategy) if the cops start on the vertices $\ell(C_0)$. In G_2, we let the robber choose an arbitrary starting vertex $y_0 \in V(G_2)$ such that $(x_0, y_0) \in \mathfrak{S}$, i.e., $\ell(y_0) = x_0$. Such a vertex always exists according to our definition of a stutter simulation. Note: Since $\ell(y_0) = x_0 \notin \ell(C_0)$, we have $y_0 \notin C_0$, so this is a safe move for the robber in G_2.

Now consider a position (C_i, y_i) of the game on G_2 and let $C_{i+1} \in [V(G_2)]^{\leq k}$ be the next move of the cops. We have to define an appropriate response for the robber. To that end, consider the position $\big(\ell(C_i), \ell(y_i)\big)$ of the game on G_1. Now suppose that the next move of the cops in G_1 is $\ell(C_{i+1})$. Let $x_{i+1} \in V(G_1)$ be the vertex that is chosen by the robber in response to that (according to his winning strategy on G_1). Furthermore, let π_1 be the walk that the robber uses to get from $x_i = \ell(y_i)$ to x_{i+1}. We determine the next vertex for the robber in G_2 by simulating that move: Since $(\ell(y_i), y_i) \in \mathfrak{S}$, Proposition 1 is applicable. Hence, there exists a vertex $y_{i+1} \in V(G_2)$ with $(x_{i+1}, y_{i+1}) \in \mathfrak{S}$, i.e., with $\ell(y_{i+1}) = x_{i+1}$, such that y_{i+1} is reachable from y_i via some walk π_2 with $\ell(\pi_2) = id(\pi_1)$. We let the robber choose an arbitrary y_{i+1} with this property and move to that vertex. This is illustrated in Fig. 4.

Correctness of the strategy Consider a play prefix that is played on G_2 according to the strategy defined above and ends at the position (C_i, y_i):

$$(C_0, y_0)\,(C_1, y_1)\,\ldots\,(C_i, y_i)$$

By definition of the G_2-strategy, the corresponding play prefix

$$\big(\ell(C_0), \ell(y_0)\big)\;\big(\ell(C_1), \ell(y_1)\big)\;\ldots\;\big(\ell(C_i), \ell(y_i)\big)$$

on G_1 is consistent with the winning strategy for the robber on G_1.

Because x_{i+1} (as chosen above) is the next move of the robber according to his winning strategy, we have $x_{i+1} \notin \ell(C_{i+1})$. Since $\ell(y_{i+1}) = x_{i+1}$, we have $\ell(y_{i+1}) \notin \ell(C_{i+1})$, which implies $y_{i+1} \notin C_{i+1}$.

Furthermore, the walk π_1 (as chosen above) does not contain any vertex controlled by a cop. Since $\ell(\pi_2) = id(\pi_1)$, this means that $\ell(\pi_2) \cap \ell(C_i) \cap \ell(C_{i+1}) = \emptyset$, which implies that π_2 does not contain any vertex in $C_i \cap C_{i+1}$. Hence, the robber escapes if he follows the strategy defined above. □

With Lemma 5.1, we obtain the following corollary:

Corollary 5.1. *Let p be a structured reactive program satisfying φ_n (as defined in Section 5.3.1). Then $tw(G_p) \geq tw(H_{2^n})$.*

To obtain a lower bound for the tree-width of the transition graph of any program satisfying φ_n, we use the following theorem:

Theorem 4. *(See [17]). Let H_d be the hypercube of dimension d. Then we have*

$$tw(H_d) \geq c \cdot \frac{2^d}{\sqrt{d}} \quad \text{for some constant } c > 0.$$

Thus, with Corollary 5.1 we have the following result:

Corollary 5.2. *Let p be a structured reactive program satisfying φ_n (as defined in Section 5.3.1). Then*

$$tw(G_p) \geq c \cdot \frac{2^{2^n}}{\sqrt{2^n}} \quad \text{for some constant } c > 0.$$

5.6. Step three: tree-width of program transition graphs

In the last section, we have established that the transition graph of any program that satisfies the specification φ_n (as defined in Section 5.3.1) has large tree-width (doubly exponential in n). We will now show that a structured program can only have a transition graph with large tree-width if it uses a large number of variables. This is closely related to the fact that the control-flow graphs of structured programs are so-called series-parallel graphs, which have bounded tree-width (see [18,19]).

Theorem 5. *Let p be a structured reactive program over m Boolean variables. The tree-width of its transition graph is at most $3 \cdot 2^m - 1$, i.e., $tw(G_p) \leq 3 \cdot 2^m - 1$.*

Proof. We have to show that $3 \cdot 2^m$ cops have a winning strategy against the robber on the transition graph G_p (in the game specified in Definition 5.4).

We show the following stronger statement by induction over the structure of the program p: There exists a winning strategy for $3 \cdot 2^m$ cops against the robber on G_p such that during any play that is consistent with that strategy, the robber is always *cut off* from the entry and exit states of G_p. We say that the robber is cut off from a vertex x if every walk from the robber's current vertex to x contains a vertex that is controlled by a cop.

Induction base: If p is an atomic program then G_p has exactly $2 \cdot 2^m$ vertices (2^m entry states and 2^m exit states). Since there are $3 \cdot 2^m$ cops, a cop can be sent to each vertex at the beginning of the game. Now the robber has to go to a vertex with a cop, so he is captured in the next step. This also implies that he is cut off from the entry and exit states.

Induction step: We have to consider concatenation, selection and repetition of programs.

$p =$ "$p_1; p_2$": In this case, the transition graph of p can be obtained from the transition graphs of p_1 and p_2 by merging the exit states of G_{p_1} with the corresponding entry states of G_{p_2}. Note that there can only be 2^m entry states in G_{p_2} and thus only 2^m merged states in G_p.
Thus, the cops win the game with the following strategy: In the beginning, they go to the entry and exit states of G_p and to the states that have resulted from merging the exit states of G_{p_1} with the entry states of G_{p_2}. (Note that this requires $3 \cdot 2^m$ cops.) If the robber runs to a vertex from G_{p_1}, the cops play according to their winning strategy on G_{p_1}. Otherwise, if the robber chooses a vertex from G_{p_2}, the cops play according to their winning strategy on G_{p_2}.
By the induction hypothesis, in both cases, the robber cannot escape from the p_1-part or p_2-part of the graph, respectively. Thus, the cops win, and the robber is always cut off from the entry and exit states of G_p.

$p =$ "`if` e `then` p_1 `else` p_2": The transition graph of p is constructed from the transition graphs of p_1 and p_2 by merging the corresponding exit states and specifying entry states according to the branching condition.
The cops therefore have the following winning strategy: In the beginning, they go to the exit states of G_p and those states that were entry states in G_{p_1} or G_{p_2}. (This includes all entry states of G_p and requires $3 \cdot 2^m$ cops.) Then they play their winning strategy on G_{p_1} or G_{p_2}, respectively, depending on which part of G_p the robber chooses in his first move.
The induction hypothesis implies that the cops win and that the robber is cut off from the entry and exit states of G_p.

$p =$ "`while` e `do` p_1": The transition graph of p is obtained by merging the exit states of G_{p_1} with the corresponding entry states. To win the game, the cops can simply play their winning strategy on G_{p_1}: From the induction hypothesis, it follows that the robber is cut off from the entry and exit states of G_p. Hence, the robber can only make moves that are available within G_{p_1}, so the cops win the game on G_p. □

5.7. Putting it all together

We can now prove the following lower bound for the number of variables that are required to satisfy a given LTL specification:

Theorem 6. *There exists a family $(\varphi_n)_{n \in \mathbb{N}}$ of realizable LTL specifications whose size is quadratic in n, such that the least number of Boolean variables used by any structured reactive program satisfying φ_n is $\Omega(2^n)$.*

Proof. Let $(\varphi_n)_{n\in\mathbb{N}}$ be the family of specifications defined in Section 5.3.1. Let p be a structured reactive program over m Boolean variables that satisfies φ_n. Theorem 5 and Corollary 5.2 yield the following inequations:

$$3 \cdot 2^m - 1 \geq tw(G_p) \geq c \cdot \frac{2^{2^n}}{\sqrt{2^n}} \quad \text{for some constant } c > 0.$$

This implies $m \geq 2^n - \mathrm{ld}\left(\sqrt{2^n}\right) + \mathrm{ld}\left(\frac{c}{3}\right)$. Hence, the required number of variables is $\Omega(2^n)$. □

6. Conclusion

This paper provides several contributions to the study of structured reactive programs, which were introduced in [9] as a succinct formalism for the representation of reactive systems:

- We introduced a formal semantics for structured reactive programs.
- We presented a new synthesis algorithm for structured reactive programs with bounded delay, using the elementary concept of deterministic bottom-up tree automata. The resulting tree automaton recognizes the set of programs over a given set of Boolean variables that satisfy the given specification and comply with the given delay bound. The size of the tree automaton is doubly exponential in the number of Boolean variables that are available to the programs, doubly exponential in the delay bound and exponential in the size of the given Büchi automaton representing the complement of the specification.
- Furthermore, we showed a lower bound for the size of any nondeterministic tree automaton that recognizes the set of specification-compliant programs. This lower bound is doubly exponential in the number of Boolean program variables, emphasizing the importance of choosing a small yet still sufficient set of variables.
- Finally, we proved a lower bound for the number of Boolean variables that are needed to realize a given LTL specification by a structured program. This lower bound (almost) matches the exponential upper bound implied by [3].

Two natural measures for the size of a structured program are the number of program variables and the length of the program code. We provided bounds for the number of variables, but estimating the required length of the program code (or the height of the corresponding tree representation) for a given specification is an open question. Moreover, the possible tradeoff between the two measures is still to be investigated.

Acknowledgments

The author would like to thank Wolfgang Thomas for his helpful advice, Konstantinos Stavropoulos, Marcus Gelderie and Sarah Winter for fruitful discussions, and the anonymous referees for their constructive comments. The research leading to the results presented in this paper has received funding from the European Union Seventh Framework Programme (FP7/2007–2013) under Grant Agreement No. 601148 (CASSTING).

Appendix A. Inductive computation of reactivity signatures

In this appendix, we give the detailed construction of the DTA $\mathcal{B}_{\text{react}}(B) = (Q_{\text{react}}, \Sigma^B_{\text{prog}}, \delta_{\text{react}}, F_{\text{react}})$ that computes the reactivity signatures $RSig^{\text{fin}}(p)$ and $RSig^{\infty}(p)$ for a program p, as indicated in Section 3.6. The construction should ensure that

- for any expression e over B: $\delta^*_{\text{react}}(e) = \left\{\sigma \in Val_B \mid [\![e]\!](\sigma) = 1\right\}$,
- for any program p over B: $\delta^*_{\text{react}}(p) = \left(RSig^{\text{fin}}(p), RSig^{\infty}(p)\right)$.

Thus we let $Q_{\text{react}} = Q_{\text{expr}} \cup Q_{\text{prog-react}} \cup \{\bot\}$, where

- $Q_{\text{expr}} = 2^{Val_B}$ (as in the construction of $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\bar{R}})$) and
- $Q_{\text{prog-react}} = 2^{Val_B \times \{0,1\}^2 \times Val_B} \times 2^{Val_B}$.

In order to accept precisely the reactive programs over the variables B, we choose the following set of final states, with σ_0 being the initial variable valuation (where all variables have the value 0):

$$F_{\text{react}} = \left\{ (Z^{\text{fin}}, Z^{\infty}) \in Q_{\text{react}} \mid \sigma_0 \notin Z^{\infty}, \text{ and there is no } (\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}} \text{ with } \sigma = \sigma_0 \right\}.$$

The transitions to evaluate expressions are defined as for the DTA $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\bar{R}})$. To define the rest of the transition function, assume that $V, V_1, V_2 \in Q_{\text{expr}}$ and $(Z_1^{\text{fin}}, Z_1^{\infty}), (Z_2^{\text{fin}}, Z_2^{\infty}) \in Q_{\text{prog-react}}$.

- $\delta_{\text{react}}(V, \texttt{assign}-b) = (Z^{\text{fin}}, \emptyset)$
  with $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}}$ iff $\sigma' = \sigma[b/c]$ with $c = 1$ iff $\sigma \in V$, and $f_{\text{in}} = f_{\text{out}} = 0$.
- $\delta_{\text{react}}(\texttt{input}\ \vec{b}) = (Z^{\text{fin}}, \emptyset)$
  with $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}}$ iff $\sigma' = \sigma[\vec{b}/\vec{c}_{\text{in}}]$ for some $\vec{c}_{\text{in}} \in \Sigma_{\text{in}}$, and $f_{\text{in}} = 1$, $f_{\text{out}} = 0$.
- $\delta_{\text{react}}(\texttt{output}\ \vec{b}) = (Z^{\text{fin}}, \emptyset)$
  with $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}}$ iff $\sigma' = \sigma$ and $f_{\text{in}} = 0$, $f_{\text{out}} = 1$.
- $\delta_{\text{react}}\big((Z_1^{\text{fin}}, Z_1^{\infty}), (Z_2^{\text{fin}}, Z_2^{\infty}), ;\big) = (Z^{\text{fin}}, Z^{\infty})$ with
  - $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}}$ iff there exist σ'', $f_{\text{in}}^1, f_{\text{out}}^1, f_{\text{in}}^2, f_{\text{out}}^2$ such that $(\sigma, f_{\text{in}}^1, f_{\text{out}}^1, \sigma'') \in Z_1^{\text{fin}}$, $(\sigma'', f_{\text{in}}^2, f_{\text{out}}^2, \sigma') \in Z_2^{\text{fin}}$ and $f_{\text{in}} = \max\{f_{\text{in}}^1, f_{\text{in}}^2\}$, $f_{\text{out}} = \max\{f_{\text{out}}^1, f_{\text{out}}^2\}$,
  - $\sigma \in Z^{\infty}$ iff either $\sigma \in Z_1^{\infty}$, or there exist σ', $f_{\text{in}}, f_{\text{out}}$ such that $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z_1^{\text{fin}}$ and $\sigma' \in Z_2^{\infty}$.
- $\delta_{\text{react}}\big(V, (Z_1^{\text{fin}}, Z_1^{\infty}), (Z_2^{\text{fin}}, Z_2^{\infty}), \texttt{if}\big) = (Z^{\text{fin}}, Z^{\infty})$ with
  - $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}}$ iff one of the following holds:
    1. $\sigma \in V$ and $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z_1^{\text{fin}}$, or
    2. $\sigma \notin V$ and $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z_2^{\text{fin}}$,
  - $\sigma \in Z^{\infty}$ iff one of the following holds:
    1. $\sigma \in V$ and $\sigma \in Z_1^{\infty}$, or
    2. $\sigma \notin V$ and $\sigma \in Z_2^{\infty}$.
- Analogously to the construction of the DTA $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$, we define *closure*$(Z)$ for a relation $Z \subseteq Val_B \times \{0,1\}^2 \times Val_B$ in order to describe the reactivity signatures for a program of the form $p =$ "$\texttt{while}\ e\ \texttt{do}\ p_1$". More precisely, *closure*(Z) is the smallest relation $Y \subseteq Val_B \times \{0,1\}^2 \times Val_B$ such that
  - $(\sigma, 0, 0, \sigma) \in Y$ for all $\sigma \in Val_B$, and
  - $\big(\sigma, f_{\text{in}}^1, f_{\text{out}}^1, \sigma'\big) \in Y$, $\big(\sigma', f_{\text{in}}^2, f_{\text{out}}^2, \sigma''\big) \in Z$ implies $\big(\sigma, \max\{f_{\text{in}}^1, f_{\text{in}}^2\}, \max\{f_{\text{out}}^1, f_{\text{out}}^2\}, \sigma''\big) \in Y$.

  Now we let

  $\delta_{\text{react}}\big(V, (Z_1^{\text{fin}}, Z_1^{\infty}), \texttt{while}\big) = (Z^{\text{fin}}, Z^{\infty})$ with
  - $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z^{\text{fin}}$ iff $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in closure(Z)$ and $\sigma' \notin V$,
  - $\sigma \in Z^{\infty}$ iff at least one of the following holds:
    1. there exist $\sigma' \in V \cap Z_1^{\infty}$, $f_{\text{in}}, f_{\text{out}} \in \{0,1\}$ such that $(\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in closure(Z)$,
    2. there exist $\sigma' \in V$, $f_{\text{in}}^1, f_{\text{out}}^1, f_{\text{in}}^2, f_{\text{out}}^2 \in \{0,1\}$ such that $\big(\sigma, f_{\text{in}}^1, f_{\text{out}}^1, \sigma'\big) \in closure(Z)$ and $\big(\sigma', f_{\text{in}}^2, f_{\text{out}}^2, \sigma'\big) \in closure(Z)$ with $\min\{f_{\text{in}}^2, f_{\text{out}}^2\} = 0$,

  where $Z = \big\{ (\sigma, f_{\text{in}}, f_{\text{out}}, \sigma') \in Z_1^{\text{fin}} \mid \sigma \in V \big\}$.
- $\delta_{\text{react}}(\ldots) = \bot$ in all other cases.

Appendix B. Inductive computation of delay signatures

In this appendix, we construct the DTA $\mathcal{B}_{\text{delay}}(B, k) = (Q_{\text{delay}}, \Sigma_{\text{prog}}^B, \delta_{\text{delay}}, F_{\text{delay}})$ computing the delay signatures $DSig^{\leq}(p, k)$ and $DSig^{>}(p, k)$ for a program p, as indicated in Section 3.6, in such a way that

- for any expression e over B: $\delta_{\text{delay}}^*(e) = \big\{ \sigma \in Val_B \mid [\![e]\!](\sigma) = 1 \big\}$,
- for any program p over B: $\delta_{\text{delay}}^*(p) = \big(DSig^{\leq}(p, k), DSig^{>}(p, k)\big)$.

The set of states is therefore $Q_{\text{delay}} = Q_{\text{expr}} \cup Q_{\text{prog-delay}} \cup \{\bot\}$, where

- $Q_{\text{expr}} = 2^{Val_B}$ (as in the construction of $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$) and
- $Q_{\text{prog-delay}} = 2^{(Val_B \times \{-k, \ldots, k\})^2} \times 2^{Val_B \times \{-k, \ldots, k\}}$.

In order to accept precisely the programs with delay $\leq k$ over the variables B, we choose the following set of final states, where σ_0 is the initial variable valuation:

$$F_{\text{delay}} = \big\{ (Z^{\leq}, Z^{>}) \in Q_{\text{delay}} \mid (\sigma_0, 0) \notin Z^{>} \big\}.$$

Again, the transitions to evaluate expressions are defined as for the DTA $\mathcal{B}_{\text{sat}}(B, k, \mathcal{A}_{\overline{R}})$. In the following, we assume that $V, V_1, V_2 \in Q_{\text{expr}}$ and $(Z_1^{\leq}, Z_1^{>}), (Z_2^{\leq}, Z_2^{>}) \in Q_{\text{prog-delay}}$.

- $\delta_{\text{delay}}(V, \texttt{assign}-b) = (Z^{\leq}, \emptyset)$
  with $\big((\sigma, d), (\sigma', d')\big) \in Z^{\leq}$ iff $\sigma' = \sigma[b/c]$ with $c = 1$ iff $\sigma \in V$, and $d' = d \in \{-k, \ldots, k\}$.

- $\delta_{\text{delay}}(\texttt{input } \vec{b}) = (Z^{\leq}, Z^{>})$ with
  - $\big((\sigma, d), (\sigma', d')\big) \in Z^{\leq}$ iff $\sigma' = \sigma[\vec{b}/\vec{c}_{\text{in}}]$ for some $\vec{c}_{\text{in}} \in \Sigma_{\text{in}}$, and $d' = d + 1$ with $d \in \{-k, \dots, k-1\}$,
  - $(\sigma, d) \in Z^{>}$ iff $d = k$.
- $\delta_{\text{delay}}(\texttt{output } \vec{b}) = (Z^{\leq}, Z^{>})$ with
  - $\big((\sigma, d), (\sigma', d')\big) \in Z^{\leq}$ iff $\sigma' = \sigma$ and $d' = d - 1$ with $d \in \{-k+1, \dots, k\}$,
  - $(\sigma, d) \in Z^{>}$ iff $d = -k$.
- $\delta_{\text{delay}}\big((Z_1^{\leq}, Z_1^{>}), (Z_2^{\leq}, Z_2^{>}), ;\big) = (Z^{\leq}, Z^{>})$ with
  - $\big((\sigma, d), (\sigma', d')\big) \in Z^{\leq}$ iff there exist $\sigma'' \in \mathit{Val}_B$, $d'' \in \{-k, \dots, k\}$ such that $\big((\sigma, d), (\sigma'', d'')\big) \in Z_1^{\leq}$ and $\big((\sigma'', d''), (\sigma', d')\big) \in Z_2^{\leq}$,
  - $(\sigma, d) \in Z^{>}$ iff either $(\sigma, d) \in Z_1^{>}$, or there exist $\sigma' \in \mathit{Val}_B$, $d' \in \{-k, \dots, k\}$ such that $\big((\sigma, d), (\sigma', d')\big) \in Z_1^{\leq}$ and $(\sigma', d') \in Z_2^{>}$.
- $\delta_{\text{delay}}\big(V, (Z_1^{\leq}, Z_1^{>}), (Z_2^{\leq}, Z_2^{>}), \texttt{if}\big) = (Z^{\leq}, Z^{>})$ with
  - $\big((\sigma, d), (\sigma', d')\big) \in Z^{\leq}$ iff one of the following holds:
    1. $\sigma \in V$ and $\big((\sigma, d), (\sigma', d')\big) \in Z_1^{\leq}$, or
    2. $\sigma \notin V$ and $\big((\sigma, d), (\sigma', d')\big) \in Z_2^{\leq}$,
  - $(\sigma, d) \in Z^{>}$ iff one of the following holds:
    1. $\sigma \in V$ and $\sigma \in Z_1^{>}$, or
    2. $\sigma \notin V$ and $\sigma \in Z_2^{>}$.
- For a relation $Z \subseteq (\mathit{Val}_B \times \{-k, \dots, k\})^2$, let $\mathit{closure}(Z)$ denote the reflexive transitive closure of Z. Then the delay signatures for a program of the form $p =$ "$\texttt{while}$ e $\texttt{do}$ p_1" can be determined in the following way:

  $\delta_{\text{delay}}\big(V, (Z_1^{\leq}, Z_1^{>}), \texttt{while}\big) = (Z^{\leq}, Z^{>})$ with
  - $\big((\sigma, d), (\sigma', d')\big) \in Z^{\leq}$ iff $\big((\sigma, d), (\sigma', d')\big) \in \mathit{closure}(Z)$ and $\sigma' \notin V$,
  - $(\sigma, d) \in Z^{>}$ iff there exist $\sigma' \in V \cap Z_1^{>}$, $d' \in \{-k, \dots, k\}$ with $\big((\sigma, d), (\sigma', d')\big) \in \mathit{closure}(Z)$,

  where $Z = \big\{ \big((\sigma, d), (\sigma', d')\big) \in Z_1^{\leq} \mid \sigma \in V \big\}$.
- $\delta_{\text{delay}}(\dots) = \bot$ in all other cases.

References

[1] J.R. Büchi, L.H. Landweber, Solving sequential conditions by finite-state strategies, Trans. Am. Math. Soc. 138 (1969) 295–311, http://dx.doi.org/10.2307/1994916.

[2] M.O. Rabin, Automata on Infinite Objects and Church's Problem, American Mathematical Society, 1972.

[3] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proceedings of the 16th Symposium on Principles of Programming Languages, POPL '89, ACM, 1989, pp. 179–190, http://dx.doi.org/10.1145/75277.75293.

[4] O. Kupferman, M.Y. Vardi, Church's problem revisited, Bull. Symb. Log. 5 (2) (1999) 245–263, http://dx.doi.org/10.2307/421091.

[5] M. Gelderie, M. Holtmann, Memory reduction via delayed simulation, in: J. Reich, B. Finkbeiner (Eds.), Proceedings of the International Workshop on Interactions, Games and Protocols, IWIGP '11, in: Electronic Proceedings in Theoretical Computer Science, vol. 50, 2011, pp. 46–60, http://dx.doi.org/10.4204/EPTCS.50.4.

[6] S. Schewe, B. Finkbeiner, Bounded synthesis, in: K.S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (Eds.), Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis, ATVA '07, in: Lecture Notes in Computer Science, vol. 4762, Springer, 2007, pp. 474–488, http://dx.doi.org/10.1007/978-3-540-75596-8_33.

[7] R. Ehlers, Symbolic bounded synthesis, in: T. Touili, B. Cook, P. Jackson (Eds.), Proceedings of the 22nd International Conference on Computer Aided Verification, CAV '10, in: Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 365–379, http://dx.doi.org/10.1007/978-3-642-14295-6_33.

[8] R. Rosner, Modular synthesis of reactive systems, Ph.D. thesis, Weizmann Institute of Science, 1992.

[9] P. Madhusudan, Synthesizing reactive programs, in: M. Bezem (Ed.), Proceedings of Computer Science Logic CSL '11 – 25th International Workshop/20th Annual Conference of the EACSL, in: Leibniz International Proceedings in Informatics, vol. 12, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011, pp. 428–442, http://dx.doi.org/10.4230/LIPIcs.CSL.2011.428.

[10] Y. Lustig, M.Y. Vardi, Synthesis from component libraries, in: L. Alfaro (Ed.), Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS '09, in: Lecture Notes in Computer Science, vol. 5504, Springer, 2009, pp. 395–409, http://dx.doi.org/10.1007/978-3-642-00596-1_28.

[11] M. Holtmann, L. Kaiser, W. Thomas, Degrees of lookahead in regular infinite games, in: L. Ong (Ed.), Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS '10, in: Lecture Notes in Computer Science, vol. 6014, Springer, 2010, pp. 252–266, http://dx.doi.org/10.1007/978-3-642-12032-9_18.

[12] B. Aminof, F. Mogavero, A. Murano, Synthesis of hierarchical systems, in: F. Arbab, P.C. Ölveczky (Eds.), Proceedings of the 8th International Symposium on Formal Aspects of Component Software, FACS '11, in: Lecture Notes in Computer Science, vol. 7253, Springer, 2012, pp. 42–60, http://dx.doi.org/10.1007/978-3-642-35743-5_4.

[13] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, M. Weiglhofer, Specify, compile, run: Hardware from PSL, Electron. Notes Theor. Comput. Sci. 190 (4) (2007) 3–16, http://dx.doi.org/10.1016/j.entcs.2007.09.004.

[14] M. Gelderie, Strategy machines and their complexity, in: B. Rovan, V. Sassone, P. Widmayer (Eds.), Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science, MFCS '12, in: Lecture Notes in Computer Science, vol. 7464, Springer, 2012, pp. 431–442, http://dx.doi.org/10.1007/978-3-642-32589-2_39.

[15] W. Thomas, Languages, automata, and logic, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. 3, Springer, 1997, pp. 389–455, http://dx.doi.org/10.1007/978-3-642-59126-6_7.

[16] P. Seymour, R. Thomas, Graph searching and a min–max theorem for tree-width, J. Comb. Theory, Ser. B 58 (1) (1993) 22–33, http://dx.doi.org/10.1006/jctb.1993.1027.

[17] L. Sunil Chandran, T. Kavitha, The treewidth and pathwidth of hypercubes, Discrete Math. 306 (3) (2006) 359–365, http://dx.doi.org/10.1016/j.disc.2005.12.011.

[18] H.L. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, Theor. Comput. Sci. 209 (1–2) (1998) 1–45, http://dx.doi.org/10.1016/S0304-3975(97)00228-4.

[19] M. Thorup, All structured programs have small tree width and good register allocation, Inf. Comput. 142 (2) (1998) 159–181, http://dx.doi.org/10.1006/inco.1997.2697.