Master Thesis

on

BTSL* Model Checking with Fairness for Reo

Ilham Winata Kurnia

4 December 2008

Revision: 28 December 2008

European Master in Computational Logic
Fakultät Informatik, Technische Universität Dresden
Faculdade de Ciência e Tecnologia, Universidade Nova de Lisboa

Overseeing Professor:	Prof. Dr. rer. nat. habil. Christel Baier
Advisors:	Dipl.-Inf. Sascha Klüppelholz,
	Dipl.-Inf. Tobias Blechmann

BTSL* Model Checking for Reo with Fairness

Ilham Winata Kurnia

4 December 2008

Statement of Academic Honesty

I hereby declare that I have not used any auxiliary means for my thesis work other than that cited in my thesis.

Dresden, 4 December 2008

Ilham Winata Kurnia
Matr. No.: 3368704

Abstract

Reo is an exogenous coordination language designed to describe the interaction between component instances in a component based system. Constraint automata have been introduced to provide a compositional and operational semantics for Reo, and they can serve as the basis for model checking.

This thesis introduces Full Branching Time Stream Logic (BTSL*), a combination of CTL and Dynamic LTL which offers the ability to reason about states and actions in a Reo circuit. A procedure for deciding the validity of BTSL* formulas in a Reo circuit is presented, with special attention is given to the treatment of finite paths and fairness assumptions. A model checker for a restricted version of BTSL* is developed as a module of Vereofy, the model checking toolkit for Reo, and experimental results are presented.

Acknowledgement

I would like to express my sincere gratitude to the following people who made this research possible:

Prof. Dr. Christel Baier for allowing me to work in her energetic research group.

Sascha Klüppelholz and Tobias Blechmann for their guidance, patience, constructive criticism and recommendations that made this study a learning experience of immeasurable value to me.

Joachim Klein for numerous suggestions and discussions, and also for the nifty things he introduced to the group.

Marcus Größer for his in depth review of the thesis.

Frederico Franzosi and Vu Quoc Huy, my friends and colleagues, whom I thank for their patience and constant support during the writing process.

Prof. Dr. Steffen Hölldobler and staff for all the effort to make the European Master in Computational Logic program run well, also with the generous funding of Erasmus Mundus.

friends, lecturers and assistants for helping me enjoy the two years of this master program.

Finally, I would like to give special thanks to my parents and brother for their life-long love and support. Without them, this work could not have been completed.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Component-based software development has made its mark in the last decade with the celebrated concept of middleware. A system developed in this way uses various independent components as building blocks and coordinates these components such that the interaction between the components forms the intended system [BB00]. Most significant challenges which arise in component-based software development are related to component integration and the concept of coordination can be used to solve these issues.

The coordination concept is not restricted to the software development area. Malone and Crowston [MC94] synthesize work done with respect to coordination in many areas, such as economy, biology, linguistics, psychology and computer science. They give a general definition that coordination is "managing dependencies between activities". In the context of software development, the definition given by Carriero and Gelernter [CG92] fits well: "coordination is the process of building programs by gluing together active pieces". They go further by defining coordination models as "the glue that binds separate activities into an ensemble". Coordination languages are used to describe how these activities communicate with each other.

Arbab and Papadopoulos divide coordination languages into two categories: *data-driven* and *control-driven* [AP98]. A data-driven coordination language defines the state of computation at any moment as the actual configurations of the coordinated components and values being received and sent at that moment. In control-driven coordination languages, the values of the data involved in the communication are almost not considered. Instead, the coordination components are separated from the computational components which are treated as "black boxes with clearly defined input/output interfaces". Nevertheless, the applications of these languages are not bound by this categorization. Linda [Gel85, CG92], Laura [Tol98], and GAMMA [BM90] are examples of data-driven coordination languages, while MANIFOLD [AHS93], PCL [DS96], and TOOLBUS [BK96] serve as examples of control-driven coordination languages.

Reo (derived from the greek word $\rho\epsilon\omega$ meaning 'I flow') is a channel-based control-driven coordination language proposed by Arbab [Arb04] which uses channel compositions, resulting in coordinator components called connectors (also known as circuits in some literature), to glue components together. What makes Reo different from other coordination languages is that the composition propagates synchronization and exclusion constraints through its connectors [PC08]. Reo follows the notion of *exogenous coordination* in IWIM [Arb96] which supports separation of responsibilities and anonymous communication. Components need only to concentrate on their work and the connector is not aware of the nature of the coordinated components, which need not know whom they are communicating with.

Constraint automata (CA) have been introduced as an operational semantics of Reo connectors [BSAR06]. The states of a constraint automaton for a Reo connector represent the configurations, while the transitions represent the observable data flow at some point in time and (possibly) a change of configuration. While it is possible to represent any Reo connector directly as a constraint automaton, [BSAR06] provides a way to do this compositionally, the

same way as one would model coordination using Reo.

Since CA are close to standard automata such as the finite automata and ω-automata and also labelled transition systems, one can apply, through some modifications, known techniques to solve verification problems. For example, the problem of deciding whether two Reo connectors are equivalent has been tackled in [BSAR06] and [BB07], while model checking the Reo connectors against temporal logic specifications is presented in [ABBR04, Cla05, KB07, Kle, KB08].

1.1 Contribution

In the context of the Reo Model Checking tool, Vereofy, developed at Technical University Dresden (http://www.vereofy.de), three logics have been developed. They are Branching Time Stream Logic (BTSL) [KB07], $\text{LTL}_{\text{I/O}}$ [Kle], and Alternating-Time Stream Logic [KB08].

In this thesis, I address the model checking problem following the approach given in [KB07], by extending the Branching Time Stream Logic (BTSL) to the CTL*-like BTSL* that can be used to characterize both finite and infinite behavior. I also report on an implementation of a restricted BTSL* model checker.

By applying some modification similar to what is done to CTL*, we can represent fairness assumptions of Reo connectors, which is not discussed in [KB07]. The fairness issue appears from two different sources in Reo connectors. The first one is from the components, where we assume that the processing speed of each component is not infinitely faster than the others. The second source is related to the nondeterminstic selection of data from multiple sources in Reo connectors. By analyzing these two different sources, the fairness definition used in this thesis is similar to the transition-based fairness stated in [LPS81, Nis86, Kwi89, MP92].

1.2 Convention

Throughout this thesis, the following convention is adhered unless stated otherwise. Symbols in capital latin letters (e.g. $AP, Q, \mathbb{Q}$) generally denote sets, while symbols in small latin letters (e.g. a) denote elements of sets. Letter p, possibly with subscripts, is used to represent atomic propositions. Letter c, possibly with subscripts, is also used to label transitions. Letters s, z, b, possibly with subscripts, are used to represent states in automata. Greek letters are used predominantly to represent temporal logic formulas. Single capital letters formatted in sans-serif font or single words formatted in roman slanted small letters denote functions. Other conventions used are introduced as needed.

1.3 Thesis Structure

This thesis is structured as follows. In Chapter 2, Reo and constraint automata are described. This chapter also includes the description of the temporal logics BTSL and DLTL and how to use these logics to perform model checking. In Chapter 3, I describe the syntax and semantics of BTSL*, including how to incorporate fairness and how to model check a Reo circuit given a specification in BTSL*. Chapter 4 provides details of my implementation of a BTSL* model checker along with some experimental results. At the end of these chapters, a brief summary is given. Chapter 5 concludes the thesis and states some future work from this thesis.

Chapter 2

Preliminaries and Related Work

Baier and Katoen state that model checking is a verification technique that explores all possible system states in a brute force manner [BK08]. Via constraint automata (CA), Reo connectors can be verified against a specification usually described using temporal logic. There have been a number of attempts in defining temporal logic to easily specify desired properties of Reo connectors, such as Time Scheduled Data Stream Logic (TSDSL) [ABBR04], Reconfiguration CTL* (ReCTL*) [Cla05], Branching Time Stream Logic (BTSL) [KB07], and Alternating Stream Logic (ASL) [KB08]. Brief descriptions of each are given as follows.

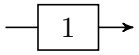
TSDSL is a combination of Linear Temporal Logic (LTL) [Pnu77] and timed regular expressions to reason about timed data streams [BSAR06] of timed constraint automata. ReCTL* combines TSDSL with CTL* and adds a reconfiguration modality to reason about Reo connectors in presence of dynamic reconfiguration where it corresponds to changing the connectors between components. BTSL is a subset of ReCTL* (it excludes the reconfiguration modality and includes only the CTL fragment) except that the model checking procedure does not rely on timed data stream, and instead uses a combination of standard CTL model checking method and automata-based approaches for linear time logics. ASL combines features of BTSL and Alternating-time Temporal Logic (ATL) [AHK02] and it is used to provide a multi-agent semantics for constraint automata.

This chapter is structured as follows. The first two sections cover Reo and CA. The next section elaborates on the description of BTSL and its model checking algorithm. The fourth section describes Dynamic Linear-Time Temporal Logic (DLTL) [HT99] and how to use the tableau-based model checking approach ([GM06]) to verify properties of constraint automata. The model checking algorithms of BTSL and DLTL are the basis to verify properties specified in BTSL*. This chapter is closed with a short summary.

2.1 Reo

Reo is a channel-based coordination language that offers features such as loose coupling among components, support for distribution and mobility of heterogeneous components, exogenous coordination, and dynamic reconfigurability [DA04]. Reo uses *connectors* to coordinate components and connectors are built compositionally from primitive ones, called *channels*. Reo assumes every component instance has one or more active entities where input/output (I/O) operations can be performed. It takes into account only these active entities during the coordination process and abstracts away from the internal interaction which happens within a component instance. In the following sections, we see how channels are composed together to coordinate interactions between component instances.

Table 2.1: Samples of basic channel types

Name	Figure	Description
Sync		This synchronous communication channel accepts data from the source iff it can also write the data to the sink simultaneously.
LossySync		This lossy communication channel always accepts data from the source and if there is a matching I/O operation at the sink end then the data are transfered. Otherwise, the data is dropped.
SyncDrain		An I/O operation pair performed on the synchronous drain channel succeeds only if both write to the ends simultaneously. The data written to this channel is lost.
AsyncSpout(*pat*)		This channel guarantees that two operations on its two ends never succeed simultaneously. The values given out by this channel are specified by the pattern *pat*.
Filter(*pat*)		This communication channel always accepts data item written to the source node as long as it does not match the specified pattern *pat*. Otherwise, it behaves as a synchronous channel.
FIFO1		FIFO1 channel is an asynchronous channel with a bounded buffer of capacity 1 data item. The source always accepts data item until the buffer is filled. The appropriate operations on the sink end of the channel receive the buffer content in FIFO order.

2.1.1 Channels

A channel in Reo has precisely two directed ends and each end is one of the following two kinds: *source* or *sink*. A source end accepts data into its channel, while a sink end writes out data from its channel. Each channel type defines the kind of each channel end, and also how a pair of I/O operations performed at its ends can succeed.

In terms of its ends, channels are categorized into 3 classes: *communication*, *drain*, and *spout* channels. A channel is categorized as a communication (drain and spout, respectively) channel if one end is a source and the other is a sink (both ends are sources and sinks, respectively).

In terms of how I/O-operations performed at their ends may succeed, channels are categorized also into 3 classes: *synchronous*, *asynchronous*, and *lossy*. A channel is called synchronous if the I/O-operation pair can succeed only simultaneously. An asynchronous channel may have a (possibly bounded) buffer to hold data items consumed by its source, but not yet dispensed through its sink. The data transferred in a communication that happens in a lossy channel may be lost if nobody accepts it.

Table 2.1 provides some examples of basic channel types offered in [Arb04]. Custom channels may be created by defining the standard set of operations given in the same paper.

2.1.2 Nodes

A channel's ability to accept or to write data successfully depends on the nodes its ends are attached to and how the data flows on those nodes. Each channel end is attached to exactly one node.

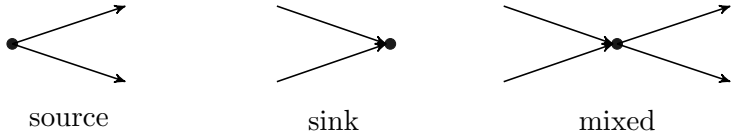


Figure 2.1: Illustration of source, sink, and mixed nodes

Based on the channel ends connected to them, nodes are divided into three types: *source*, *sink* and *mixed*.

Source node (also called input node). A node is a source node if only source channel ends are connected to this node. A write operation on this node succeeds only if all the source ends accept the data item, and thus the written data is propagated to each source end. This node is said to act as a *replicator*.

Sink node (output node). A node is a sink node if only sink channel ends are attached to this node. A read operation from this node succeeds only if there is at least one sink end offering a suitable data item. If there are more than one channel sink offering a suitable data item, then one is picked **nondeterministically**.

Mixed node (internal node). A node is a mixed node if there are at least one sink channel end and one source channel end attached to this node. The behavior of this node is a combination of source node and sink node, repeated infinitely. In every iteration, it nondeterministically selects a suitable data item from sink ends offering that data item, and pumps this item to all source ends attached to it. Note that the condition that all source ends need to accept the data item still holds.

Figure 2.1 gives a graphical illustration of each channel type. Active entities of component instances are only allowed to connect to source and sink nodes and at most one entity can connect to a node at a time.

2.1.3 Connectors

A Reo connector is a multigraph (a pair of nodes can be connected via more than one edge) where the edges are channels, and the nodes are as described in the previous section. The formal definition is described in the following, borrowing some notations from [Cla05].

Let E be a denumerable set of channel ends (small serif letters such as a and b are used to denote its elements), and let the function $kind : E \rightarrow sink, source$ tell the kind of each channel end. A channel is denoted $Ch^{Type}_{\mathsf{a},\mathsf{b}}$, where *Type* denotes the channel type (see section 2.1.1 for examples) and a and b are distinct channel ends. Note that *kind* should be consistent with the type of the corresponding channel.

Definition 1. A Reo connector $\mathcal{C} = (Ch, \mathcal{N})$ consists of a set of channels Ch and a node set $\mathcal{N}$ which satisfies the constraint that for every pair of channels $Ch^{T}_{\mathsf{a},\mathsf{b}}, Ch^{T'}_{\mathsf{c},\mathsf{d}} \in Ch$, a, b, c, and d are distinct. Each node in the node set is a set of channel ends and the $\mathcal{N}$ is a pairwise nonintersecting set of nodes. To simplify the notation, each node is written as a concatentation of all its channel ends, such as abc which denotes a node of three channel ends.

Example 1 (Write cue regulator and exclusive router)**.** Using the definition above, the connector given in Figure 2.2(a) is represented by $(\{Ch^{Sync}_{\mathsf{a},\mathsf{b}}, Ch^{Sync}_{\mathsf{c},\mathsf{d}}, Ch^{SyncDrain}_{\mathsf{e},\mathsf{f}}\}, \{\mathsf{a}, \mathsf{bce}, \mathsf{d}, \mathsf{f}\})$. This connector in essense allows data input into node a to travel to node d if the controller standing by at f wishes so, in which case the data is written to d. Described in another way, it is a

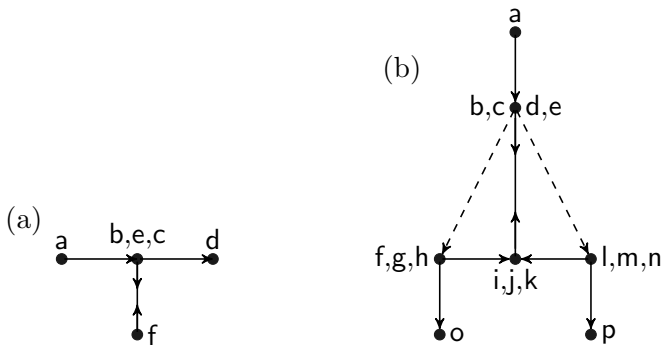


Figure 2.2: Reo connectors: (a) Write-cue regulator; (b) Exclusive router

connector where an entity can count and regulate the data flow between channels ab and cd by the number of write operations on node f.

Connector given in Figure 2.2(b) is called the *exclusive router* as its function is to deliver data from node a to either node o or node p (exclusively) at any given moment, provided nodes o and p are willing to accept data. The selection of the nodes that succeed in passing the data is done nondeterministically, unless only one of the nodes wish to accept the data (which then causes the data to be delivered to that node). The formal representation of this connector is $(\{Ch^{Sync}_{a,b}, Ch^{Sync}_{g,i}, Ch^{Sync}_{h,o}, Ch^{Sync}_{l,k}, Ch^{Sync}_{n,p}, Ch^{LossySync}_{c,f}, Ch^{LossySync}_{e,m}, Ch^{SyncDrain}_{d,j}\}, \{a, bcde, fgh, ijk, lmn, o, p\})$.

The rest of the thesis uses the visual representation of Reo to depict examples instead of the aforementioned textual representation.

To compose complex connectors and manipulate the topology of connectors, Reo defines 5 operations on nodes: *create*, *forget*, *join*, *split*, and *hide*. Operation *create* is used to create a channel and a node for each two new resulting channel ends. Operation *forget* is used to tell Reo that a channel end is not needed anymore. Operations *join* and *split* are duals in the sense that *join* takes two nodes a and b and join them together to create node ab while *split* splits a node ab to two nodes a and b according to where the split is done. Operation *hide* is used to abstract away a node so its topology cannot be modified any further. Figure 2.3 provides an example on how nodes are joined and split.

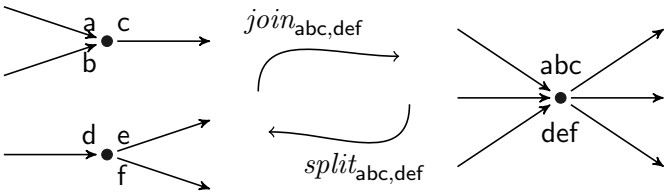


Figure 2.3: Joining nodes abc and def and splitting node abcdef

2.2 Constraint Automata

Constraint automata (CA) [BSAR06] serve as one of possible operational semantics for Reo connectors (other examples can be seen in [MSA06, AR02]). A state in a constraint automaton that represents a Reo connector is a possible configuration (e.g., the content of the buffer of FIFO channels) while the transitions represent possible data flows and effects the flows have on the configuration (e.g., FIFO buffer being emptied).

In the following sections, I summarise the main concepts of CA in relation to Reo connectors. The syntax used in this section follows that from [KB08] which departs slightly from [BSAR06].

In particular, transitions are given in the form of $s \xrightarrow{c} t$, where c is a concurrent I/O operation which tells us what data items are flowing from each node in the connector when we follow this transition.

2.2.1 Concurrent I/O Operations

Recall from Section 2.1.3 that $\mathcal{N}$ is the nonempty finite set of nodes in a Reo connector. Let *Data* be the finite data domain. A *concurrent I/O operation* c is a function from $\mathcal{N}$ to $Data \cup \{\perp\}$, where $\perp$ means "undefined" or "no flow". The set $\mathsf{Nodes}(c)$ is the subset of $\mathcal{N}$ such that for every node $\mathsf{a} \in \mathsf{Nodes}(c)$, $c(\mathsf{a}) \in Data$, while $c(\mathsf{a}) = \perp$ for $\mathsf{a} \notin \mathsf{Nodes}(c)$. $\mathsf{Nodes}(c)$ is also called the set of *active* nodes of c.

The precise meaning of a concurrent I/O operation c is as follows. Each *active* source node a writes data item $c(\mathsf{a})$ to all source channel ends attached to a. Each *active* sink node a takes data item $c(\mathsf{a})$ from one of sink channel ends attached to a. Each mixed node a takes data item $c(\mathsf{a})$ from one of sink channel ends attached to a and simultaneously pushes this item to all source channel ends attached to a. When this I/O operation is performed, there is no data flow at all other nodes $\mathsf{b} \in \mathcal{N} \backslash \mathsf{Nodes}(c)$.

We use a special symbol $c_\emptyset$ to denote the *empty* concurrent I/O operation where $\mathsf{Nodes}(c) = \emptyset$. This operation is used to represent any internal step of some component or a non-observable step where the data flow only occurs at some hidden nodes.

CIO is referred as the set of all possible concurrent I/O operations. Since we assume $\mathcal{N}$ and *Data* to be finite, CIO is also finite. We extend this set by another symbol $\surd$ to allow reasoning about the data flow in a Reo connector that indicates that the data flow has stopped, and denote this extended set by $\mathsf{CIO}_{\surd}$.

2.2.2 I/O Constraints

We would like to be able to group concurrent I/O operations together, and thus provide a concise representation of all possible transitions. We call this representation *I/O constraints* (denoted *ioc*) and we use $CIO(ioc)$ to represent the set of concurrent I/O operations which is a subset of all possible concurrent I/O operations restricted by *ioc*. The following states the syntax of I/O constraints in Backus-Naur Form.

Definition 2 (Syntax of I/O Constraints)**.** For $\mathsf{a} \in \mathcal{N}$, $\mathsf{a}_1, \ldots, \mathsf{a}_k$ of pairwise distinct nodes in $\mathcal{N}$ and $\mathsf{D} \subseteq Data^k$, the set of I/O constraints is defined by:

$$ioc ::= tt \mid ff \mid \mathsf{a} \mid \neg\mathsf{a} \mid (d_{\mathsf{a}_1}, \ldots, d_{\mathsf{a}_k}) \in \mathsf{D} \mid ioc_1 \wedge ioc_2 \mid ioc_1 \vee ioc_2$$

The semantics of I/O constraints is as expected.

$$\begin{array}{lll}
ioc = tt & \text{iff} & CIO(ioc) = \mathsf{CIO} \\
ioc = ff & \text{iff} & CIO(ioc) = \emptyset \\
ioc = \mathsf{a} & \text{iff} & CIO(ioc) = \{c \mid c \in \mathsf{CIO} \wedge \mathsf{a} \in \mathsf{Nodes}(c)\} \\
ioc = \neg\mathsf{a} & \text{iff} & CIO(ioc) = \{c \mid c \in \mathsf{CIO} \wedge \mathsf{a} \notin \mathsf{Nodes}(c)\} \\
ioc = (d_{\mathsf{a}_1}, \ldots, d_{\mathsf{a}_k}) \in \mathsf{D} & \text{iff} & CIO(ioc) = \{c \mid c \in \mathsf{CIO} \wedge \{\mathsf{a}_1, \ldots, \mathsf{a}_k\} \subseteq \mathsf{Nodes}(c) \wedge \\
 & & \qquad (c(\mathsf{a}_1), \ldots, c(\mathsf{a}_k)) \in \mathsf{D}\} \\
ioc = ioc_1 \wedge ioc_2 & \text{iff} & CIO(ioc) = CIO(ioc_1) \cap CIO(ioc_2) \\
ioc = ioc_1 \vee ioc_2 & \text{iff} & CIO(ioc) = CIO(ioc_1) \cup CIO(ioc_2)
\end{array}$$

Additionally, we use simplified notations derived from I/O constraint syntax to intuitively represent a set of I/O operations. For example, $d_{\mathsf{a}} = d_{\mathsf{b}}$ represents I/O constraint $(d_{\mathsf{a}}, d_{\mathsf{b}}) \in \{(d_1, d_2) \in Data^2 \mid d_1 = d_2\}$ (this also implies that a and b must be active nodes) and $d_A = 0$ represents $(d_A) \in \{(0)\}$.

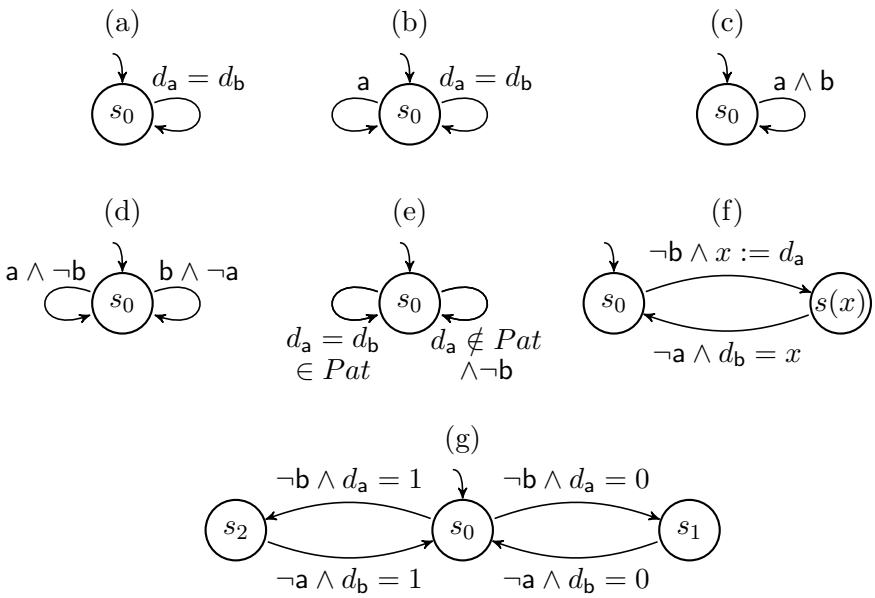


Figure 2.4: CA for sample basic channels in Table 2.1. (a) Sync. (b) LossySync. (c) SyncDrain. (d) AsyncSpout. (e) Filter(Pat). (f) Parameterized FIFO1. (g) FIFO1 of binary domain.

2.2.3 Formal Definition

Definition 3 (Constraint Automata)**.** A constraint automaton is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0, AP, \mathsf{L})$ where Q is a nonempty finite set of states, $\mathcal{N}$ is a finite nonempty set of nodes, $\longrightarrow$ is the transition relation of $\mathcal{A}$ which is a subset of $Q \times \mathsf{CIO} \times Q$, $Q_0 \subseteq Q$ is a nonempty set of initial states, AP is a finite set of atomic propositions, and $\mathsf{L} : Q \to 2^{AP}$ is a labeling function. We write $s \xrightarrow{c} t$ instead of $(s, c, t) \in \longrightarrow$, and we define the set of all I/O-operations enabled in $s \in Q$ as $\mathsf{CIO}(s) \stackrel{\text{def}}{=} \{c \in \mathsf{CIO} \mid s \xrightarrow{c} t \text{ for some } t \in Q\}$.

Constraint automata for each sample basic channels given in Table 2.1 can be seen in Figure 2.4. The channel ends are assumed to be connected to nodes named a and b. Figure 2.4(f) also serves as an example of parameterized constraint automata which, without going into much detail, are an extension to simplify the picture of constraint automata with non-trivial guards [BSAR06]. The transitions are labeled using I/O constraints. We note that atomic propositions cannot be extracted from Reo connectors. However, by labeling states appropriately, we can reason more about its behavior.

As the case with Reo connectors, the rest of the thesis uses the visual illustration of constraint automata instead of the cumbersome textual representation. We also group together transitions that connect the same ordered pair of states and use the corresponding I/O constraint as the label of the transition.

Definition 4 (Terminal States)**.** A state s in a constraint automaton is called a terminal state if data flow may stop in state s. This happens if all enabled concurrent I/O operations require some interaction with a component connected to a source/sink node. The main cause is that components have the liberty to stop providing corresponding write or read operations. This leads to two alternate formal definitions. Formally, state s is called a terminal state iff for all $c \in \mathsf{CIO}(s)$, the node set $\mathsf{Nodes}(c) \neq \emptyset$. Equivalently, state s is terminal iff $c_\emptyset \notin \mathsf{CIO}(s)$.

Note that it is not required that the data flow stops once a terminal state is reached. The data flow continues as long as there is an enabled concurrent I/O operation c where the components involved agree to interact using c as the I/O specification.

Example 2. Figure 2.5 represents a constraint automaton $\mathcal{A} = (\{s_0, s_1, s_2, s_3\}, \{\mathsf{a}, \mathsf{b}\}, \{s_0 \xrightarrow{\{\mathsf{a}:\perp,\mathsf{b}:\perp\}} s_1, s_1 \xrightarrow{\{\mathsf{a}:\perp,\mathsf{b}:\perp\}} s_0, s_0 \xrightarrow{\{\mathsf{a}:\perp,\mathsf{b}:1\}} s_2, s_1 \xrightarrow{\{\mathsf{a}:1,\mathsf{b}:\perp\}} s_2, s_2 \xrightarrow{\{\mathsf{a}:1,\mathsf{b}:\perp\}} s_3, s_2 \xrightarrow{\{\mathsf{a}:\perp,\mathsf{b}:1\}} s_3, s_2 \xrightarrow{\{\mathsf{a}:1,\mathsf{b}:1\}} s_3\}, \{s_0\}, \{term\}, \{s_0 : \emptyset, s_1 : \emptyset, s_2 : term, s_3 : term\})$ with the data domain being a singleton $\{1\}$ (we are interested only if a node is active or not) that features two terminal states. Note that the figure uses I/O constraints as transition labels to represent the automaton $\mathcal{A}$ in a compact manner. States s_2 and s_3 are terminal states since $c_\emptyset$ is not contained in the enabled sets of concurrent I/O operations of s_2 $\mathsf{CIO}(s_2)$ and s_3 $\mathsf{CIO}(s_3)$, while the other two states are not.

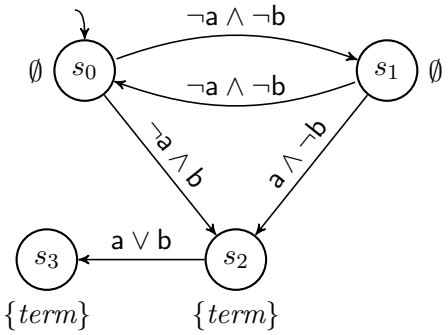


Figure 2.5: Constraint automaton with 2 terminal states

To reason about CA behavior, we need the notion of runs and paths.

Definition 5 (Run)**.** A *run* in a constraint automaton $\mathcal{A}$ is a finite or infinite sequence of instances of consecutive transitions $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \dots$ where $\forall i \geq 0\ (s_i \in Q \wedge s_i \xrightarrow{c_{i+1}} s_{i+1})$. The length of the run $|\theta| \in \mathbb{N} \cup \{\omega\}$ is defined as the number of transitions taken in θ, where ω is the length of an infinite run.

If a run is finite and finishes in a terminal state s_k, then we can append the run with a pseudo-transition $\xrightarrow{\surd} s_k$. We use this pseudo-transition for runs whose data flows stop.

The following notion of maximal run is needed to reason about maximal behavior of constraint automata.

Definition 6 (Maximal Run)**.** A *maximal run* is either an infinite run or a finite run that ends with a transition labeled by $\surd$. We use *MaxRuns*(s) to denote the set of maximal runs starting in s, and simply *MaxRuns* to denote the set of all maximal runs in $\mathcal{A}$ which start from any initial state $q_0 \in Q_0$. We also call a maximal run a *path*.

We use the notion *prefix*(θ, i) to denote the prefix of the run θ of length $i \leq |\theta|$ (i.e. $s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \dots \xrightarrow{c_i} s_i$). Similarly, the notion *suffix*(θ, i) denotes the suffix of the run θ where $i < |\theta|$ (i.e. $s_i \xrightarrow{c_{i+1}} s_{i+1} \xrightarrow{c_{i+2}} \dots$).

We call the extraction of concurrent I/O operations from a run θ its *I/O stream*.

Definition 7 (I/O Stream)**.** The *I/O stream* $ios(\theta)$ of a run θ is a word over $\mathsf{CIO}_{\surd}$ obtained by taking the projection to the labels of the transitions. In other words, if $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \dots$, then $ios(\theta) = c_1, c_2, \dots$. $IOS = \mathsf{CIO}^\omega \cup \mathsf{CIO}^*\surd$ denotes the set of all I/O streams.

Example 3. The following are possible runs from a FIFO1 channel of binary domain (see Figure 2.4).

- $s_0 \xrightarrow{\{\mathsf{a}:0,\mathsf{b}:\perp\}} s_1 \xrightarrow{\{\mathsf{a}:\perp,\mathsf{b}:0\}} s_0$ is a finite run. Its I/O stream is $\{\mathsf{a} : 0, \mathsf{b} : \perp\}, \{\mathsf{a} : \perp, \mathsf{b} : 0\}$.

- $s_0 \xrightarrow{\{a:0,b:\perp\}} s_1 \xrightarrow{\{a:\perp,b:0\}} s_0 \xrightarrow{\{a:0,b:\perp\}} s_1 \xrightarrow{\surd} s_1$ is a maximal finite run. Its I/O stream is $\{a:0, b:\perp\}, \{a:\perp, b:0\}, \{a:0, b:\perp\}, \surd$.
- $s_0 \xrightarrow{\{a:0,b:\perp\}} s_1 \xrightarrow{\{a:\perp,b:0\}} s_0 \xrightarrow{\{a:1,b:\perp\}} s_2 \xrightarrow{\{a:\perp,b:1\}} s_0 \ldots$ is a (maximal) infinite run. Its I/O stream is $\{a:0, b:\perp\}, \{a:\perp, b:0\}, \{a:1, b:\perp\}, \{a:\perp, b:1\}, \ldots$.

2.3 Branching Time Stream Logic

The *Branching Time Stream Logic* (BTSL) is an extension of Computational Tree Logic (CTL) [CES86, CGP99] with additional features absorbed from Propositional Dynamic Logic (PDL) [FL79] and Timed Data Stream Logic (TDSL) [BSAR06]. It has been developed in [KB07] to allow reasoning about the control and data flow of a constraint automaton.

2.3.1 Syntax and Semantics

A BTSL signature is a tuple $(AP, \mathcal{N}, Data)$ which consists of a finite nonempty set AP of atomic propositions, a finite nonempty node set $\mathcal{N}$, and a finite nonempty data domain $Data$. BTSL has a two level syntax such that formulas in BTSL are classified into *state formulas* (denoted by capital Greek letters Φ and Ψ) and *path formulas* (denoted by small Greek letter φ). One path formula operator can take an extra argument which is called regular I/O stream expressions (denoted by small Greek letter α).

Definition 8 (BTSL Syntax)**.** The abstract syntax of BTSL is given as follows where $p \in AP$, $c \in \mathsf{CIO}$.

$$\begin{aligned}
\Phi &::= true \mid p \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_1 \mid \exists\ \varphi \mid \forall\ \varphi \\
\varphi &::= \Phi_1\ \mathbf{U}\ \Phi_2 \mid \langle\alpha\rangle\Phi_1 \\
\alpha &::= c \mid stop \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^* \mid \alpha_1;\alpha_2
\end{aligned}$$

Informally, the state formulas and the until operator $\mathbf{U}$ is the same as in CTL. The PDL-like syntax $\langle\alpha\rangle\Phi_1$ (read: sometime after α is performed, Φ_1 holds) is used to represent the condition that the desired path starts with a finite prefix where the data flow is a word in the language $\mathcal{L}_{\mathcal{N}}(\alpha)$ defined by the regular I/O stream expression α.

Definition 9 (BTSL Semantics)**.** Given a constraint automaton $\mathcal{A}$, for a state $s \in Q$, the satisfaction relation $\models$ for state formulas is as follows.

$$\begin{array}{lll}
\mathcal{A}, s \models true & & \\
\mathcal{A}, s \models p & \text{iff} & p \in \mathsf{L}(s) \\
\mathcal{A}, s \models \neg\Phi & \text{iff} & \mathcal{A}, s \not\models \Phi \\
\mathcal{A}, s \models \Phi \wedge \Psi & \text{iff} & \mathcal{A}, s \models \Phi \text{ and } \mathcal{A}, s \models \Psi \\
\mathcal{A}, s \models \exists\ \varphi & \text{iff} & \mathcal{A}, \theta \models \varphi \text{ for some } \theta \in MaxRuns(s) \\
\mathcal{A}, s \models \forall\ \varphi & \text{iff} & \mathcal{A}, \theta \models \varphi \text{ for all } \theta \in MaxRuns(s)
\end{array}$$

For maximal run $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$, the satisfaction relation $\models$ for path formulas is defined as follows.

$$\begin{array}{lll}
\mathcal{A}, \theta \models \Phi\ \mathbf{U}\ \Psi & \text{iff} & \exists\ 0 \le i \le |\theta| :\ (\mathcal{A}, s_i \models \Psi \wedge \forall\ 0 \le j < i :\ \mathcal{A}, s_j \models \Phi) \\
\mathcal{A}, \theta \models \langle\alpha\rangle\Phi & \text{iff} & \exists\ 0 \le i \le |\theta| :\ (\mathcal{A}, s_i \models \Phi \wedge ios(prefix(\theta, i)) \in \mathcal{L}_{\mathcal{N}}(\alpha))
\end{array}$$

The semantics for the regular I/O expressions α is as follows.

$$\begin{array}{lcl}
\alpha = c & \Leftrightarrow & \mathcal{L}_{\mathcal{N}}(\alpha) = \{c\} \\
\alpha = stop & \Leftrightarrow & \mathcal{L}_{\mathcal{N}}(\alpha) = \{\surd\} \\
\alpha = \alpha_1 \cup \alpha_2 & \Leftrightarrow & \mathcal{L}_{\mathcal{N}}(\alpha) = \mathcal{L}_{\mathcal{N}}(\alpha_1) \cup \mathcal{L}_{\mathcal{N}}(\alpha_2)
\end{array}$$

Table 2.2: Derived BTSL operators

$\exists \bigcirc \Phi$	$\equiv$	$\exists\langle \mathsf{CIO}\rangle\Phi$	$\forall \bigcirc \Phi$	$\equiv$	$\neg\exists \bigcirc \neg\Phi$
$\exists\Diamond\Phi$	$\equiv$	$\exists(\mathit{true}\ \mathbf{U}\ \Phi)$	$\forall\Box\Phi$	$\equiv$	$\neg\exists\Diamond\neg\Phi$
$\exists\Box\Phi$	$\equiv$	$\neg\forall(\mathit{true}\ \mathbf{U}\ \neg\Phi)$	$\forall\Diamond\Phi$	$\equiv$	$\neg\exists\Box\neg\Phi$
$\exists(\Phi\ \mathbf{R}\ \Psi)$	$\equiv$	$\neg\forall(\neg\Phi\ \mathbf{U}\ \neg\Psi)$	$\forall(\Phi\ \mathbf{R}\ \Psi)$	$\equiv$	$\neg\exists(\neg\Phi\ \mathbf{U}\ \neg\Psi)$
$\exists[\alpha]\Phi$	$\equiv$	$\neg\forall\langle\alpha\rangle\neg\Phi$	$\forall[\alpha]\Phi$	$\equiv$	$\neg\exists\langle\alpha\rangle\neg\Phi$

The semantics for Kleene star (*) and concatenation ($;$) operations is more elaborate as we need to ensure that the data flow stop symbol $\surd$ only occurs at the end of an I/O stream.

$$\alpha = \alpha_1^* \quad \Leftrightarrow \quad \mathcal{L}_{\mathcal{N}}(\alpha) = \bigcup_{n\geq 0} \mathcal{L}_{\mathcal{N}}(\alpha_1)^n \text{ where } \mathcal{L}_{\mathcal{N}}(\alpha_1)^0 = \{\epsilon\},$$
$$\mathcal{L}_{\mathcal{N}}(\alpha_1)^{i>0} = \{\sigma_1 \dots \sigma_i \mid \forall\ 1 \leq j < i : \sigma_j \in \mathcal{L}_{\mathcal{N}}(\alpha_1) \cap \mathsf{CIO}^*, \sigma_i \in \mathcal{L}_{\mathcal{N}}(\alpha_1)\}$$

$$\alpha = \alpha_1;\alpha_2 \quad \Leftrightarrow \quad \mathcal{L}_{\mathcal{N}}(\alpha_1;\alpha_2) = \{\sigma_1\surd \mid \sigma_1\surd \in \mathcal{L}_{\mathcal{N}}(\alpha_1)\} \cup \{\sigma_1\sigma_2 \mid \sigma_1 \in \mathcal{L}_{\mathcal{N}}(\alpha_1) \cap \mathsf{CIO}^* \wedge \sigma_2 \in \mathcal{L}_{\mathcal{N}}(\alpha_2)\}$$

In addition, we also use $\alpha = \mathsf{CIO}$ to denote $\mathcal{L}_{\mathcal{N}}(\alpha) = \mathsf{CIO}$.

When it is clear from the context which automaton we deal with, we drop $\mathcal{A}$ from $\mathcal{A}, s \models \Phi$ and use the notation $s \models \Phi$. We use the notation $\mathcal{A} \models \Phi$ to state that the constraint automaton $\mathcal{A}$ satisfies Φ. This is fulfilled iff for every state s in Q_0 of $\mathcal{A}$, $s \models \Phi$.

Using this semantics, we can derive other operators, such as *next* step operator $\bigcirc$, *eventually* $\Diamond$, *always* $\Box$, *release* $\mathbf{R}$, and *conditional always* $[\alpha]$ ($[\alpha]\Phi$ is read whenever α is performed, Φ holds). Other propositional operators can be derived using the usual techniques. The equivalences are given in Table 2.2.

Example 4. For a Sync channel with a as the source node and b as the sink node, the BTSL formula $\forall\Box\forall\langle \mathit{stop} \cup d_{\mathsf{a}} = d_{\mathsf{b}}\rangle \mathit{true}$ holds. This formula states that the constraint I/O operations used in all runs must synchronously transmit data items from a to b and runs may end when the components connected to a and b refuse to perform more I/O operations.

For a LossySync channel with the same custom, a similar BTSL formula $\forall\Box\forall\langle \mathit{stop} \cup \mathsf{a}\rangle \mathit{true}$ also holds. This formula expresses that a will always be able to send a message regardless whether b accepts it or not.

Figure 2.6 shows a more complicated Reo connector and its corresponding constraint automaton with some nodes hidden and the data values ignored. This connector simulates a while loop and has 3 internal (hidden) components: an exclusive router (see Figure 2.2(b)), a condition checker and a while body component. The BTSL formula $\exists[in; w_{in}; (w_{out}; w_{in})^*]\mathit{process}$ states that there exists an execution of a while loop such that whenever the data flow reaches w_{in}, the while body does some process.

Using some duality laws, we can construct a BTSL syntax using a small number of operators. This becomes useful in constructing the model checking algorithm for BTSL.

Definition 10 (Existential normal form for BTSL)**.** For $p \in AP$, the set of BTSL formulas in existential normal form (ENF) is given by

$$\Phi ::= \mathit{true} \mid p \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_1 \mid \exists\Phi_1\ \mathbf{U}\ \Phi_2 \mid \exists\Phi_1\ \mathbf{R}\ \Phi_2 \mid \exists\langle\alpha\rangle\Phi_1 \mid \exists[\alpha]\Phi_1$$

Theorem 1 (Existential normal form for BTSL)**.** *For each BTSL formula there exists an equivalent BTSL formula in ENF.*

Proof. The following duality laws allow elimination of the universal path quantifier and thus all BTSL formulas can be translated into the following equivalent BTSL formulas.

Algorithm 2.1 General BTSL model checking

1: **for all** $i \leq |\Phi|$ **do**
2: **for all** $\Psi \in Sub(\Phi)$ with $|\Psi| = i$ **do**
3: **switch** (Ψ):

$true$	:	$Sat(\Psi) := Q$
p	:	$Sat(\Psi) := \{s \in Q \mid p \in L(s)\}$
$p_1 \wedge p_2$	:	$Sat(\Psi) := Sat(p_1) \cap Sat(p_2)$
$\neg p$	:	$Sat(\Psi) := Q \backslash Sat(p)$
$\exists \Psi_1 \ \mathbf{U} \ \Psi_2$	:	$Sat(\exists \Psi_1 \ \mathbf{U} \ \Psi_2) :=$ the smallest subset T of Q such that: 1. $Sat(\Psi_2) \subseteq T$ 2. $s \in Sat(\Psi_1)$ and $Post(s) \cap T \neq \emptyset$ implies $s \in T$
$\exists \Psi_1 \ \mathbf{R} \ \Psi_2$	:	$Sat(\exists \Psi_1 \ \mathbf{R} \ \Psi_2) :=$ the largest subset T of Q such that: 1. $T \subseteq Sat(\Psi_2)$ 2. $s \notin Sat(\Psi_1) \cap Sat(\Psi_2)$ and $Post(s) \cap T = \emptyset$ implies $s \notin T$
$\exists \langle \alpha \rangle \Psi_1$	:	$Sat(\exists \langle \alpha \rangle \Psi_1) := Sat_{\exists \langle . \rangle}(\exists \langle \alpha \rangle \Psi_1)$ (See Theorem 2)
$\exists [\alpha] \Psi_1$	:	$Sat(\exists [\alpha] \Psi_1) := Sat_{\exists [.]}(\exists [\alpha] \Psi_1)$ (See Theorem 2)

end switch
4: $AP := AP \cup \{p_\Psi\}$
5: replace Ψ with p_Ψ
6: **for all** $s \in Sat(\Psi)$ **do**
7: $L(s) := L(s) \cup \{p_\Psi\}$
8: **end for**
9: **end for**
10: **end for**

$$\begin{aligned} \forall(\Phi \ \mathbf{U} \ \Psi) &\equiv \neg\exists(\neg\Phi \ \mathbf{R} \ \neg\Psi) \\ \forall\langle\alpha\rangle\Phi &\equiv \neg\exists[\alpha]\neg\Phi \end{aligned}$$

□

2.3.2 Model Checking Algorithm

The model checking problem for BTSL is to verify for a given constraint automaton $\mathcal{A}$ and a BTSL formula Φ whether $\mathcal{A} \models \Phi$. Model checking BTSL formulas [KB07] is similar to model checking CTL formulas, with an automata-based approach to handle the PDL-like formulas. The basic procedure is related to computing the satisfaction set $Sat(\Phi)$ in a bottom-up fashion. Algorithm 2.1 lists a way to compute $Sat(\Phi)$ for an ENF equivalent of any arbitrary BTSL state formula Φ.

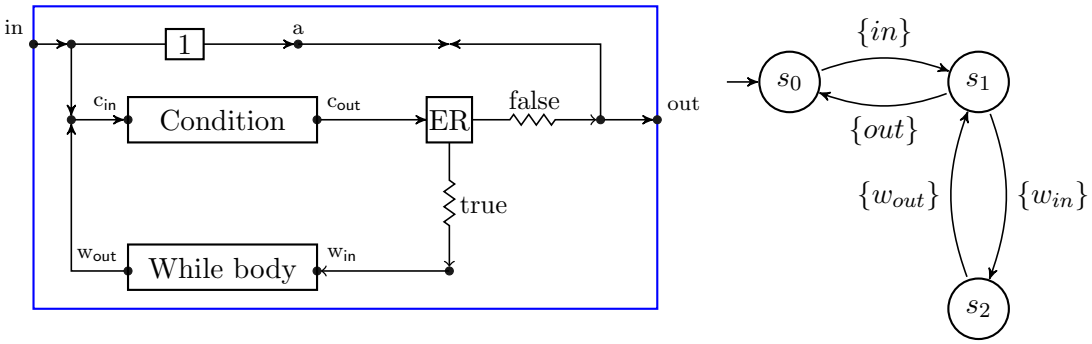


Figure 2.6: While loop Reo connector and its constraint automaton[TVMS07]

One possible approach to handle BTSL formulas of the form $\exists\langle\alpha\rangle\Phi$ and $\exists[\alpha]\Phi$ is by using an automata-based approach. First, we create a nondeterministic finite automaton (NFA) $\mathcal{Y}$ to recognize the language represented by the regular I/O stream expression α. The alphabet of $\mathcal{Y}$ is $\mathsf{CIO}\cup\{\surd\}$. This NFA can also be seen as a constraint automaton $\mathcal{Z} = (Z, \mathcal{N}, \longrightarrow, Z_0, AP, \mathsf{L})$ if we exclude for the moment the special $\surd$ transitions. The atomic proposition AP contains only one element *final*, and the labeling function L maps the states which are final (accept) states of the NFA to *final*. To deal with the $\surd$ exception, we assume Z contains a subset $Z_{\surd}$ such that

(i.) $z \xrightarrow{\surd} z'$ implies $z' \in Z_{\surd}$,

(ii.) $z \xrightarrow{c} z' \wedge z' \in Z_{\surd}$ implies $c = \surd$,

(iii.) there are no outgoing transitions from any state in $Z_{\surd}$.

Second, we need to construct the product of the constraint automaton and the NFA and label the states in the product with two atomic propositions *sat* and *final* such that a state (s, z) is assigned with *sat* if $s \models_{\mathcal{A}} \Phi$ and *final* if *final* $\in \mathsf{L}_{\mathcal{Z}}(z)$. The product of a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow_{\mathcal{A}}, Q_0, AP_{\mathcal{A}}, \mathsf{L}_{\mathcal{A}})$ with $\mathcal{Z} = (Z, \mathcal{N}, \longrightarrow_{\mathcal{Z}}, Z_0, AP_{\mathcal{Z}}, \mathsf{L}_{\mathcal{Z}})$ is $\mathcal{A} \times \mathcal{Z} = (Q \times Z, \mathcal{N}, \longrightarrow_{\mathcal{A}\times\mathcal{Z}}, Q_0 \times Z_0, AP_{\mathcal{A}\times\mathcal{Z}}, \mathsf{L}_{\mathcal{A}\times\mathcal{Z}})$ where

- $\longrightarrow_{\mathcal{A}\times\mathcal{Z}}$ is obtained from the following rules where $s, s' \in Q$ and $z, z' \in Z$:

$$\frac{s \xrightarrow{c}_{\mathcal{A}} s' \wedge z \xrightarrow{c}_{\mathcal{Z}} z' \wedge c \in \mathsf{CIO}}{(s,z) \xrightarrow{c}_{\mathcal{A}\times\mathcal{Z}} (s',z')} \qquad \frac{\text{s is terminal in } \mathcal{A} \wedge z \xrightarrow{\surd}_{\mathcal{Z}} z'}{(s,z) \xrightarrow{\surd}_{\mathcal{A}\times\mathcal{Z}} (s,z')},$$

- $AP_{\mathcal{A}\times\mathcal{Z}} = \{sat, final\}$,

- $\mathsf{L}_{\mathcal{A}\times\mathcal{Z}} : Q \times Z \to AP_{\mathcal{A}\times\mathcal{Z}}$ is the labeling function such that for $(s, z) \in Q \times Z$

$$\mathsf{L}_{\mathcal{A}\times\mathcal{Z}}(s,z) = \begin{cases} \{sat, final\} & \text{if } s \models \Phi \text{ and } final \in \mathsf{L}_{\mathcal{Z}}(z) \\ \{sat\} & \text{if } s \models \Phi \text{ and } final \notin \mathsf{L}_{\mathcal{Z}}(z) \\ \{final\} & \text{if } s \not\models \Phi \text{ and } final \in \mathsf{L}_{\mathcal{Z}}(z) \\ \{\} & \text{otherwise} \end{cases}.$$

The computation of the set of states that satisfy $\exists\langle\alpha\rangle\Phi$ and $\exists[\alpha]\Phi$ is done via the reduction presented in the following theorem.

Theorem 2 (Reduction to CTL [KB07])**.** *Given the product automaton $\mathcal{A} \times \mathcal{Z}$, the following properties hold.*

1. *$\mathcal{A}, s \models \exists\langle\alpha\rangle\Phi$ iff there exists $z_0 \in Z_0$ with $\mathcal{A} \times \mathcal{Z}, (s, z_0) \models \exists\Diamond(sat \wedge final)$*
2. *If $\mathcal{A}$ is deterministic then $\mathcal{A}, s \models \exists[\alpha]\Phi$ iff $\mathcal{A} \times \mathcal{Z}, (s, z_0) \models \exists\Box(sat \vee \neg final)$ where z_0 is an initial state of $\mathcal{Z}$.*

The correctness of the CTL portion of the Algorithm 2.1 can be seen in [CES86] and the rest follows from Theorem 2.

Example 5. The formula $\exists\langle\mathsf{CIO}; d_{\mathsf{b}} = 0\rangle empty$ for a FIFO1 channel with binary domain specifies that there is a way to take a data item 0 from the channel and that makes the buffer empty. To check whether this formula is satisfied, we first create an NFA $\mathcal{Z}$ that recognizes $\mathsf{CIO}; d_{\mathsf{b}} = 0$, as illustrated by the top part of Figure 2.7. The product $\mathcal{A} \times \mathcal{Z}$ is then calculated by the rules given before (the reachable part is portrayed by the bottom part of Figure 2.7), and then the model checker applies the CTL model checking methods for the formula $\exists\Diamond(sat \wedge final)$ on $\mathcal{A} \times \mathcal{Z}$. Since state (s_0, z_2) is reachable from the starting state and in that state $sat \wedge final$ is fulfilled, the channel satisfies the specification.

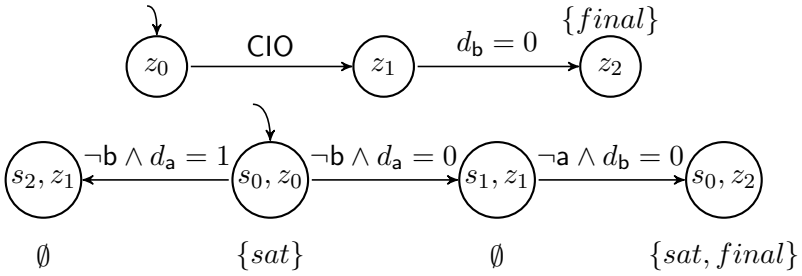


Figure 2.7: NFA for $\mathsf{CIO}; d_\mathsf{b} = 0$ and the reachable part of the product $\mathcal{A} \times \mathcal{Z}$

Remark. If the constraint automaton is deterministic, then the complexity of computing the satisfaction sets of any BTSL formula is polynomial in the size of the constraint automaton and the length of the formula [KB07].

2.4 Dynamic LTL

The famous counterpart of CTL is LTL. Likewise, BTSL also has a counterpart which combines both PDL and LTL, and this combination is called Dynamic LTL (DLTL) [HT99]. Here we extend slightly the notion of a model in [HT99] to a constraint automaton from a pair of an infinite run and a function of each node to a set of atomic propositions. Note that here we assume all runs in the constraint automaton to be infinite.

2.4.1 Syntax and Semantics

As in BTSL, the signature of DLTL is a tuple $(AP, \mathcal{N}, Data)$ which consists of a finite nonempty set AP of atomic propositions, a finite nonempty node set $\mathcal{N}$, and a finite nonempty data domain $Data$. DLTL has a one level syntax where all formulas in DLTL are classified as *path formulas* (denoted by small Greek letter φ). One path formula operator can take an extra argument which is called regular I/O stream expressions (denoted by small Greek letter α).

Definition 11 (DLTL syntax)**.** The abstract syntax of DLTL is given as follows where $a \in AP$, $c \in \mathsf{CIO}$.

$$\varphi ::= true \mid p \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \ \mathbf{U}^{\alpha} \ \varphi_2$$
$$\alpha ::= c \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^* \mid \alpha_1; \alpha_2$$

The semantics of DLTL is similar to BTSL (Section 2.3.1), except that all reasoning is now done on infinite runs. The PDL adaptation $\varphi_1 \ \mathbf{U}^{\alpha} \ \varphi_2$ is used to represent the condition that the desired run starts with a finite prefix where the data flow is a word in the language $\mathcal{L}_{\mathcal{N}}(\alpha)$ defined by the regular I/O stream expression α.

Definition 12 (DLTL semantics)**.** Given a constraint automaton $\mathcal{A}$ and an infinite run $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$, the semantics of DLTL formulas is given as follows.

$\mathcal{A}, \theta \models true$
$\mathcal{A}, \theta \models p$ iff $p \in \mathsf{L}(s_0)$
$\mathcal{A}, \theta \models \neg\varphi_1$ iff $\mathcal{A}, \theta \not\models \varphi_1$
$\mathcal{A}, \theta \models \varphi_1 \vee \varphi_2$ iff $\mathcal{A}, \theta \models \varphi_1$ or $\mathcal{A}, \theta \models \varphi_2$
$\mathcal{A}, \theta \models \varphi_1 \ \mathbf{U}^{\alpha} \ \varphi_2$ iff $\exists\ k \geq 0$ such that $ios(prefix(\theta, k)) \in \mathcal{L}_{\mathcal{N}}(\alpha)$, $suffix(\theta, k) \models \varphi_2$ and $\forall\ 0 \leq j < k : suffix(\theta, j) \models \varphi_1$

The semantics of the regular I/O stream expression α is the same as given in Definition 9 except that we ignore the *stop* condition.

We use the notation $\mathcal{A} \models \varphi$ to denote that all runs of $\mathcal{A}$ satisfy the formula φ.

Table 2.3: Derived DLTL operators

$\bigcirc\varphi$	$\equiv$	$true\ \mathbf{U}^{\mathsf{CIO}}\ \varphi$	$\varphi_1\ \mathbf{U}\ \varphi_2$	$\equiv$	$\varphi_1\ \mathbf{U}^{\mathsf{CIO}^*}\ \varphi_2$
$\Diamond\varphi$	$\equiv$	$true\ \mathbf{U}\ \varphi$	$\Box\varphi$	$\equiv$	$\neg\Diamond\neg\varphi$
$\langle\alpha\rangle\varphi$	$\equiv$	$true\ \mathbf{U}^{\alpha}\ \varphi$	$[\alpha]\varphi$	$\equiv$	$\neg\langle\alpha\rangle\neg\varphi$

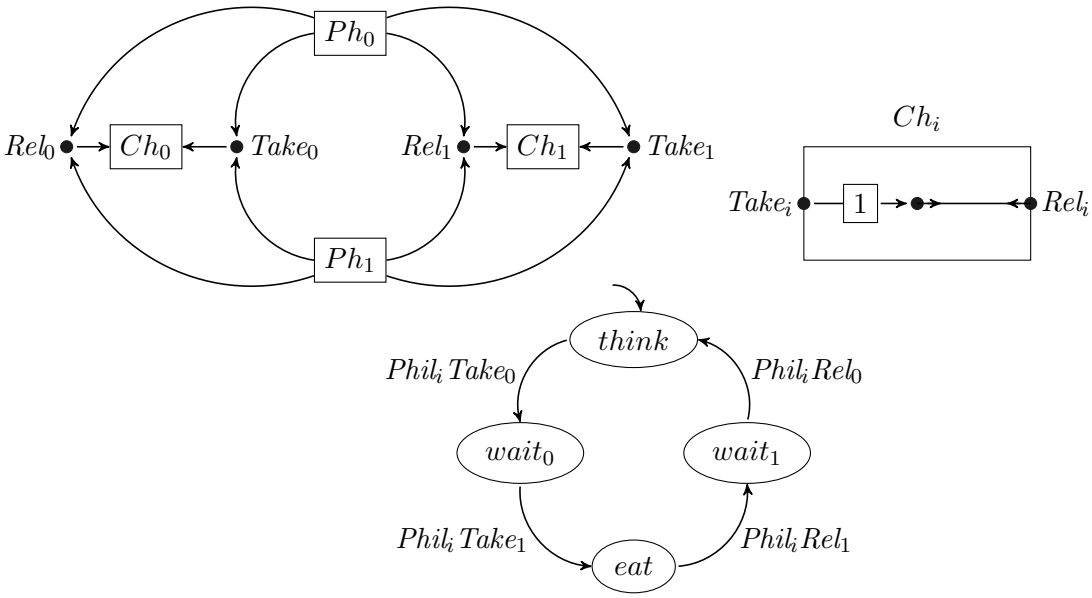


Figure 2.8: Reo connectors of dining philosophers and a chopstick, and a constraint automata for a philosopher [BB07]

Using this semantics, we can derive the usual short cut operators ($\bigcirc, \Box, \Diamond, \langle\alpha\rangle, [\alpha]$, and the regular $\mathbf{U}$). The derived operators are shown in Table 2.3.

Example 6. Figure 2.8 shows a Reo connector that depicts the classical dining philosophers coordination model [Dij71] for 2 philosophers, and a constraint automaton that defines the exact behavior of a philosopher. In this simple situation, the following DLTL formula specifies the deadlock freedom.

$$(\Box\Diamond\langle Phil_0\,Take_0\rangle true \rightarrow \Box\Diamond\langle Phil_0\,Take_1\rangle true) \wedge (\Box\Diamond\langle Phil_1\,Take_0\rangle true \rightarrow \Box\Diamond\langle Phil_1\,Take_1\rangle true)$$

The first part of the transition label indicates the philosopher in action, while the second indicates the chopstick.

2.4.2 Model Checking Algorithm

The DLTL model checking procedure [HT99, GM06] is essentially the same as the automata-based approach proposed to check LTL formulas [VW86]. The idea relies on the fact that for each DLTL formula φ, a nondeterministic Büchi automaton (NBA) can be constructed. Instead of directly trying to solve $\mathcal{A} \models \varphi$, we look for a path θ in $\mathcal{A}$ such that $\theta \models \neg\varphi$. If no such path is found, then $\mathcal{A} \models \varphi$. Algorithm 2.2 describes the steps needed to check a DLTL formula. The construction of the NBA $\mathcal{B}$ of φ is very involved and we urge the reader to consult [GM06] for a detailed instruction. Here we summarize the basic idea of the construction given there.

First we redefine the until syntax in terms of finite automata $\mathcal{Y}_\alpha$ where the language it recognizes is the language represented by the regular I/O stream expression α. More precisely,

Algorithm 2.2 DLTL model checking

1: Construct an NBA $\mathcal{B} = (B, \longrightarrow_{\mathcal{B}}, B_0, B_F)$ such that $\mathcal{L}_{\mathcal{B}} = \mathcal{L}_{\mathcal{N}}(\neg\varphi)$.
2: Construct the product automaton $\mathcal{A} \times \mathcal{B} = (Q \times B, \mathcal{N}, \longrightarrow_{\mathcal{A}\times\mathcal{B}}, Q_0 \times B_0, F')$.
3: **if** there exists a path θ in $\mathcal{A} \times \mathcal{B}$ such that the acceptance condition of $\mathcal{B}$ is fulfilled **then**
4: **return** $\mathcal{A} \not\models \varphi$
5: **else**
6: **return** $\mathcal{A} \models \varphi$
7: **end if**

we create an ϵ-free NFA $\mathcal{Y}_\alpha = (Q, \delta, Q_F)$ with a single original initial state q_0 (see e.g. [HSW01] for such a construction), where Q is a finite set of states, $\delta : Q \times \mathsf{CIO} \to 2^Q$ is the transition function, and Q_F is the set of final states. Given a state $q \in Q$, we denote $\mathcal{Y}_\alpha(q)$ an automaton $\mathcal{Y}_\alpha$ with initial state q. For each occurrence of the until operator $\varphi_1 \mathbf{U}^\alpha \varphi_2$ in the DLTL formula φ, we transform it to $\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q_0)} \varphi_2$.

Definition 13 (Axioms of DLTL [HT99, GM06])**.** The rule set used for generating the tableau is based on the following three axioms.

(A1) $\bigvee_{c\in\mathsf{CIO}} \langle c\rangle \mathit{true}$
(A2) $\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q)} \varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}} \langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q')} \varphi_2)$ (q is a final state)
(A3) $\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q)} \varphi_2 \equiv \varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}} \langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q')} \varphi_2$ (q is not a final state)

The first axiom states that the run must be infinite. The next two axioms are the expansion axioms for the until formula.

2.4.2.1 Tableau Computation

A signed formula is an expression $\mathbf{T}\varphi$ or $\mathbf{F}\varphi$, where φ is an (unsigned) formula [Smu68] which we informally read as "φ is true" or "φ is false", respectively. The tableau procedure for DLTL introduced in [GM06] takes as input a set of signed formulas and produces a set of sets of signed formulas. The result is produced by following a rule set of the following format.

$\Phi \Rightarrow \Psi_1, \Psi_2$: if Φ belongs to the set of formulas, then add Ψ_1 and Ψ_2 to the set.
$\Phi \Rightarrow \Psi_1 \mid \Psi_2$: if Φ belongs to the set of formulas, then create two copies of the set and add Ψ_1 to one set, and Ψ_2 to the other.

Definition 14 (DLTL tableau rule set)**.** The rules used in the tableau generation are as follows.

TAnd: $\mathbf{T}\varphi_1 \wedge \varphi_2 \Rightarrow \mathbf{T}\varphi_1, \mathbf{T}\varphi_2$
FAnd: $\mathbf{F}\varphi_1 \wedge \varphi_2 \Rightarrow \mathbf{F}\varphi_1 \mid \mathbf{F}\varphi_2$
TOr: $\mathbf{T}\varphi_1 \vee \varphi_2 \Rightarrow \mathbf{T}\varphi_1 \mid \mathbf{T}\varphi_2$
FOr: $\mathbf{F}\varphi_1 \vee \varphi_2 \Rightarrow \mathbf{F}\varphi_1, \mathbf{F}\varphi_2$
TNeg: $\mathbf{T}\neg\varphi_1 \Rightarrow \mathbf{F}\varphi_1$
FNeg: $\mathbf{F}\neg\varphi_1 \Rightarrow \mathbf{T}\varphi_1$
TUntilFS: $\mathbf{T}\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q)} \varphi_2 \Rightarrow \mathbf{T}(\varphi_2 \vee (\varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}} \langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q')} \varphi_2))$ (q is a final state)
TUntilNFS: $\mathbf{T}\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q)} \varphi_2 \Rightarrow \mathbf{T}(\varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}} \langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q')} \varphi_2)$ (q is not a final state)
FUntilFS: $\mathbf{F}\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q)} \varphi_2 \Rightarrow \mathbf{F}(\varphi_2 \vee (\varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}} \langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q')} \varphi_2))$ (q is a final state)
FUntilNFS: $\mathbf{F}\varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q)} \varphi_2 \Rightarrow \mathbf{F}(\varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}} \langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1 \mathbf{U}^{\mathcal{Y}_\alpha(q')} \varphi_2)$ (q is not a final state)

The boolean operator $\wedge$ and the modality $\langle c\rangle$ are used as primitive operators and can be derived from the equivalence given at the end of Section 2.4.1.

Definition 15 (Elementary DLTL formulas)**.** Elementary DLTL formulas are signed formulas $\mathbf{T}\Phi$ or $\mathbf{F}\Phi$ where Φ is either *true*, *false*, a proposition or $\langle c\rangle\varphi$. In other words, elementary formulas are formulas to which none of the rules above are applicable.

Definition 16 (Consistent set of signed formulas)**.** A set of signed formulas S is called consistent if none of the following cases are applicable.

- $\mathbf{T}\mathit{false} \in S$
- $\mathbf{F}\mathit{true} \in S$
- $\mathbf{T}\varphi \in S$ and $\mathbf{F}\varphi \in S$
- $\mathbf{T}\langle c_1\rangle\varphi_1 \in S$ and $\mathbf{T}\langle c_2\rangle\varphi_2 \in S$ with $c_1 \neq c_2$

The last case maintains the linear time property of a run, that no two actions can be taken in the same state. Note that this is not required for $\mathbf{F}\langle c\rangle\varphi \in S$ due to the $\mathbf{F}$ sign.

Given a set of signed formulas S, function $\mathit{tableau}(S)$ generates the set of sets of formulas that are generated from S by repeatedly applying the rule set given in Definition 14 until no more rules are applicable. All non-elementary formulas in each of the set of formulas must be expanded. If the expansion of a set causes the resulting set to be inconsistent, we throw away that resulting set.

2.4.2.2 NBA Generation

Generating a proper NBA for a DLTL formula is more difficult than for an LTL formula because we need to keep track of the status of all the until subformulas (i.e. whether all final states of the NFA have been reached in order). To overcome this problem, we disinguish the signed until formulas by giving them a parity label attached to the $\mathbf{T}$ and $\mathbf{F}$ signs. The parity label functions as an indicator whether the until formula is a *derived* formula or not.

As for building a generalized NBA for LTL [GPVW96], we use the tableau function described in the previous section to build a graph which defines the states and the transitions of the automaton. The nodes of the graph are the sets of formulas obtained through the tableau construction. Each set of formulas $\mathcal{F}$ also contains formulas of the kinds $\mathbf{T}\langle c\rangle\varphi$ and $\mathbf{F}\langle c\rangle\varphi$ which we consider as triggers to create new nodes. The *next state* operator $\langle c\rangle$ suggests that we connect through edges labeled by c the new nodes built from the application of the tableau procedure on the set of formulas that should hold in the next state $(\{\mathbf{T}\varphi \mid \mathbf{T}\langle c\rangle\varphi\} \cup \{\mathbf{F}\varphi \mid \mathbf{F}\langle c\rangle\varphi\})$.

Definition 17 (Derived and new until formula)**.** The signed formulas $\varphi = \mathbf{T}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q')}\ \varphi_2$ of node n' are said to be *derived* from $\mathbf{T}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2$ of node n if it is the result of applying the TUntilFS or TUntilNFS rule and expanding the next state operator $\langle c\rangle$ to create n' (i.e. $q' \in \delta(q, c)$). If a node contains an until formula which is not derived from a predecessor node, the formula is said to be *new*. In the case that in a node an until formula is both derived and new, we consider it as a derived formula.

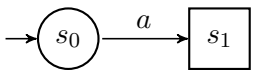


Figure 2.9: A sample NFA $\mathcal{Y}$

Example 7. Let $\mathcal{Y}$ be the NFA in Figure 2.9, and $\mathcal{W}$ another NFA. Assume that we are expanding the signed formulas $\mathbf{T}p\ \mathbf{U}^{\mathcal{Y}(s_0)}\ (q\ \mathbf{U}^{\mathcal{W}(t)}\ r)$ and $\mathbf{T}p\ \mathbf{U}^{\mathcal{Y}(s_1)}\ (q\ \mathbf{U}^{\mathcal{W}(t)}\ r)$. Since s_0 is not a final state in $\mathcal{Y}$, we apply the TUntilNFS rule to get $\mathbf{T}p \wedge \langle a\rangle p\ \mathbf{U}^{\mathcal{Y}(s_1)}\ (q\ \mathbf{U}^{\mathcal{W}(t)}\ r)$ and further rule applications yield $\mathbf{T}p$ and $\mathbf{T}\langle a\rangle p\ \mathbf{U}^{\mathcal{Y}(s_1)}\ (q\ \mathbf{U}^{\mathcal{W}(t)}\ r)$. On the other hand s_1 is a final state. Thus, we apply the TUntilFS rule to get $\mathbf{T}q\ \mathbf{U}^{\mathcal{W}(t)}\ r$. So in this tableau expansion, we end up with a *new* signed until formula $\mathbf{T}q\ \mathbf{U}^{\mathcal{W}(t)}\ r$, while the **next** generated node contains the *derived* signed until formula $\mathbf{T}p\ \mathbf{U}^{\mathcal{Y}(s_1)}\ (q\ \mathbf{U}^{\mathcal{W}(t)}\ r)$.

Given a DLTL formula φ, we generate a graph G_φ that will be converted into a Büchi automaton. The graph G_φ is a tuple (V, E), where V is the set of nodes of the form $(\mathcal{F}, x, f)$ ($\mathcal{F}$ is a set of signed formulas resulting from the tableau procedure, $x \in \{0, 1\}$, and $f \in \{\downarrow, \checkmark\}$) and E is the set of edges representing a subset of $V \times \mathsf{CIO} \times V$. The label x is used to indicate which of the until formulas are derived, while f is used to indicate the accepting states in the Büchi automaton counterpart of G_φ.

We extend the signed formulas such that all until formulas signed with $\mathbf{T}$ have a label 0 or 1 so their form becomes $\mathbf{T}^l\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2$ where $l \in \{0, 1\}$. Signed formulas $\mathbf{T}^0\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2$ and $\mathbf{T}^1\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2$ are considered to be different.

For each node $(\mathcal{F}, x, f)$ we assign the label of each signed until formula in $\mathcal{F}$ by the following rules.

- If it is a *derived* signed until formula, then the label is the same as the label of the until formula it derives from in the predecessor node.
- If it is a *new* signed until formula, then the label is $1 - x$.

To reinforce these rules, the tableau function is modified to receive an additional parameter x. Furthermore, we also tag the formulas with next state operator $\mathbf{T}\langle c\rangle\varphi$ to store information regarding the label of the until formula that produces $\mathbf{T}\langle c\rangle\varphi$.

Algorithm 2.3 shows how the graph G_φ is created. The function *create_graph* returns the tuple (V, E) as well as I, the set of initial nodes, which is necessary for the conversion to the Büchi automaton. U is the set of nodes whose successor nodes are yet to be determined.

Given a graph G_φ, an NBA $\mathcal{B}_\varphi = (B, \longrightarrow_{\mathcal{B}}, B_0, B_F)$ can be obtained by taking directly the nodes and edges of the graph as the states and transitions of $\mathcal{B}_\varphi$ (i.e. $B = V$ and $\longrightarrow_{\mathcal{B}} = E$). Furthermore, I acts as the set of initial states ($B_0 = I$). The set of accepting states of $\mathcal{B}_\varphi$ is all states whose corresponding nodes are of the form $(\mathcal{F}, x, \checkmark)$ ($B_F = \{(\mathcal{F}, x, f) \mid (\mathcal{F}, x, f) \in V \wedge f = \checkmark\}$).

The construction of the graph ensures that the x and f values of each node have the following properties.

- if a node contains $(0, \checkmark)$ then its successor nodes contain $(1, \downarrow)$.
- if a node contains $(1, \checkmark)$ then its successor nodes contain $(0, \downarrow)$.
- if a node contains $(0, \downarrow)$ then its successor nodes contain either $(0, \downarrow)$ or $(0, \checkmark)$.
- if a node contains $(1, \downarrow)$ then its successor nodes contain either $(1, \downarrow)$ or $(1, \checkmark)$.

Therefore, each run goes through alternating sequences of 0 and 1 in the x values and possibly ends with an infinite sequence of nodes whose x values are all the same. We call a sequence of nodes with the same x value $l \in \{0, 1\}$ an l-sequence. A node receives a $\checkmark$ as its f value if all until obligations have been fulfilled and this ends the l-sequence and starts a new $(1 - l)$-sequence. Thus, a proper accepting run must contain infinitely many nodes having $\checkmark$ as its f value, and consequently the number of nodes in all its 0-sequences and 1-sequences must be finite.

Example 8 (NBA for φ)**.** Consider a DLTL formula $\varphi = \Box\langle(a; a)^+\rangle p$ where the alphabet is a singleton $\{a\}$ and α^+ is an abbreviation of $\alpha; \alpha^*$. Using the equivalences provided at the end of Section 2.4.1 and generating NFA for the two until formulas, φ is rewritten as $\neg(\mathit{true}\ \mathbf{U}^{\mathcal{Y}_{\Sigma^*}(t_0)}\ \neg(\mathit{true}\ \mathbf{U}^{\mathcal{Y}_{(a;a)^+}(s_0)}\ p))$. Figure 2.10 shows NFA for both regular I/O stream expressions, and the generated 34 state NBA $\mathcal{B}$. We rename $\mathcal{Y}_{\Sigma^*}$ as $\mathcal{V}$ and $\mathcal{Y}_{(a;a)^+}$ as $\mathcal{W}$. The boxed states are the accepting states of the NBA, and all edges are labeled with a. Due to space limitation, the detailed labeling of each node is not given. To give a taste, here are the complete labels for nodes q_0, the initial state, and q_4, one of the accepting states.

Algorithm 2.3 *create_graph*(φ)

1: $I := \emptyset$
2: **for each** $\mathcal{F} \in tableau(\{\mathbf{T}\varphi\}, 0)$ **do**
3: $\quad I := I \cup \{\mathcal{F}, 0, \checkmark\}$
4: **end for**
5: $U := I; V := I; E := \emptyset$
6: **while** $U \neq \emptyset$ **do**
7: $\quad$ remove $n = (\mathcal{F}, x, f)$ from U
8: $\quad$ **if** $f = \checkmark$ **then**
9: $\quad\quad x' := 1 - x$
10: $\quad$ **else** {at least one until formula is not resolved}
11: $\quad\quad x' := x$
12: $\quad$ **end if**
13: $\quad$ **for each** $\mathcal{F}' \in tableau(\{\mathbf{T}\varphi \mid \mathbf{T}\langle c\rangle\varphi \in \mathcal{F}\} \cup \{\mathbf{F}\varphi \mid \mathbf{F}\langle c\rangle\varphi \in \mathcal{F}\}, x')$ **do**
14: $\quad\quad$ **if** $f = \checkmark$ **then**
15: $\quad\quad\quad f' :=\downarrow$
16: $\quad\quad$ **else if** there exists no $\mathbf{T}^{x'}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2 \in \mathcal{F}'$ **then**
17: $\quad\quad\quad f' := \checkmark$
18: $\quad\quad$ **else**
19: $\quad\quad\quad f' :=\downarrow$
20: $\quad\quad$ **end if**
21: $\quad\quad n' := (\mathcal{F}', x', f')$
22: $\quad\quad$ **if** $\exists\ n'' \in V$ such that $n'' = n'$ **then**
23: $\quad\quad\quad E := E \cup \{(n, c, n'')\}$
24: $\quad\quad$ **else**
25: $\quad\quad\quad V := V \cup \{n'\}$
26: $\quad\quad\quad E := E \cup \{(n, c, n')\}$
27: $\quad\quad\quad U := U \cup \{n'\}$
28: $\quad\quad$ **end if**
29: $\quad$ **end for**
30: **end while**
31: **return** $\langle V, E, I\rangle$

$$
\begin{aligned}
q_0 = \big(\quad & \{\mathbf{T}^1\neg(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p)), \mathbf{T}\langle a\rangle true, \mathbf{F}true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p),\\
& \mathbf{F}\neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p) \vee (true \wedge \langle a\rangle(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p))),\\
& \mathbf{F}\neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p), \mathbf{F}(true \wedge \langle a\rangle(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p))),\\
& \mathbf{T}^1(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p), \mathbf{F}\langle a\rangle(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p)), \mathbf{T}true,\\
& \mathbf{T}(true \wedge \langle a\rangle(true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p)), \mathbf{T}^1\langle a\rangle(true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p)\}, 0, \checkmark\big)\\
q_4 = \big(\quad & \{\mathbf{T}\langle a\rangle true, \mathbf{T}true, \mathbf{F}true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p), \mathbf{T}^0(true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p),\\
& \mathbf{T}^0((true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p) \vee (true\ \mathbf{U}^{\mathcal{W}(s_3)}\ p)), \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p,\\
& \mathbf{F}\neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p) \vee (true \wedge \langle a\rangle(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p))),\\
& \mathbf{F}\neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p), \mathbf{F}(true \wedge \langle a\rangle(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p))),\\
& \mathbf{F}\langle a\rangle(true\ \mathbf{U}^{\mathcal{V}(t_0)}\ \neg(true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p)), \mathbf{T}^0 true \wedge \langle a\rangle(true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p),\\
& \mathbf{T}^0\langle a\rangle(true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p), \mathbf{T}^0 true \wedge (\langle a\rangle((true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p) \vee (true\ \mathbf{U}^{\mathcal{W}(s_3)}\ p))),\\
& \mathbf{T}^0\langle a\rangle((true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p) \vee (true\ \mathbf{U}^{\mathcal{W}(s_3)}\ p)), \mathbf{T}^0(true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p), \mathbf{T}p\}, 1, \checkmark\big)
\end{aligned}
$$

An example of an accepting run in $\mathcal{B}$ is $q_0q_1(q_2q_4q_8q_{14}q_{21}q_{32})^\omega$ where the significant labels of each node are as follows.

$$
\begin{aligned}
q_0 &= (\{\mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p\}, 0, \checkmark)\\
q_1 &= (\{\mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p\}, 1, \downarrow)\\
q_2 &= (\{\mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p, \mathbf{T}p\}, 1, \downarrow)\\
q_4 &= (\{\mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p, \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p, \mathbf{T}p\}, 1, \checkmark)\\
q_8 &= (\{\mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p, \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p, \mathbf{T}p\}, 0, \downarrow)\\
q_{14} &= (\{\mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p, \mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p, \mathbf{T}p\}, 0, \downarrow)\\
q_{21} &= (\{\mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p, \mathbf{T}p\}, 0, \checkmark)\\
q_{32} &= (\{\mathbf{T}^0 true\ \mathbf{U}^{\mathcal{W}(s_0)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p, \mathbf{T}^1 true\ \mathbf{U}^{\mathcal{W}(s_2)}\ p, \mathbf{T}p\}, 1, \downarrow)
\end{aligned}
$$

This run guarantees that each until formula with the label l is fulfilled before the beginning of the next $(1-l)$-sequence.

All labelings are necessary so we can differentiate between a correct accepting run and a seemingly, but falsely, accepting run. In fact, in this case, the distance between any states which have the label $(\{\mathbf{T}^l true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p\}, l, \downarrow)$ to any accepting states of label $(\{\mathbf{T}^{1-l} true\ \mathbf{U}^{\mathcal{W}(s_1)}\ p\}, l, \downarrow)$ is odd, which correctly enforces the acceptance condition of the regular expression $(a;a)^+$.

2.4.2.3 Product Automata

Definition 18 (Product automata)**.** Product automaton $\mathcal{A} \times \mathcal{B}$ is a tuple $(Q \times B, \mathcal{N}, \longrightarrow_{\mathcal{A}\times\mathcal{B}}, Q_0 \times B_0, F')$ where $F' \subseteq Q \times B = \{(s,b) \mid b \in F\}$, $b = (\mathcal{F}, x, f)$, $b' = (\mathcal{F}', x', f')$ and $\longrightarrow_{\mathcal{A}\times\mathcal{B}}$ is the smallest relation defined by the rule below.

$$
\frac{\begin{array}{cc} s \xrightarrow{c}_{\mathcal{A}} s' & b \xrightarrow{c}_{\mathcal{B}} b' \\ \{\varphi \mid \mathbf{T}\varphi \in \mathcal{F} \wedge \varphi \in AP\} \subseteq \mathsf{L}(s) & \{\varphi \mid \mathbf{F}\varphi \in \mathcal{F} \wedge \varphi \in AP\} \cap \mathsf{L}(s) = \emptyset \\ \{\varphi \mid \mathbf{T}\varphi \in \mathcal{F}' \wedge \varphi \in AP\} \subseteq \mathsf{L}(s') & \{\varphi \mid \mathbf{F}\varphi \in \mathcal{F}' \wedge \varphi \in AP\} \cap \mathsf{L}(s') = \emptyset \end{array}}{(s,b) \xrightarrow{c}_{\mathcal{A}\times\mathcal{B}} (s',b')}
$$

The notion of an accepting run in this product automaton is the same as in the Büchi automaton, i.e. at least one product state in F' has to be visited infinitely often. We can verify whether a constraint automaton fulfill a specification given as a DLTL formula by checking for emptiness in the product automaton, as described by Algorithm 2.2.

Remark. DLTL has the same expressive power as the monadic second-order theory of infinite sequences over alphabet Σ [HT99] and thus it is strictly more expressive than LTL. Despite their expressiveness, the DLTL model checking problem is PSPACE-complete [GM06], the same type of complexity as for LTL model checking.

2.5 Chapter Summary

In this chapter, we summarized the main concepts of Reo connectors and explained how we can construct constraint automata that serve as the operational semantics of these connectors. One of the benefits of having constraint automata is that we can verify specifications using standard model checking techniques with some modifications. We considered two temporal logics, BTSL and DLTL, to reason about constraint automata and explained how model checking is performed using these logics. The next chapter will introduce the temporal logic BTSL* which is a combination of DLTL and BTSL and we will see how we can deal with fairness.

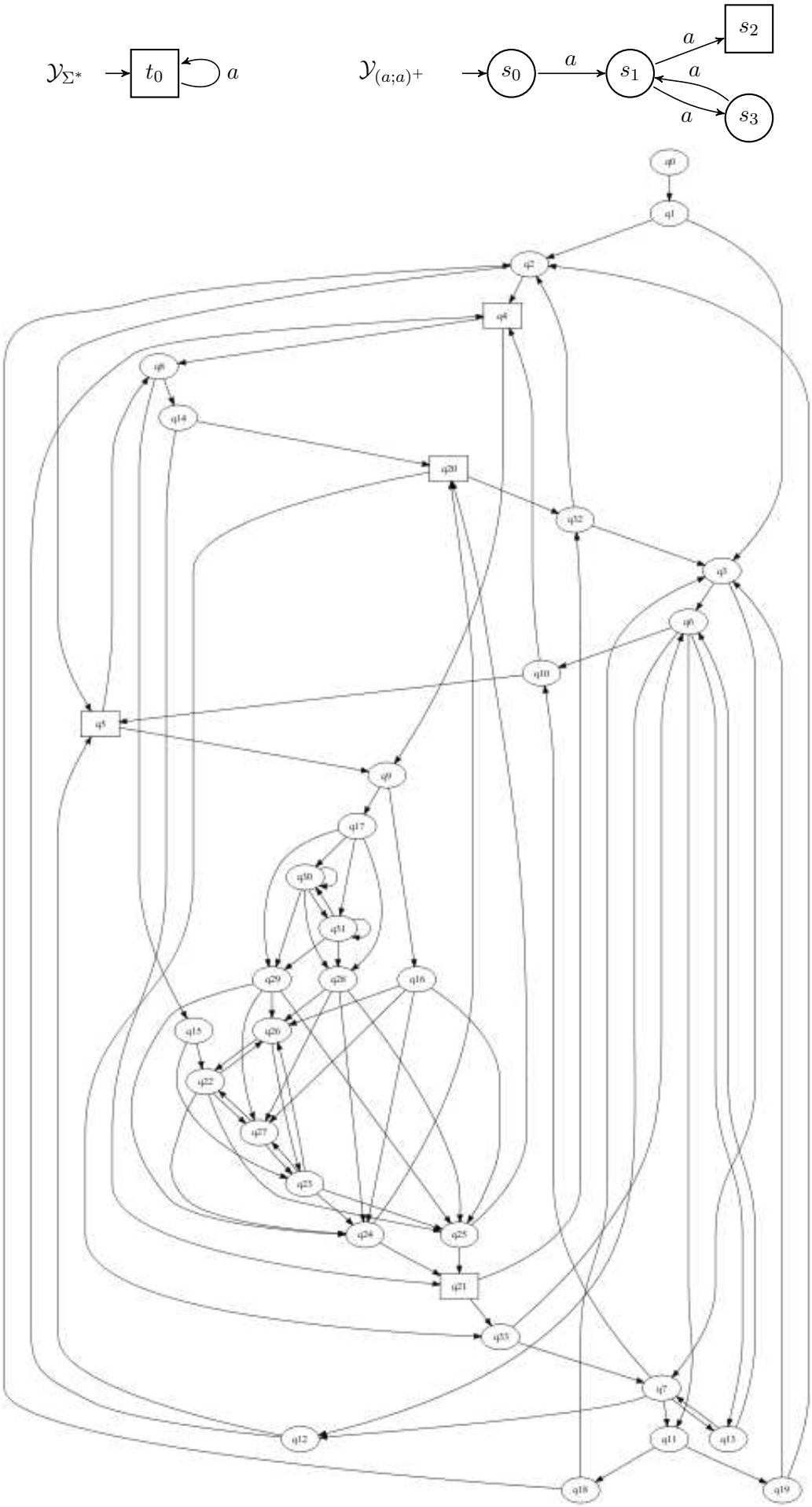


Figure 2.10: NFA for Σ^* and $(a;a)^+$, and NBA for $\Box\langle(a;a)^+\rangle p$

Chapter 3

Model Checking of BTSL* with Fairness

This chapter introduces BTSL*: a temporal logic which allows universal or existential path quantification combined with any path property that can be specified in DLTL or BTSL. We can use BTSL* to express fairness assumptions on how the Reo connectors behave. Since the behavior of the Reo connectors is captured especially in the transitions of the corresponding constraint automaton, the fair assumptions are transition-based.

The logic of BTSL* is similar to the logic of ReCTL* [Cla05], however ReCTL* deals only with infinite runs. BTSL* follows the approach of BTSL which includes the operator *stop* to allow reasoning with finite runs. This feature is useful for handling deadlock configurations that might appear in Reo networks [KB07].

This chapter is structured as follows. The first section defines the syntax and semantics of BTSL*. The second section discusses how fairness is specified in BTSL*. The third section describes how BTSL* model checking is performed. A discussion on how to handle finite runs is given in the following section. The fifth section is dedicated to showing how fairness in BTSL is handled, as we can treat them in a simpler manner than using the BTSL* model checking algorithm. This chapter is concluded with a short summary.

3.1 Syntax and Semantics

Definition 19 (Syntax of BTSL*)**.** The abstract syntax of BTSL* is given as follows where $p \in AP$, $c \in \mathsf{CIO}$.

$$\Phi ::= \mathit{true} \mid p \mid \neg\Phi_1 \mid \Phi_1 \vee \Phi_2 \mid \exists\ \varphi \mid \forall\ \varphi$$
$$\varphi ::= \Phi \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1\ \mathbf{U}^{\alpha}\ \varphi_2$$
$$\alpha ::= c \mid \mathit{stop} \mid \alpha_1 \cup \alpha_2 \mid \alpha_1^* \mid \alpha_1; \alpha_2$$

The semantics of BTSL* draws similarity to the semantics of BTSL as defined in Definition 9. The main difference lies in the path formulas, where it is possible to apply propositional operations directly on the path level and path level nesting is allowed. The following definition provides the semantics of the path formulas in BTSL*. The semantics of the state formulas and the regular I/O stream expressions remain the same as for BTSL.

Definition 20 (BTSL* Semantics for Path Formulas)**.** Given a maximal run $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$, the satisfaction relation $\models$ for path formulas is defined as follows.

$$
\begin{array}{lll}
\theta \models \Phi & \text{iff} & s_0 \models \Phi \\
\theta \models \neg\varphi_1 & \text{iff} & \theta \not\models \varphi_1 \\
\theta \models \varphi_1 \vee \varphi_2 & \text{iff} & \theta \models \varphi_1 \text{ or } \theta \models \varphi_2 \\
\theta \models \varphi_1 \ \mathbf{U}^{\alpha}\ \varphi_2 & \text{iff} & \exists\ 0 \leq k < |\theta| \text{ such that } \mathit{ios}(\mathit{prefix}(\theta,k)) \in \mathcal{L}_{\mathcal{N}}(\alpha),\ \mathit{suffix}(\theta,k) \models \varphi_2 \text{ and} \\
 & & \forall\ 0 \leq j < k : \mathit{suffix}(\theta,j) \models \varphi_1
\end{array}
$$

Following from the semantics, we can reuse the same short hand notations that are listed in Table 2.3. From the semantics, it is clear that BTSL is a sublogic of BTSL*. DLTL can also be viewed as a sublogic of BTSL*, because a DLTL formula φ holds for a constraint automaton $\mathcal{A}$ iff a BTSL* formula $\forall\varphi$ holds for that constraint automaton as well. Additionally, the one step transition c can be represented using $\langle c \rangle \mathit{true}$ in BTSL*, and $\surd$ as $\langle \mathit{stop} \rangle$.

3.2 Fairness

Fairness is a well-known concept for parallel systems and it is related to the lack of progress of some component in the system that is caused by a certain undesirable phenomenon [Kwi89]. In Reo, such a phenomenon may be caused by unrealistic behavior where components being infinitely faster than the others and the nondeterminism of a sink/mixed node (see Section 2.1.2) favors only certain options. Fairness assumptions are used to rule out this kind of behavior and focus the verification close to the way the components are implemented.

The behavior of a Reo connector is captured in the transitions of its corresponding constraint automaton. We deal here with *transition-based fairness* where no transition that is infinitely often enabled should be ignored indefinitely. A concurrent I/O operation $c \in \mathsf{CIO}$ is called enabled (denoted as $\mathit{enabled}(c)$) in state s if there is a transition from s to an arbitrary state t whose label is c. We assume the standard types of fairness with respect to some sets of concurrent I/O operations: unconditional fairness (*ufair*), strong fairness (*sfair*) and weak fairness (*wfair*).

Definition 21 (Fairness conditions)**.** Given sets of concurrent I/O operations $C_1, \ldots, C_k, D_1, \ldots, D_k, \subseteq \mathsf{CIO}$,

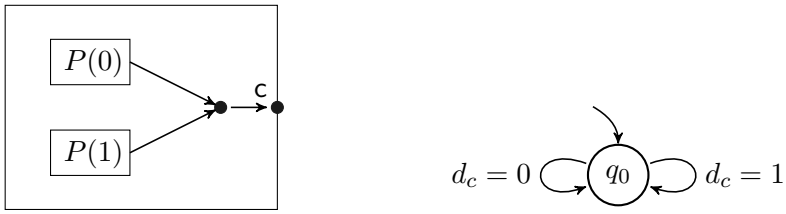
$$
\begin{array}{rcl}
\mathit{ufair} & = & \bigwedge\limits_{0<i\leq k} \Box\Diamond \mathit{taken}(C_i) \\
\mathit{sfair} & = & \bigwedge\limits_{0<i\leq k} (\Box\Diamond \mathit{enabled}(C_i) \rightarrow \Box\Diamond \mathit{taken}(D_i)) \\
\mathit{wfair} & = & \bigwedge\limits_{0<i\leq k} (\Diamond\Box \mathit{enabled}(C_i) \rightarrow \Box\Diamond \mathit{taken}(D_i))
\end{array}
$$

where $\mathit{enabled}(C) = \bigvee\limits_{c\in C} \mathit{enabled}(c)$ and $\mathit{taken}(C) = \bigvee\limits_{c\in C} c$.
These sets are represented symbolically using I/O constraints.

Unconditional fairness (also called *impartiality* [LPS81]) allows I/O constraints to occur unconditionally. In other words, this fairness assumption guarantees that each I/O constraint C_i is eventually fulfilled regardless of the state of the constraint automaton. Of the three types of fairness presented in Definition 21, unconditional fairness is the strongest one.

Example 9 (Unconditional fairness)**.** Figure 3.1 illustrates a Reo connector that behaves as a random bit generator. This connector contains 2 producer components that always produce the data item as specified in their parameter (e.g. $P(0)$ means that the component produces data item 0 whenever required). Theoretically, it is possible that this generator only produces bit 0 (or, symmetrically, only bit 1). However, any reasonable implementation of the connector would not allow this to happen. Thus, we use the unconditional fairness assumption to characterize that every bit is generated infinitely often: $\Box\Diamond \mathit{taken}(\{d_c = 0\}) \wedge \Box\Diamond \mathit{taken}(\{d_c = 1\})$.

Strong fairness (also called *compassion* or simply *fairness* [Kwi89]) allows us to give some conditions into the fairness assumption, which is the possibility of fulfilling certain I/O constraints. When the specified I/O constraints are enabled infinitely often, then they must be



(a) Random bit generator component (b) Random bit generator CA

Figure 3.1: Random bit generator

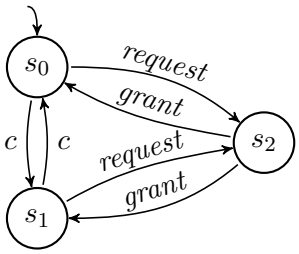


Figure 3.2: A 3 state constraint automaton

satisfied/taken infinitely often as well. Therefore, this definition disallows paths in which an I/O constraint is enabled infinitely often, but only taken a finite number of times.

Example 10 (Strong fairness)**.** Going back to Figure 2.8 which illustrates the dining philosopher connector, to assert that the first philosopher does not have his access denied to any of the chopsticks even though he had the chance to take them (i.e. no philosopher is infinitely faster than the other), we use the following strong fairness restrictions: $(\Box\Diamond \mathit{enabled}(\{\mathit{Phil}_0\,\mathit{Take}_0\}) \rightarrow \Box\Diamond \mathit{taken}(\{\mathit{Phil}_0\,\mathit{Take}_0\}))$ and $(\Box\Diamond \mathit{enabled}(\{\mathit{Phil}_0\,\mathit{Take}_1\}) \rightarrow \Box\Diamond \mathit{taken}(\{\mathit{Phil}_0\,\mathit{Take}_1\}))$. This restriction is necessary to verify that the first philosopher is free from starvation.

Weak fairness (also called *justice* [LPS81]) is a tightened version of strong fairness. In weak fairness, an I/O constraint must be enabled continuously forever from some point on in order to force that the I/O constraint is taken infinitely often. Stated differently, weak fairness disregards all paths where an I/O constraint that is enabled continuously is being postponed forever.

Example 11 (Weak fairness)**.** Figure 3.2 shows a 3 state constraint automaton that represent a Reo connector with two components: requester and grantor. The requester has 2 states whose details we are not too interested in except that from both states it can always request for some kind of permission from the grantor. The requester component suggests that at some point in time, it needs to request something from the grantor. However, this necessity is not captured within the constraint automaton. We can use the weak fairness assumption of $(\Diamond\Box \mathit{enabled}(\{\mathit{request}\}) \rightarrow \Box\Diamond \mathit{taken}(\{\mathit{request}\})$ to assure that this desired behavior occurs.

The relationship between these three types of fairness is a hierarchy of implication [BK08]. Under the same I/O constraint, unconditional fairness implies strong fairness and strong fairness implies weak fairness. This means that unconditional fairness rules out more paths than strong fairness, and the same goes for strong fairness with weak fairness. For example, the design of the dining philosopher connector never fails on weak fairness of one *Take* operation, as the enabledness of that operation will go on and off as the chopsticks may be taken by the other philosopher. In this case, strong fairness is needed to capture the desired behavior that all philosophers need to have equal opportunity to actually take a chopstick. Note that this strong fairness does not exclude the possibility of having deadlock (i.e. none of the philosopher can

grab enough chopsticks to eat), in which case the enabledness condition of the fairness is not fulfilled causing the strong fairness assumption to be satisfied.

3.2.1 Fair Semantics of BTSL*

Definition 22 (Fair runs)**.** A run $\theta \in \mathit{MaxRuns}(s)$ starting in the state s is called fair under the fairness assumption $F = \mathit{ufair} \wedge \mathit{sfair} \wedge \mathit{wfair}$ iff either θ is a finite run or $\theta \models F$. The set of all fair runs starting in state s is defined as

$$\mathit{FairRuns}(s) = \{\theta \in \mathit{MaxRuns}(s) \mid \theta \models F \vee |\theta| \in \mathbb{N}\} \ .$$

Definition 23 (Semantics of Fair BTSL*)**.** For BTSL* fairness assumption $F = \mathit{ufair} \wedge \mathit{sfair} \wedge \mathit{wfair}$, the relation $\models_F$ is defined as follows.

$s \models_F \mathit{true}$

$s \models_F p$	iff	$p \in L(s)$
$s \models_F \neg\Phi_1$	iff	$s \not\models_F \Phi_1$
$s \models_F \Phi_1 \vee \Phi_2$	iff	$s \models_F \Phi_1$ or $s \models_F \Phi_2$
$s \models_F \exists\ \varphi$	iff	there exists a fair run θ starting in s such that $\theta \models_F \varphi$
$s \models_F \forall\ \varphi$	iff	for all fair runs θ starting in s, $\theta \models_F \varphi$
$\theta \models_F \Phi$	iff	$s_0 \models_F \Phi$
$\theta \models_F \neg\varphi_1$	iff	$\theta \not\models_F \varphi_1$
$\theta \models_F \varphi_1 \vee \varphi_2$	iff	$\theta \models_F \varphi_1$ or $\theta \models_F \varphi_2$
$\theta \models_F \varphi_1\ \mathbf{U}^{\alpha}\ \varphi_2$	iff	$\exists 0 \leq k < \lvert\theta\rvert$ such that $\mathit{ios}(\mathit{prefix}(\theta,k)) \in \mathcal{L}_{\mathcal{N}}(\alpha)$, $\mathit{suffix}(\theta,k) \models_F \varphi_2$ and $\forall 0 \leq j < k : \mathit{suffix}(\theta,j) \models_F \varphi_1$

This fair semantics is almost the same as the basic BTSL* semantics, except for the changes in dealing with the quantifiers. As shown in the definition, the semantics for $\exists\ \varphi$ and $\forall\ \varphi$ only deals with fair runs.

Example 12. In Figure 3.3 we find a constraint automaton $\mathcal{A}$ for the dining philosopher component given in Figure 2.8. In this situation, all edges are labeled with only single active nodes which describe the action of one of the two philosophers (numbered 0 and 1). In the figure, P stands for philosopher, T stands for taking a chopstick, and R stands for releasing a chopstick. P_0T_0 therefore means that philosopher 0 takes chopstick 0, and P_1R_1 means that philosopher 1 releases chopstick 1. In this closed Reo connector, all internal components are willing to perform I/O operations they can do.

After applying the strong fairness assumption given in Example 10 to both philosophers, we wish to know if this configuration ensures that no philosopher starves. This can be specified using the BTSL* formula $\forall(\Box\Diamond \mathit{eat}_0 \wedge \Box\Diamond \mathit{eat}_1)$.

We claim that $\mathcal{A} \models_F \forall(\Box\Diamond \mathit{eat}_0 \wedge \Box\Diamond \mathit{eat}_1)$. The fairness assumption excludes all paths which cycle infinitely on one side of the automaton and finitely on the other side because each philosopher has the opportunity to take chopstick 0 infinitely often. Therefore, all paths which fall into our consideration cycle on both sides infinitely often. As a result, the states that contain the atomic proposition eat_0 or eat_1 are visited infinitely often and for all fair paths, $\Box\Diamond \mathit{eat}_0 \wedge \Box\Diamond \mathit{eat}_1$ is satisfied. In other words, all philosophers never starve. Without the fairness assumption, we can construct a counterexample where only one philosopher eats infinitely often (cycles happen only on one side of the automaton).

Example 13. In this example, we exclude all finite maximal runs from our reasoning as it uses a closed Reo connector with no outside influence. Continuing from Example 11, we want to ascertain that the grantor component will always have a job to do. The BTSL* formula $\forall\Box\Diamond\langle \mathit{grant}\rangle \mathit{true}$ captures this simple requirement. With the weak fairness assumption $(\Diamond\Box \mathit{enabled}(\{\mathit{request}\}) \rightarrow \Box\Diamond \mathit{request})$ in place, the constraint automaton drawn in Figure 3.2

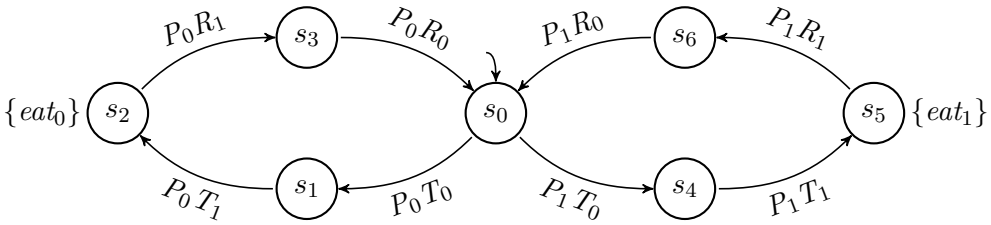


Figure 3.3: Constraint automaton for 2 dining philosophers

satisfies the requirement. In any fair path, state s_2, which is the state where the grantor contemplates on its action, must be visited infinitely often since the only transition *request* leading to that state is forced to be taken eventually. The only option the state has is that it grants the requester the access. Since this happens infinitely often, any fair path satisfies $\Box\Diamond\langle grant\rangle true$, and hence the constraint automaton fulfills our requirement.

3.3 Model Checking Algorithm

Fairness assumption can be integrated into standard BTSL* as it is the case with the encapsulation of fairness in CTL*. Using the semantics given in Definition 23, for any formula of the form $\forall\ \varphi$, we put a precondition on the path that it has to be fair. In other words, we deal with the formula $\forall\ (F \rightarrow \varphi)$. Formulas of the form $\exists\ \varphi$ insist on the existence of a fair path to be satisfied. Therefore, its translation to standard BTSL* is $\exists\ (F \wedge \varphi)$.

Checking BTSL* formulas with fairness assumption is similar to checking CTL* formulas with fairness. We combine the state based approach of BTSL with DLTL model checker performing checks for the path formulas. Because the DLTL model checker works by checking the emptiness of the language covered by a path formula (i.e. no path satisfies the path formula), we adjust formulas of the form $\exists\varphi$ by using the following equivalence.

$$s \models \exists\varphi \quad \text{iff} \quad s \not\models \forall\neg\varphi \quad \text{(BTSL* semantics) iff} \quad s \not\models \neg\varphi \quad \text{(DLTL semantics).}$$

Algorithm 3.1 checks a BTSL* formula given the fairness assumption F. If no fairness assumption is needed, then we can replace F with $true$.

Remark. The time complexity of model checking a BTSL* formula Φ with fairness assumption F on a constraint automaton $\mathcal{A}$ is equivalent to $O(size(\mathcal{A}) \cdot exp(|F| + |\Phi|))$. We can perform a polynomial reduction from BTSL* model checking to DLTL model checking as can be seen from Algorithm 3.1 which then yields this result.

3.4 Dealing with Finite Runs

One problem yet to be addressed is how to handle finite runs. Usually, the main problem with finite runs in the context of linear temporal logic is that "there is no consensus on defining LTL over finite traces" [BLS08]. As stated Section 2.4.1, DLTL defined in [HT99] only deals with infinite traces. The problem lies on how to define the next step semantics and how to handle empty runs [EFH+03]. For the latter case, we do not have empty runs since constraint automata always have initial states. Furthermore, the semantics of DLTL enables us to immediately adopt the proposal given in [MP95] to distinguish between between the *strong* next step operator $\bigcirc$ and the *weak* next step operator $\tilde{\bigcirc}$. Intuitively, the strong next step operator $\bigcirc$ means that we are still able to take another step in the run where φ holds. The weak next step operator

Algorithm 3.1 Algorithm for fair BTSL* model checking

```
1:  for all i ≤ |Φ| do
2:    for all Ψ ∈ sub(Φ) with |Ψ| = i do
3:      switch (Ψ):
            true        :  Sat_fair(Ψ) := Q
            a           :  Sat_fair(Ψ) := {s ∈ Q | a ∈ L(s)}
            enabled(C)  :  Sat_fair(Ψ) := {s ∈ Q | C ∩ CIO(s) ≠ ∅}
            a1 ∨ a2     :  Sat_fair(Ψ) := Sat_fair(a1) ∪ Sat_fair(a2)
            ¬a          :  Sat_fair(Ψ) := Q\Sat_fair(a)
            ∃φ          :  Sat_fair(ψ) := Q\Sat_DLTL(F → ¬φ)
            ∀φ          :  Sat_fair(ψ) := Sat_DLTL(F → φ)
        end switch
4:      if Ψ is a state formula then
5:        AP := AP ∪ {a_Ψ}
6:        replace every occurrence of Ψ with a_Ψ
7:        for all s ∈ Sat_fair(Ψ) do
8:          L(s) := L(s) ∪ {a_Ψ}
9:        end for
10:     end if
11:   end for
12: end for
13: return Q_0 ⊆ Sat_fair(Φ)
```

$\tilde{\bigcirc}$ relaxes this requirement by allowing a finite run to satisfy $\tilde{\bigcirc}\varphi$ iff either the first state of the finite run is already in its final state, or if we are able to move forward one step and φ is satisfied in the next state. Formally, $\bigcirc\varphi \stackrel{\text{def}}{=} true\ \mathbf{U}^{\mathsf{CIO}}\ \varphi \equiv \langle \mathsf{CIO} \rangle \varphi$, and $\tilde{\bigcirc}\varphi \stackrel{\text{def}}{=} \neg(true\ \mathbf{U}^{\mathsf{CIO}} \neg\varphi) \equiv [\mathsf{CIO}]\varphi$.

We are left with two main issues related to finite runs: the fairness assumption and the model checking procedure. First, we depart slightly from Definition 5 such that a finite run is of the form $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} s_n \xrightarrow{\surd} s_n \xrightarrow{\surd} s_n \xrightarrow{\surd} \ldots$ where the length of the run is $|\theta| = \omega$.

3.4.1 Fair Finite Runs

Definition 22 states that all finite runs are fair runs. However, the following example illustrates that we need to modify our fairness assumptions to incorporate the finite runs requirement.

Example 14. Let $\mathsf{CIO} = \{a, b\}$ and the unconditional fairness assumption be $\Box\Diamond taken(\{a\}) \equiv \neg(true\ \mathbf{U}^{\mathsf{CIO}^*}\ \neg(true\ \mathbf{U}^{\mathsf{CIO}^*}\ (true \mathbf{U}^a true)))$. The finite run consisting of two states $\theta = s_0 \xrightarrow{b} s_1 \xrightarrow{\surd} s_1 \xrightarrow{\surd} s_1 \xrightarrow{\surd} \ldots$ does not satisfy the fairness assumption.

The reason why the fairness assumption template given in Definition 21 does not hold for all finite runs lies on the semantics of the $\Diamond taken(D)$ operator which is too strong for our need. What we want here is that finite runs always satisfy this subformula. To do so, we add to the set of concurrent I/O operations D the $\surd$ symbol.

Lemma 3. *Let $D = \{d_1, \ldots, d_m\}$ where for all $1 \leq i \leq m$, $d_i \in \mathsf{CIO}$.*

- *For all finite runs $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} s_n \xrightarrow{\surd} s_n \xrightarrow{\surd} s_n \xrightarrow{\surd} \ldots$, $\theta \models \Box\Diamond taken(D \cup \{\surd\})$.*
- *For all infinite runs θ: if $\theta \models \Box\Diamond taken(D)$ then $\theta \models \Box\Diamond taken(D \cup \{\surd\})$.*

Proof. The proof for the infinite run is trivial, since in an infinite path, any transition with the symbol $\surd$ is never taken. Thus, adding $\surd$ into $D = \{d_1, \ldots, d_m\}$ does not cause any difference.

To prove the finite run part, we use the strict DLTL equivalent of $\Box\Diamond taken(D \cup \{\surd\})$ which is $\neg(true\ \mathbf{U}^{\mathsf{CIO}^*}\ \neg(true\ \mathbf{U}^{\mathsf{CIO}^*}\ (true\mathbf{U}^{(d_1\cup\cdots\cup d_m\cup\surd)}true)))$. By the semantics given in Definition 20, $true\ \mathbf{U}^{\mathsf{CIO}^*}\ (true\ \mathbf{U}^{(d_1\cup\cdots\cup d_m\cup\surd)}true)$ is always satisfied by an arbitrary finite run θ since we can always pick $k = n$ such that $ios(prefix(\theta, k)) \in \mathsf{CIO}^*$ and the next transition label is $\surd$ (naturally, *true* holds in all states). Therefore, $true\ \mathbf{U}^{\mathsf{CIO}^*}\ \neg(true\ \mathbf{U}^{\mathsf{CIO}^*}\ (true\ \mathbf{U}^{(d_1\cup\cdots\cup d_m\cup\surd)}true))$ is never satisfied and $\Box\Diamond taken(D \cup \{\surd\})$ holds for the finite run θ. □

Corollary. *Because $\Box\Diamond taken(D \cup \{\surd\})$ is always the consequent part of any fairness assumptions (or in the case of the unconditional fairness assumption: the assumption itself), finite runs are always fair under all fairness assumptions.*

3.4.2 Model Checking Finite Runs

In LTL, capturing the finite run semantics is usually done by adding a transition from all terminal states to a special 'sink' state, from which only self-transitions are possible [CCO+05]. However, this approach is not enough for DLTL as we have to deal with regular expressions, which are also able to explicitly represent finite runs. The following states 2 extensions to the tableau method specified in Section 2.4.2 to accept proper finite runs and the resulting algorithm is listed as Algorithm 3.2.

1. The extension of the alphabet of the NBA from CIO to $\mathsf{CIO} \cup \{\surd\}$. This way, at any moment, we can produce a (possibly non accepting) finite run. As a result, the until formula expansion rules used in the *tableau* function also include the $\surd$ symbol.

2. We separate the treatment in the *create_graph* algorithm (see Algorithm 2.3) between the nodes that contain $\mathbf{T}\langle c\rangle true, c \in \mathsf{CIO}$ as one of their formulas and the nodes that contain $\mathbf{T}\langle\surd\rangle true$. This distinction is required as once we take the $\surd$-transition, we can only take $\surd$-transitions afterwards. This is reflected in the following set of tableau rules for the until formulas, which is used by the $tableau_{\surd}$ function.

   TUntilFS$_{\surd}$: $\mathbf{T}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2 \Rightarrow \mathbf{T}(\varphi_2 \vee (\varphi_1 \wedge \bigvee_{c\in\{\surd\}}\langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q')}\ \varphi_2))$
   (q is a final state)

   TUntilNFS$_{\surd}$: $\mathbf{T}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2 \Rightarrow \mathbf{T}(\varphi_1 \wedge \bigvee_{c\in\{\surd\}}\langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q')}\ \varphi_2)$
   (q is not a final state)

   FUntilFS$_{\surd}$: $\mathbf{F}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2 \Rightarrow \mathbf{F}(\varphi_2 \vee (\varphi_1 \wedge \bigvee_{c\in\{\surd\}}\langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q')}\ \varphi_2))$
   (q is a final state)

   FUntilNFS$_{\surd}$: $\mathbf{F}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2 \Rightarrow \mathbf{F}(\varphi_1 \wedge \bigvee_{c\in\{\surd\}}\langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q')}\ \varphi_2)$
   (q is not a final state)

   Furthermore, as having a $\surd$-transition means that the last state is fixed, we need to accumulate the atomic propositions produced by the until formula expansions as to maintain consistency in the new sets of formulas. Example 15 illustrates this necessity.

The first extension is clearly needed as we need to mark when a run is a finite run, and the second extension ensures finite runs are represented in the NBA as $b_0 \xrightarrow{c_1} b_1 \xrightarrow{c_2} \ldots \xrightarrow{\surd} b_k \xrightarrow{\surd} b_{k+1} \xrightarrow{\surd} \ldots$. Note that we do not need to have a complete NFA, as the until expansion rule $\varphi_1 \wedge \bigvee_{c\in\mathsf{CIO}}\langle c\rangle \bigvee_{q'\in\delta(q,c)} \varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q')}\ \varphi_2$ is evaluated to false if it does not have any outgoing transition (in other words, we cannot take a next step from that state). This implies that the resulting set of signed formulas contains $\mathbf{T}false$ — which causes the set to be inconsistent and therefore the node is not expanded — or $\mathbf{F}false$ — which enables us to stop dealing with any derivatives from that until formula in the next step.

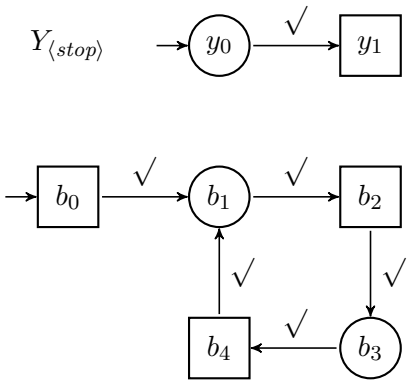


Figure 3.4: The NFA for *stop* and the incorrect NBA representing $p \wedge \langle stop \rangle \neg p$

Example 15. Let $\varphi = p \wedge \langle stop \rangle \neg p$. The regular expression *stop* is represented by the NFA $\mathcal{Y}$ given in Figure 3.4. Using the proposed extension without the accumulation of atomic propositions, the following nodes are produced, and the resulting NBA $\mathcal{B}$ is shown in Figure 3.4.

$b_0 = (\{\mathbf{T}\langle\surd\rangle true, \mathbf{T}(p \wedge (true\ \mathbf{U}^{\mathcal{Y}(y_0)}\ \neg p)), \mathbf{T}p, \mathbf{T}^1(true\ \mathbf{U}^{\mathcal{Y}(y_0)\neg p}),$
$\mathbf{T}(true \wedge \langle\surd\rangle(true\ \mathbf{U}^{\mathcal{Y}(y_1)}\ \neg p)), \mathbf{T}true, \mathbf{T}\langle\surd\rangle(true\ \mathbf{U}^{\mathcal{Y}(y_1)}\ \neg p)\}, 0, \checkmark)$
$b_1 = (\{\mathbf{T}\langle\surd\rangle true, \mathbf{T}true, \mathbf{T}^1(true\ \mathbf{U}^{\mathcal{Y}(y_1)}\ \neg p), \mathbf{T}(\neg p \vee (true \wedge false)), \mathbf{T}\neg p\}, 1, \downarrow)$
$b_2 = (\{\mathbf{T}\langle\surd\rangle true, \mathbf{T}true\}, 1, \checkmark)$
$b_3 = (\{\mathbf{T}\langle\surd\rangle true, \mathbf{T}true\}, 0, \downarrow)$
$b_4 = (\{\mathbf{T}\langle\surd\rangle true, \mathbf{T}true\}, 0, \checkmark)$

The NBA $\mathcal{B}$ has an accepting run $b_0 \xrightarrow{\surd} b_1 \xrightarrow{\surd} b_2 \xrightarrow{\surd} b_3 \xrightarrow{\surd} b_4 \xrightarrow{\surd} b_1 \ldots$. It suggests that a run θ of a constraint automaton satisfies φ if it is of length one, and on the first state, it satisfies p and on the second state it satisfies $\neg p$. Since θ is a finite run, it has the form $s_0 \xrightarrow{\surd} s_0 \xrightarrow{\surd} \ldots$. In other words, the first and second states are the same. However, p and $\neg p$ cannot be satisfied together in the same state. Therefore, the language accepted by the NBA $\mathcal{L}_{\mathcal{N}}(\mathcal{B}(\varphi))$ should have been empty in the first place.

By forcing the accumulation of atomic propositions, the set of signed formulas in b_1 is inconsistent and b_1 does not exists in the NBA $\mathcal{B}'$. Subsequent states are then not generated. As $\mathcal{B}'$ only consists of a single state without no outgoing transition, then no accepting run can be produced by the $\mathcal{B}'$ (recall that all accepting runs in an NBA must be infinite). Therefore, $\mathcal{L}_{\mathcal{N}}(\mathcal{B}'(\varphi)) = \emptyset$ as expected.

3.4.2.1 Correctness of the Extensions

In this section, we show that the satisfiability of a DLTL formula φ can be decided by building the NBA accepting all the models of the formula. The proof is heavily derived from the correctness proof given in [GM06].

First, several notations are introduced.

Let $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, B_0, B_F)$ be an NBA over the alphabet $\Sigma = \mathsf{CIO} \cup \{\surd\}$. Let $\sigma \in \mathsf{CIO}^{\omega} \cup \mathsf{CIO}^*\surd^{\omega}$. $prf(\sigma)$ is the set of finite prefixes of σ and the comparator $\leq$ used on prefixes of σ is the usual prefix ordering over Σ^*. Then a run of $\mathcal{B}$ over σ is a map $\rho : prf(\sigma) \rightarrow B$ such that $\rho(\epsilon) \in B_0$ and $\rho(\tau) \xrightarrow{c} \rho(\tau c)$ for each $\tau c \in prf(\sigma)$ with $c \in \Sigma$. Given a run ρ, $\rho(\tau).\mathcal{F}$ denotes the $\mathcal{F}$ field of the node $\rho(\tau)$ and similar treatment goes for the x and f fields. Given a set $\mathcal{F}$ of signed formulas, $\bigwedge \mathcal{F}$ denotes the conjunction of the formulas in $\mathcal{F}$, where the signs $\mathbf{T}$ are removed and the signs $\mathbf{F}$ are replaced by $\neg$. The sets $Pos(\mathcal{F})$ and $Neg(\mathcal{F})$ are the sets of positive and negative propositions in $\mathcal{F}$ ($Pos(\mathcal{F}) = \{p \in AP \mid \mathbf{T}p \in \mathcal{F}\}$ and $Neg(\mathcal{F}) = \{p \in AP \mid \mathbf{F}p \in \mathcal{F}\}$).

Algorithm 3.2 *create_graph*(φ)

$I := \emptyset$
for each $\mathcal{F} \in tableau(\{\mathbf{T}\varphi\}, 0)$ **do**
  $I := I \cup \{\mathcal{F}, 0, \checkmark\}$
end for
$U := I; V := I; E := \emptyset$
while $U \neq \emptyset$ **do**
  remove $n = (\mathcal{F}, x, f)$ from U
  if $f = \checkmark$ **then**
    $x' := 1 - x$
  else {at least one until formula is not resolved}
    $x' := x$
  end if
  if $\mathbf{T}\langle\surd\rangle true \in \mathcal{F}$ **then**
    $S := tableau_{\surd}(\{\mathbf{T}\varphi \mid \mathbf{T}\langle\surd\rangle\varphi \in \mathcal{F}\} \cup \{\mathbf{F}\varphi \mid \mathbf{F}\langle\surd\rangle\varphi \in \mathcal{F}\}$
      $\cup\{\mathbf{T}p \mid \mathbf{T}p \in \mathcal{F} \wedge p \in AP\} \cup \{\mathbf{F}p \mid \mathbf{F}p \in \mathcal{F} \wedge p \in AP\}, x')$
  else
    $S := tableau(\{\mathbf{T}\varphi \mid \mathbf{T}\langle c\rangle\varphi \in \mathcal{F}\} \cup \{\mathbf{F}\varphi \mid \mathbf{F}\langle c\rangle\varphi \in \mathcal{F}\}, x')$
  end if
  for each $\mathcal{F}' \in S$ **do**
    if $f = \checkmark$ **then**
      $f' :=\downarrow$
    else if there exists no $\mathbf{T}^{x'}\varphi_1\ \mathbf{U}^{\mathcal{Y}_\alpha(q)}\ \varphi_2 \in \mathcal{F}'$ **then**
      $f' := \checkmark$
    else
      $f' :=\downarrow$
    end if
    $n' := (\mathcal{F}', x', f')$
    if $\exists\ n'' \in V$ such that $n'' = n'$ **then**
      $E := E \cup \{(n, c, n'')\}$
    else
      $V := V \cup \{n'\}$
      $E := E \cup \{(n, c, n')\}$
      $U := U \cup \{n'\}$
    end if
  end for
end while
return $\langle V, E, I\rangle$

We also let $M = (\sigma, Val)$ be a model, where $Val : prf(\sigma) \rightarrow 2^{AP}$ is a valuation function. Note that σ cannot be any arbitrary element of Σ^ω as shown in the following lemma.

Lemma 4. *All words that result in an infinite run on the NBA $\mathcal{B}$ generated by Algorithm 3.2 is in $\mathsf{CIO}^\omega \cup \mathsf{CIO}^*\surd^\omega$.*

Proof. This follows from the second extension and that the nonextended version of Algorithm 3.2 serves infinite words over CIO. □

Also, if σ ends with $\surd^\omega$, then for each $\tau \in prf(\sigma)$ such that its last letter is $\surd$ and for each $\tau' > \tau$, $Val(\tau) = Val(\tau')$. This requirement is on par with the requirement of a finite run on the constraint automata that the last state before the run terminates remains the same to the states after the termination.

Lemma 5 (Conversion to constraint automata)**.** *Given a model $M = (\sigma, Val)$ where $\sigma \in CIO^{\omega} \cup CIO^{*}\surd^{\omega}$, we can construct a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0, AP, \mathsf{L})$ such that a run θ of $\mathcal{A}$ is equivalent to M.*

Proof. Let $Q = 2^{AP}$, $\sigma = c_1c_2c_3\ldots$, $Q_0 = \{Val(\epsilon)\}$, $\longrightarrow = \{(s_i, c_i, s_{i+1}) \mid \exists i : q = Val(c_1 \ldots c_i) \wedge s_{i+1} = Val(c_1 \ldots c_{i+1})\}$, $\mathsf{L}(s) = s$ for $s \in Q$. It is clear that the run θ equivalent to M can be defined as $s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$ where $s_i = Val(c_1c_2 \ldots c_i)$ for all $i \geq 0$. □

Using Lemma 5, the satisfaction relation $\models$ for a model $M = (\sigma, Val)$ and a prefix $\tau \in prf(\sigma)$ can be derived easily.

Proposition 6 ([GM06])**.** *Let $\sigma \in CIO^{\omega} \cup CIO^{*}\surd^{\omega}$ and $\rho : prf(\sigma) \to B$ be a (non necessarily accepting) run. For each $\tau \in prf(\sigma)$ and for each $\boldsymbol{T}^{\mathsf{I}}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y)} \ \varphi_2 \in \rho(\tau).\mathcal{F}$ one of the following holds.*

1. *$\forall\tau'$ such that $\tau\tau' \in prf(\sigma) : \boldsymbol{T}^{\mathsf{I}}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y')} \ \varphi_2 \in \rho(\tau\tau').\mathcal{F}$ and $y' \in \delta^*(y, \tau')$*
2. *$\exists\tau'$ such that $\tau\tau' \in prf(\sigma) : \boldsymbol{T}^{\mathsf{I}}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y')} \ \varphi_2 \in \rho(\tau\tau').\mathcal{F}$, $\boldsymbol{T}\varphi_2 \in \rho(\tau\tau').\mathcal{F}$ and no successor node of $\rho(\tau\tau')$ contains an until formula derived from $\boldsymbol{T}^{\mathsf{I}}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y')} \ \varphi_2$. Moreover, for every τ'' such that $\epsilon \leq \tau'' < \tau'$, $\boldsymbol{T}\varphi_1 \in \rho(\tau\tau'').\mathcal{F}$.*

Furthermore, for each $\boldsymbol{F}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y)} \ \varphi_2 \in \rho(\tau).\mathcal{F}$, the following holds.

3. *$\forall\tau'$ such that $\tau\tau' \in prf(\sigma)$: if $\tau' \in \mathcal{L}_{\mathcal{N}}(\mathcal{Y}_\alpha(y))$ then either $\boldsymbol{F}\varphi_2 \in \rho(\tau\tau').\mathcal{F}$ or there is τ'' such that $\epsilon \leq \tau'' < \tau', \boldsymbol{F}\varphi_1 \in \rho(\tau\tau'').\mathcal{F}$.*

Proposition 6 summarizes what is expected of the progress of the until formulas in the NBA according to the tableau rules and Algorithm 3.2. In an accepting run, condition 2 of this proposition must hold for all until formulas and all nodes, as shown by the following theorem.

Theorem 7 ([GM06])**.** *Let $\sigma \in CIO^{\omega} \cup CIO^{*}\surd^{\omega}$ and $\rho : prf(\sigma) \to B$ be a run. Then for each $\tau \in prf(\sigma)$ and for each $\boldsymbol{T}^{\mathsf{I}}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y)} \ \varphi_2 \in \rho(\tau).\mathcal{F}$, condition 2 of Proposition 6 holds if and only if ρ is an accepting run.*

Lemma 8 ([GM06])**.** *Let s be a set of formulas and tableau$(\mathcal{F}) = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$. Then $\bigwedge \mathcal{F} \leftrightarrow \bigvee_{1 \leq i \leq m} \bigwedge \mathcal{F}_i$.*

The previous lemma states that the tableau approach always causes a set of signed formula to support its expansion and vice versa. This lemma then supports the following statement about the properties of the node expansions in Algorithm 3.2.

Lemma 9 ([GM06])**.** *Let $M = (\sigma, Val)$ be a model, $\tau \in prf(\sigma)$, and let $n = (\mathcal{F}, x, f)$ be a node of the graph such that $M, \tau \models \bigwedge \mathcal{F}$. Then, there exists a successor $n' = (\mathcal{F}', x', f')$ of n such that $M, \tau c \models \bigwedge \mathcal{F}'$, where $\tau c \in prf(\sigma)$. Moreover, if $\boldsymbol{T}\varphi_1 \ \boldsymbol{U}^{\mathcal{Y}_\alpha(y)} \ \varphi_2 \in \mathcal{F}'$ where y is a final state and $M, \tau c \models \varphi_2$, then $\boldsymbol{T}\varphi_2 \in \mathcal{F}'$.*

Theorem 10 (Completeness of Algorithm 3.2 [GM06])**.** *Let $M = (\sigma, Val)$ and $M, \epsilon \models \varphi$. Then $\sigma \in \mathcal{L}_{\mathcal{N}}(\mathcal{B}(\varphi))$.*

What is left to show is that Algorithm 3.2 is sound. We divide it into two cases, where $\sigma \in \mathsf{CIO}^{\omega}$ and $\sigma \in \mathsf{CIO}^{*}\surd^{\omega}$.

Theorem 11 (Soundness of Algorithm 3.2 for $\sigma \in \mathsf{CIO}^{\omega}$ [GM06])**.** *Let $\sigma \in CIO^{\omega} \cap \mathcal{L}_{\mathcal{N}}(\mathcal{B}(\varphi))$. Then there is a model $M = (\sigma, Val)$ such that $M, \epsilon \models \varphi$.*

Theorem 12 (Soundness of Algorithm 3.2 for $\sigma \in \mathsf{CIO}^{*}\surd^{\omega}$)**.** *Let $\sigma \in CIO^{*}\surd^{\omega} \cap \mathcal{L}_{\mathcal{N}}(\mathcal{B}(\varphi))$. Then there is a model $M = (\sigma, Val)$ such that $M, \epsilon \models \varphi$.*

Proof. Let ρ be an accepting run. For each $\tau \in prf(\sigma)$ let $\rho(\tau) = (\mathcal{F}_\tau, x_\tau, f_\tau)$. Let $\tau_\surd$ be the shortest prefix of σ such that its last letter is $\surd$. The model $M = (\sigma, Val)$ can be obtained by defining $Val(\tau) \in 2^{AP}$ such that

- if $\tau < \tau_\surd$, $Pos(\mathcal{F}_\tau) \subseteq Val(\tau)$ and $Val(\tau) \cap Neg(\mathcal{F}_\tau) = \emptyset$, and
- if $\tau \geq \tau_\surd$, $Pos(\mathcal{F}_{\tau_{lim}}) \subseteq Val(\tau)$ and $Val(\tau) \cap Neg(\mathcal{F}_{\tau_{lim}}) = \emptyset$, where τ_{lim} is a prefix of σ such that its length (i.e. the number of letters τ_{lim} has) is more than the number of possible consistent sets of signed formulas derived from φ. Note that this number, while exponentially large, is finite since the alphabet and the number of states in the generated NFA which represent the regular I/O expressions are finite. This guarantees that there is no $\tau'' > \tau_{lim}$ such that $Pos(\mathcal{F}_{\tau''}) \supset Pos(\mathcal{F}_{\tau_{lim}})$ or $Neg(\mathcal{F}_{\tau''}) \supset Neg(\mathcal{F}_{\tau_{lim}})$.

By structural induction [GM06], one can prove that for each τ and for each formula φ', if $\mathbf{T}\varphi' \in \mathcal{F}_\tau$ then $M, \tau \models \varphi'$, and if $\mathbf{F}\varphi' \in \mathcal{F}_\tau$ then $M, \tau \not\models \varphi'$. In particular, for until formulas signed with $\mathbf{T}$ Theorem 7 and Proposition 6 condition 2 can be used, while for until formulas signed with $\mathbf{F}$ Proposition 6 condition 3 is used. From $\mathbf{T}\varphi' \in \mathcal{F}_\epsilon$, it follows that $M, \epsilon \models \varphi$. □

Theorem 13. *A DLTL formula φ is satisfiable if and only if $\mathcal{L}_\mathcal{N}(\mathcal{B}(\varphi)) \neq \emptyset$.*

Proof. This follows from Theorem 10, 11, and 12. □

3.4.2.2 Product Rules

Definition 18 is extended to accomodate the $\surd$ transition. In particular, we adopt the terminal rule given in Section 2.3.2.

Definition 24 (Product automata, revisited)**.** Product automaton $\mathcal{A} \times \mathcal{B}$ is a tuple $(Q \times B, \mathcal{N}, \longrightarrow_{\mathcal{A}\times\mathcal{B}}, Q_0 \times B_0, F')$ where $F' \subseteq Q \times B = \{(s, b) \mid b \in F\}$, $b = (\mathcal{F}, x, f)$, $b' = (\mathcal{F}', x', f')$ and $\longrightarrow_{\mathcal{A}\times\mathcal{B}}$ is the smallest relation defined by the rules below.

$$\frac{\begin{array}{cc} s \xrightarrow{c}_\mathcal{A} s' & b \xrightarrow{c}_\mathcal{B} b' \\ \{\varphi \mid \mathbf{T}\varphi \in \mathcal{F} \wedge \varphi \in AP\} \subseteq \mathsf{L}(s) & \{\varphi \mid \mathbf{F}\varphi \in \mathcal{F} \wedge \varphi \in AP\} \cap \mathsf{L}(s) = \emptyset \\ \{\varphi \mid \mathbf{T}\varphi \in \mathcal{F}' \wedge \varphi \in AP\} \subseteq \mathsf{L}(s') & \{\varphi \mid \mathbf{F}\varphi \in \mathcal{F}' \wedge \varphi \in AP\} \cap \mathsf{L}(s') = \emptyset \end{array}}{(s, b) \xrightarrow{c}_{\mathcal{A}\times\mathcal{B}} (s', b')}$$

$$\frac{\begin{array}{cc} s \text{ is terminal} & b \xrightarrow{\surd}_\mathcal{B} b' \\ \{\varphi \mid \mathbf{T}\varphi \in \mathcal{F}' \wedge \varphi \in AP\} \subseteq \mathsf{L}(s) & \{\varphi \mid \mathbf{F}\varphi \in \mathcal{F}' \wedge \varphi \in AP\} \cap \mathsf{L}(s) = \emptyset \end{array}}{(s, b) \xrightarrow{\surd}_{\mathcal{A}\times\mathcal{B}} (s, b')}$$

The notion of an accepting run in this product automaton remains the same as given in Section 2.4.2.3. And since the extensions, in principle, only enlrage the alphabet of the NBA, the DLTL model checking problem with finite run semantics is also PSPACE-complete.

3.5 Fairness in BTSL

As in the problem of CTL model checking with fairness [CE82, BK08], the problem of BTSL model checking with fairness can be reduced to the problem of BTSL model checking (without fairness and with some minor modification) and the problem of computing $Sat_{fair}(\exists\Box a)$ for the atomic proposition a. For the latter, we can use Algorithm 3.4, so we only need to know how to use the BTSL (without fairness) model checking algorithm to compute $Sat_{fair}(\Phi)$. Algorithm 3.3 outlines such an exploitation.

The following lemmas provide the basis for checking subformulas with quantifiers: $\exists\mathbf{U}$, $\exists\mathbf{R}$, $\exists\langle\alpha\rangle$ and $\exists[\alpha]$. In addition, there is a lemma characterizing the fair satisfaction set of $\exists\Box a$ which is used extensively to indicate whether there exists a fair run starting from each state.

Algorithm 3.3 Algorithm for fair BTSL model checking

compute $Sat_{fair}(\exists\Box true) = \{s \in Q \mid FairRuns(s) \neq \emptyset\}$
for all $q \in Sat_{fair}(\exists\Box true)$ **do**
  $L(s) := L(s) \cup \{a_{fair}\}$
end for
for all $i \leq |\Phi|$ **do**
  for all $\Psi \in sub(\Phi)$ with $|\Psi| = i$ **do**
    switch (Ψ):

$true$	:	$Sat_{fair}(\Psi) := Q$
a	:	$Sat_{fair}(\Psi) := \{s \in Q \mid a \in L(s)\}$
$a_1 \vee a_2$	:	$Sat_{fair}(\Psi) := Sat_{fair}(a_1) \cup Sat_{fair}(a_2)$
$\neg a$	:	$Sat_{fair}(\Psi) := Q \backslash Sat_{fair}(a)$
$\exists a_{\Psi_1}$ **U** a_{Ψ_2}	:	$Sat_{fair}(\exists a_{\Psi_1}$ **U** $a_{\Psi_2}) := Sat(\exists a_{\Psi_1}$ **U** $(a_{\Psi_2} \wedge a_{fair}))$
$\exists a_{\Psi_1}$ **R** a_{Ψ_2}	:	$Sat_{fair}(\exists a_{\Psi_1}$ **R** $a_{\Psi_2}) := Sat(\exists a_{\Psi_2}$ **U** $(a_{\Psi_1} \wedge a_{\Psi_2} \wedge a_{fair})) \cup Sat_{fair}(\exists\Box a_{\Psi_2})$
$\exists\langle\alpha\rangle a_{\Psi_1}$	:	$Sat_{fair}(\exists\langle\alpha\rangle\Psi_1) := Sat_{fair\exists\langle\cdot\rangle}(\exists\langle\alpha\rangle\Psi_1)$ (See Lemma 16)
$\exists[\alpha] a_{\Psi_1}$	:	$Sat_{fair}(\exists[\alpha]\Psi_1) := Sat_{fair\exists[\cdot]}(\exists[\alpha]\Psi_1)$ (See Lemma 17)

    end switch
    if Ψ is a state formula **then**
      $AP := AP \cup \{a_\Psi\}$
      replace every occurrence of Ψ with a_Ψ
      for all $s \in Sat_{fair}(\Psi)$ **do**
        $L(s) := L(s) \cup \{a_\Psi\}$
      end for
    end if
  end for
end for
return $Q_0 \subseteq Sat_{fair}(\Phi)$

Lemma 14 (Until for a fair satisfaction relation [BK08])**.** *$s \models_F \exists a_1$ **U** a_2 iff there exists a finite run prefix $prefix(\theta, k) = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} s_k$ with $0 \leq k < |\theta|$ and $\theta \in MaxRuns(s)$ such that $s_i \models a_1$ for $0 \leq i < k$, $s_k \models a_2$, and $FairRuns(s_k) \neq \emptyset$.*

Lemma 15 (Release for a fair satisfaction relation)**.** *$s \models_F \exists a_1$ **R** a_2 iff at least one of the following is satisfied.*

1. *there exists a finite run prefix $prefix(\theta, k) = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} s_k$ with $0 \leq k < |\theta|$ and $\theta \in MaxRuns(s)$ such that $s_k \models a_1$, for $0 \leq i \leq k$, $s_i \models a_2$, and $FairRuns(s_k) \neq \emptyset$.*
2. *$s \in Sat_{fair}(\exists\Box a_2)$.*

Proof. $\Rightarrow$: Assume $s \models_F \exists a_1$ **R** a_2. Then, at least one of the following cases is true.

1. there exists a fair run $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$ where $s_0 = s$, $s_k \models a_1$, $k \geq 0$, $s_i \models a_2$ for all $0 \leq i \leq k$. Since θ is fair, then $suffix(\theta, k)$ is also fair. Therefore, $FairRuns(s_k) \neq \emptyset$.
2. There exists a fair run $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$ where $s_0 = s$ and for all $0 \leq i < |\theta|$, $s_i \models a_2$. Thus, $s \in Sat_{fair}(\exists\Box a_2)$.

$\Leftarrow$: There are two cases.

1. Assume $s \in Sat_{fair}(\exists\Box a_2)$. Then there exists a fair run θ starting in s such that $\theta \models a_1$ **R** a_2. Therefore, by the semantics of $\models_F$, $s \models_F \exists a_1$ **R** a_2.

2. Assume that there exists a finite run prefix $s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} s_k$ with $k \geq 0$ and for all $0 \leq i \leq k$ $s_i \models a_2$, such that $s_k \models a_1$ and $\mathit{FairRuns}(s_k) \neq \emptyset$. Thus, there is a fair run $\theta' = s_k \xrightarrow{c_{k+1}} s_{k+1} \xrightarrow{c_{k+2}} \ldots$ starting in s_k. Consequently, $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} s_k \xrightarrow{c_{k+1}} \ldots$ is a fair run starting in s such that $\theta \models a_1 \mathbf{R} a_2$. Therefore, $s \models_F \exists a_1 \mathbf{R} a_2$.

□

Computing the fair satisfaction sets of the conditional future ($\exists\langle\alpha\rangle\Phi$) and the conditional always ($\exists[\alpha]\Phi$) operators without fairness assumption reduces to CTL model checking as per Theorem 2. Therefore, on the product automaton, we can use the fair satisfaction check for CTL with some modification. The following lemmas provide the basis of the modification.

Lemma 16 (Conditional future for a fair satisfaction relation)**.** *$s \models_F \exists\langle\alpha\rangle a$ iff there exists a finite accepting run $\theta' = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k)$ with $k = |\theta'|$, $s_0 = s$, and z_0 is an initial state of $\mathcal{Z}$, $\mathit{ios}(\theta') \in \mathcal{L}_{\mathcal{N}}(\alpha)$, $s_k \models a$ and $\mathit{FairRuns}(s_k) \neq \emptyset$.*

Proof. $\Rightarrow$: Assume $s \models_F \exists\langle\alpha\rangle a$. By the $\models_F$ semantics, there exists a fair run $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots$ such that there exists $k \geq 0$ such that $s_k \models a$, and $\mathit{ios}(\mathit{prefix}(\theta, k)) \in \mathcal{L}_{\mathcal{N}}(\alpha)$. Since θ is a fair run, $\mathit{suffix}(\theta, k)$ is also a fair run. Thus, $\mathit{FairRuns}(s_k) \neq \emptyset$. Furthermore, since $\mathit{ios}(\mathit{prefix}(\theta, k)) \in \mathcal{L}_{\mathcal{N}}(\alpha)$, there is an accepting run $\theta' = z_0 \xrightarrow{c_1} z_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} z_k$ of the NFA $\mathcal{Z}$ with $z_0 \in Z_0$ running parallel to the system's constraint automaton. Therefore, there exists a finite accepting run $\theta' = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k)$ in the product automaton $\mathcal{A} \times \mathcal{Z}$.
$\Leftarrow$: Assume that there exists a finite accepting run $\theta' = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k)$ with $k = |\theta'|$, $s_0 = s$, and z_0 is an initial state of $\mathcal{Z}$, $\mathit{ios}(\theta') \in \mathcal{L}_{\mathcal{N}}(\alpha)$, $s_k \models a$ and $\mathit{FairRuns}(s_k) \neq \emptyset$. Since θ' is finite, then $\theta = s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots s_k \xrightarrow{s_{k+1}} \ldots$ is a fair run as well. Since $\theta \models_F \langle\alpha\rangle a$, then $s \models_F \exists\langle\alpha\rangle a$. □

Lemma 17 (Conditional always for a fair satisfaction relation)**.** *If $\mathcal{A}$ is deterministic, then $s \models_F \exists[\alpha] a$ iff one of the following conditions is achieved.*

- *there exists a finite run $\theta' = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{\checkmark} (s_n, z_n)$ where $s_0 = s$ and $z_0 \in Z_0$ such that for all $0 \leq i \leq n$, $s_i \models a$ or $z_i \models \neg \mathit{final}$.*
- *there exists a finite run prefix $\mathit{prefix}(\theta', k) = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k)$ and a cycle run $(s_k, z_k) \xrightarrow{c_{k+1}} (s_{k+1}, z_{k+1}) \xrightarrow{c_{k+2}} \ldots \xrightarrow{c_l} (s_l, z_l) \xrightarrow{c_{l+1}} (s_k, z_k)$ such that for all $0 \leq i \leq l$, $s_i \models a$ or $z_i \models \neg \mathit{final}$ and $s_0 = s$. The cycle satisfies the fairness condition F.*

Proof. Assume $\mathcal{A}$ is deterministic.
$\Rightarrow$: Assume $s \models_F \exists[\alpha] a$. Then by Theorem 2 there exists a run θ' in the product automaton $\mathcal{A} \times \mathcal{Z}$ starting in (s_0, z_0) where z_0 is the initial state of $\mathcal{Z}$ and $s_0 = s$ such that $\theta' \models \Box(a \vee \neg final)$. By the semantics of $\models_F$, the projection of the run θ to $\mathcal{A}$ (i.e., leaving out the states of $\mathcal{Z}$) is a fair run. By Lemma 18 for the product automaton and considering only the projection to $\mathcal{A}$ for conditions 4 and 5, we find that either one of the following is true.

- θ' is a finite run such that for all $0 \leq i \leq |\theta'|$, $s_i \models a$ or $z_i \models \neg \mathit{final}$.
- θ' is an infinite run that starts with a finite run prefix $\mathit{prefix}(\theta', k) = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k)$ followed by a cycle run $(s_k, z_k) \xrightarrow{c_{k+1}} (s_{k+1}, z_{k+1}) \xrightarrow{c_{k+2}} \ldots \xrightarrow{c_l} (s_l, z_l) \xrightarrow{c_{l+1}} (s_k, z_k)$ such that for all $0 \leq i \leq l$, $s_i \models a$ or $z_i \models \neg \mathit{final}$ and the cycle satisfies the fairness condition F.

$\Leftarrow$: Assume there exists a finite run $\theta' = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{\checkmark} (s_n, z_n)$ where $s_0 = s$ and $z_0 \in Z_0$ such that for all $0 \leq i \leq n$, $s_i \models a$ or $z_i \models \neg \mathit{final}$. Since it is a finite run, the projection θ to $\mathcal{A}$ is a fair run. From Theorem 2, it follows that $\theta' \models [\alpha] a$, and therefore, $s \models_F \exists[\alpha] a$.

Assume there exists a finite run prefix $\mathit{prefix}(\theta', k) = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k)$ and a cycle run $(s_k, z_k) \xrightarrow{c_{k+1}} (s_{k+1}, z_{k+1}) \xrightarrow{c_{k+2}} \ldots \xrightarrow{c_l} (s_l, z_l) \xrightarrow{c_{l+1}} (s_k, z_k)$ where $s_0 = s$ such that for all $0 \leq i \leq l$, $s_i \models a$ or $z_i \models \neg \mathit{final}$ and the cycle satisfies the fairness condition F. Consider the infinite run $\theta' = (s_0, z_0) \xrightarrow{c_1} (s_1, z_1) \xrightarrow{c_2} \ldots \xrightarrow{c_k} (s_k, z_k) \xrightarrow{c_{k+1}} (s_{k+1}, z_{k+1}) \xrightarrow{c_{k+2}} \ldots \xrightarrow{c_l} (s_l, z_l) \xrightarrow{c_{l+1}} (s_k, z_k) \ldots$. As θ' ends with an infinite cycle that satisfies the fairness condition F, it follows that θ' is a fair run. From Theorem 2, it follows that $\theta' \models [\alpha]a$. Thus, $s \models_F \exists[\alpha]a$. □

Lemma 18 (Characterization of $\mathit{Sat}_{\mathit{fair}}(\exists\Box a)$ [BK08])**.** *Assume that the fairness assumption F contains conjunctions of the following fairness conditions:*

- $\mathit{ufair}_1 = \Box\Diamond \mathit{taken}(D_{u_1})$, …, $\mathit{ufair}_m = \Box\Diamond \mathit{taken}(D_{u_m})$,
- $\mathit{sfair}_1 = (\Box\Diamond \mathit{enabled}(C_{s_1}) \rightarrow \Box\Diamond \mathit{taken}(D_{s_1}))$, …,
  $\mathit{sfair}_n = (\Box\Diamond \mathit{enabled}(C_{s_n}) \rightarrow \Box\Diamond \mathit{taken}(D_{s_n}))$, *and*
- $\mathit{wfair}_1 = (\Diamond\Box \mathit{enabled}(C_{w_1}) \rightarrow \Box\Diamond \mathit{taken}(D_{w_1}))$, …,
  $\mathit{wfair}_o = (\Diamond\Box \mathit{enabled}(C_{w_o}) \rightarrow \Box\Diamond \mathit{taken}(D_{w_o}))$.

$s \models_F \exists\Box a$ iff there exists a finite run starting in s such that all states in that run satisfy a or there exists an infinite run starting in s consisting of a fair run prefix $s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} s_k$ and a cycle $s_k \xrightarrow{c_{k+1}} s_{k+1} \xrightarrow{c_{k+2}} \ldots \xrightarrow{c_{k+l}} s_{k+l} \xrightarrow{c_{k+l+1}} s_k$ such that:

1. $s_0 = s$,
2. $s_i \models a$, *for all* $0 \leq i \leq k + l$,
3. $D_{u_i} \cap \{c_{k+1}, \ldots, c_{k+l+1}\} \neq \emptyset$ *for all* $1 \leq i \leq m$,
4. $\mathit{Sat}(\mathit{enabled}(C_{s_i})) \cap \{s_k, \ldots, s_{k+l}\} = \emptyset$ *or* $D_{s_i} \cap \{c_{k+1}, \ldots, c_{k+l+1}\} \neq \emptyset$ *for all* $1 \leq i \leq n$,
5. $\{s_k, \ldots, s_{k+l}\} \not\subseteq \mathit{Sat}(\mathit{enabled}(C_{w_i}))$ *or* $D_{w_i} \cap \{c_{k+1}, \ldots, c_{k+l+1}\} \neq \emptyset$ *for all* $1 \leq i \leq o$.

Lemma 18 provides the necessary material to compute $\mathit{Sat}_{\mathit{fair}}(\exists\Box a)$. First we define the restricted graph $G[a] = (V, E)$ as a directed graph derived from $\mathcal{A}$, where $V = \mathit{Sat}(a)$ and $E = \{s \xrightarrow{c} s' \mid s, s' \in V \text{ and } s \xrightarrow{c}_{\mathcal{A}} s'\}$. Due to the construction of the graph, infinite runs generated by $G[a]$ correspond to the infinite runs of $\mathcal{A}$ that satisfy $\Box a$. The same apply for the converse.

A state $s \in \mathit{Sat}(a)$ satisfies $\exists\Box a$ under the fairness assumption $F = \mathit{ufair} \wedge \mathit{sfair} \wedge \mathit{wfair}$ if we can construct a run starting in s that reaches a non-trivial strongly connected set of nodes SC in $G[a]$ such that

- for all $1 \leq i \leq m$: there exists $c \in D_i$ such that $\{s \xrightarrow{c} s' \mid s, s' \in SC\} \cap E \neq \emptyset$,
- for all $1 \leq i \leq n$: $SC \cap \mathit{Sat}(\mathit{enabled}(C_i)) = \emptyset$ or there exists $c \in D_i$ such that $\{s \xrightarrow{c} s' \mid s, s' \in SC\} \cap E \neq \emptyset$, and
- for all $1 \leq i \leq o$: $SC \cap \mathit{Sat}(\neg\mathit{enabled}(C_i)) \neq \emptyset$ or there exists $c \in D_i$ such that $\{s \xrightarrow{c} s' \mid s, s' \in SC\} \cap E \neq \emptyset$

We collect all nodes s of $G[a]$ into the set T such that it is part of a non-trivial strongly connected component in $G[a]$ that realizes the fairness assumption. Furthermore, we also include the terminal states that satisfy the atomic proposition a into T. Then it follows:

$$\mathit{Sat}_{\mathit{fair}}(\exists\Box a) = \{s \in S \mid \mathit{Reach}_{G[a]}(s) \cap T \neq \emptyset\} \ .$$

Algorithm 3.4 Computation of $Sat_{fair}(\exists\Box a)$

Require:

$S := Sat(a)$

$F := ufair \wedge sfair \wedge wfair$

$ufair = \bigwedge_{1\leq i\leq m} \Box\Diamond taken(D_{u_i})$

$sfair = \bigwedge_{1\leq i\leq n} \Box\Diamond enabled(C_{s_i}) \rightarrow \Box\Diamond taken(D_{s_i})$

$wfair = \bigwedge_{1\leq i\leq o} \Diamond\Box enabled(C_{w_i}) \rightarrow \Box\Diamond taken(D_{w_i})$

$Term := \{s \mid s \in Sat(a) \wedge s \text{ is a terminal state in } \mathcal{A}\}$

1: Compute the SCCs $Comp$s of the graph $G[a]$ of $\mathcal{A}$
2: $T := \emptyset$
3: **for all** non-trivial SCC $Comp$ of the graph $G[a]$ of $\mathcal{A}$ **do**
4: **if** $checkFair(Comp, m, ufair, n, sfair, o, wfair)$ **then**
5: $T := T \cup Comp$
6: **end if**
7: **end for**
8: **return** $\{s \in Q \mid Reach_{G[a]}(s) \cap (T \cup Term) \neq \emptyset\}$

$Reach_{G[a]}(s)$ is the set of states reachable from s in $G[a]$. Note that if SC is a part of a strongly connected component $Comp$ in $G[a]$, then we can start an infinite run from any of the states in $Comp$ that satisfies the fairness assumption and $\Box a$. This general schema is given in Algorithm 3.4.

What is left is determining whether the SCC $comp$ of $G[a]$ actually realizes the fairness assumption. We are interested more in the strong fairness case as the realizability of the unconditional and weak fairness assumptions can be checked at the end after we find the non-trivial strongly connected set of nodes SC that satisfies the strong fairness assumption.

Given a non-trivial strongly connected component $Comp$ of $G[a]$, first we check whether for all transition conditions D_i there exists a transition in $G[a]$ restricted to $Comp$ whose label is in D_i. If yes, then the strong fairness assumption is realizable in $Comp$ (and we proceed checking the other fairness assumptions). Otherwise, we pick one strong fairness conjunct $sfair_j$ such that the transition condition D_j is not fulfilled. In this case, we must not fulfill the left hand side of the implication. Therefore, we calculate the non-trivial SCCs $Comp'$ of $G[a]$ restricted to $Comp$ in which every state has C_j not enabled. Then we check for each of the nontrivial SCCs $Comp'$ whether it satisfies the rest of the strong fairness assumption. By doing so, we guarantee that $Comp'$ satisfies $sfair_j$. If there is no nontrivial SCCs that satisfy the rest of the strong fairness assumption (or there is no SCC produced to begin with), then we conclude that the strong fairness, and thus the whole fairness, assumption is not realizable by the SCC $Comp$. This recursive procedure is listed as Algorithm 3.5.

Theorem 19 (Time complexity of computing the fair satisfaction set of $\exists\Box a$ [BK08])**.** *For a constraint automaton $\mathcal{A}$ with N states and K transitions, and fairness assumption F with k conjuncts, the set $Sat_{fair}(\exists\Box a)$ can be computed in $O((N + K) \cdot k)$.*

3.6 Chapter Summary

BTSL*, a CTL*-like extension of BTSL, is the main topic of this chapter. We can use BTSL* to express concurrent I/O operation-based fairness assumptions, which allows us to ignore the sources of unrealistic computation of the Reo connector: domination of an internal component and unfair nondeterministic behavior of the sink/mixed nodes.

Algorithm 3.5 *checkFair*$(Comp, m, ufair, n, sfair, o, wfair)$

Require:
$Comp$ is a non-trivial SCC of $G[a]$

1: $TransLabel := \{c \mid s, s' \in Comp \text{ and } s \xrightarrow{c}_{G[a]} s'\}$
2: **if** $\forall 1 \leq i \leq n : D_{s_i} \cap TransLabel \neq \emptyset$ **then**
3: **if** $\forall 1 \leq i \leq m : D_{u_i} \cap TransLabel \neq \emptyset$ **then**
4: **if** $\forall 1 \leq i \leq o : Comp \backslash Sat(enabled(C_{w_i})) \neq \emptyset \vee D_{w_i} \cap TransLabel \neq \emptyset$ **then**
5: **return** true
6: **end if**
7: **end if**
8: **else**
9: choose an index $j \in \{1, \ldots, n\}$ where $TransLabel \cap D_{s_j} = \emptyset$
10: $sfair' = \bigwedge_{1 \leq i < j} \Box\Diamond enabled(C_{s_i}) \rightarrow \Box\Diamond taken(D_{s_i}) \wedge \bigwedge_{j < i \leq n} \Box\Diamond enabled(C_{s_i}) \rightarrow \Box\Diamond taken(D_{s_i})$
11: renumber the index in $sfair'$ accordingly
12: **if** $Comp[\neg enabled(C_j)]$ is acyclic or empty **then**
13: **return** false
14: **else**
15: compute the non-trivial SCCs of $Comp[\neg enabled(C_{s_j})]$
16: **for all** non-trivial SCC $Comp'$ of $Comp[\neg enabled(C_{s_j})]$ **do**
17: **if** *checkFair*$(Comp, m, ufair, n-1, sfair', o, wfair)$ **then**
18: **return** true
19: **end if**
20: **end for**
21: **end if**
22: **end if**
23: **return** false

Checking a model against a BTSL* formula requires an algorithm similar to the model-checking algorithm for CTL*. The main difference lies on the base tool used, which in the case of BTSL* is the DLTL model checker presented in Section 3.4.2. The DLTL model checker is extended to handle maximal finite runs.

Chapter 4

Implementation and Experimental Result

This chapter presents the experimental results I obtained by using the restricted BTSL* model checker to discover whether a Reo circuit satisfies desired properties. The first section explains why the restricted BTSL* is used as the base logic for property specification. The second section provides a quick run through the data structure used for symbolic representation of the (constraint) automata. The third section describes how the experiment is conducted and the fourth section states the obtained results.

4.1 Restricted Logic

In the implementation, I use the following fragment of BTSL* as the underlying logic to specify properties.

$$\Phi ::= true \mid p \mid \neg\Phi_1 \mid \Phi_1 \vee \Phi_2 \mid \exists\langle\alpha\rangle\Phi \mid \exists\ \varphi \mid \forall\ \varphi$$
$$\varphi ::= \Phi \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi_1 \mid \varphi_1\ \mathbf{U}\ \varphi_2$$
$$\alpha ::= ioc \mid \alpha_1 \cup \alpha_2 \mid \alpha^* \mid \alpha_1; \alpha_2$$

In short, the regular expression based operator $\langle\alpha\rangle$ is not allowed to nest with other path operators.

The main reason of this restriction is that the best known DLTL model checking algorithm generates a huge NBA even for a simple formula with a very restricted alphabet (recall that for a simple path formula $\varphi = \Box\langle(a;a)^+\rangle p$ of size 6 and a singleton alphabet, the NBA has 34 states). Another reason is that this restriction allows the usage of the proposition-based $\text{LTL}_{\text{I/O}}$ [Kle] and the BTSL implementation [KB07] that already exists within Vereofy.

4.2 Symbolic Representations

Vereofy provides a symbolic BTSL model checker implementation by means of *ordered binary decision diagrams* (OBDDs) [Bry86] which is reused for BTSL* model checking. Binary decision diagrams are a data structure to store and manipulate boolean functions. Any boolean function $f(x_1, \ldots, x_h)$ can be represented in a natural manner by a binary decision tree (BDT) of height h. A path in the BDT for $f(x_1, \ldots, x_h)$ assigns bits $b_1, \ldots, b_h$ to the x_i's and the leaves are labelled with the boolean value $f(x_1, \ldots, x_h)$. By grouping together nodes that share the same property in a bottom up manner, we have a binary decision diagram (BDD). By placing some variable ordering, we have a canonical representation of the BDD as shown in [Bry86], which is the OBDD.

The variable order of a BDD determines the size of the BDD. A bad variable order can cause the size of the BDD to be exponential in the number of variables [MT98]. Finding the optimal

variable order is NP-complete [BW96], and techniques such as static heuristics [BRKM91] and dynamic variable reordering [Rud93] are used to find a good variable order. Vereofy uses static heuristics to order the variables in the BDD, and some experiments are now being done on dynamic variable ordering.

A constraint automaton $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0, AP, L)$ is represented using OBDD in the following manner. Each state is encoded as a binary number, i.e. $Q = \{0,1\}^n$ where $n = \lceil |\log_2(Q)| \rceil$. Therefore, a set $V \subseteq Q$ is a set of binary strings of length n whose characteristic function $\chi_V(q_1, \ldots, q_n)$ can be represented by an OBDD. We can also encode the data items by binary strings. We assume without loss of generality that the size of the data domain is a power of two (2^m), i.e. $\mathcal{D} = \{0, 1, \ldots, 2^m - 1\}$. Let $k = |\mathcal{N}|$, $\mathcal{N} = \{A_1, \ldots, A_k\}$, and $\forall\ 1 \leq i \leq k\ \{d_{i_1}, \ldots, d_{i_m}\}$ = the binary representation of d_{A_i}. The transition relation $\longrightarrow \subseteq \mathcal{Q} \times \mathsf{CIO} \times \mathcal{Q}$ is a set of binary strings of length $2(n+k)$ whose characteristic function $\chi_{\longrightarrow}(q_1, \ldots, q_n, A_1, \ldots, A_k, d_{1_1}, \ldots, d_{1_m}, \ldots, d_{k_1}, \ldots, d_{k_m}, q'_1, \ldots, q'_n)$ can be represented by an OBDD. NBA and product automata are also represented in a similar manner.

4.3 Benchmarks

To evaluate the effectiveness of checking restricted BTSL* formulas, we use the dining philosopher component illustrated in Figure 2.8 as the base component. Instead of using the 4-state philosopher component, we use the 3-state philosopher component where a philosopher puts down both the left and right chopsticks at the same time after he finishes eating.

The arrangement of a connector with N dining philosophers (numbered from 1 to N) and N chopsticks (also numbered from 1 to N) is such that the left chopstick of philosopher i is chopstick i and the right chopstick is chopstick $(i \mod N) + 1$. Note that this configuration allows a deadlock to happen, which occurs when all philosophers decide to pick up their left chopstick before a philosopher has the chance to pick up his right chopstick.

Benchmarking is done on a dual-core 3.0 GHz CPU, with 2 GB of RAM, running on Ubuntu 8.04.1 kernel 2.6.24-21-generic. The compiler used is GCC C++ compiler version 4.2.4. Vereofy uses JINC BDD library [Oss] and Boost C++ libraries 1.36 (http://www.boost.org/). The time usage is measured using Vereofy's internal timer (based on `sys/resource.h` standard library) and the memory usage is measured using `top`. The result is an average of three runs.

4.4 Results

Table 4.1 shows the efficiency of Vereofy to construct the BDD-representation of the constraint automaton $\mathcal{A}$ for the dining philosopher connector with N philosophers. The second column "time" shows the time needed (in seconds) to build the automaton (the process is also known as synthesis). The third column "reach time" refers to the time needed (also in seconds) to compute the reachable states of $\mathcal{A}$ from the initial states. The fourth column and fifth column refer to the size of the generated BDD for $\mathcal{A}$ and the maximal size of the BDDs generated during the symbolic computation. The last column shows the actual memory usage (in mega bytes).

Table 4.1: Synthesis result for the dining philosophers

N	Time (s)	Reach Time (s)	BDD Nodes	Peak	RAM (MB)
80	2.09	0.44	687846	1152461	69
160	9.98	0.73	2700396	2700396	102
240	26.01	0.69	1993891	4137866	140
320	49.25	2.38	5482017	5482017	214
400	82.43	1.75	3271773	7027277	196
480	122.20	3.61	8007073	8403634	414
560	173.40	1.66	4190811	10094673	550
640	230.38	4.70	10512682	11306851	540
720	300.50	2.61	5196999	12910887	864
800	375.16	5.73	12635355	14334430	618

Table 4.2: Result of model checking BTSL formula $\forall\Box\neg(eat_i \wedge eat_{i+1})$

N	Time (s)	Steps	BDD Nodes	Peak	RAM (MB)
80	0.89	159	2195124	2195124	90
160	4.68	319	6049759	6049759	168
240	12.38	479	1728577	7835455	206
320	21.85	639	5590500	11209762	276
400	36.56	799	1520738	15618855	364
480	51.18	959	6690790	19899858	456
560	70.11	1119	7856066	21951784	498
640	92.45	1279	6156418	21716513	496
720	120.84	1439	7889178	24672646	558
800	142.74	1599	6729310	39000710	840

Table 4.3: Result of model checking BTSL formula $\forall\Box\exists\langle\mathsf{CIO}^*; take\_right_i\rangle true$

N	Time (s)	Steps	BDD Nodes	Peak	RAM (MB)
80	1.97	160	3369787	3369787	112
160	11.48	320	896589	4688146	142
240	26.58	480	1357036	7261999	194
320	48.79	640	1142630	10784052	266
400	77.37	800	2788294	15475977	360
480	114.20	960	1730963	19731680	446
560	158.79	1120	2018918	21952943	492
640	208.85	1280	2985991	21596949	488
720	269.30	1440	2823158	24672646	552
800	331.14	1600	2829429	39000710	836

Table 4.4: Result of model checking BTSL formula $\exists\langle \mathsf{CIO}^*; take_i; take_{i+1}\rangle eat_i$

N	Time (s)	Steps	BDD Nodes	Peak	RAM (MB)
80	0.64	2	1221396	1221396	69
160	0.91	2	2121359	2700456	102
240	0.15	2	4722890	4722890	144
320	0.26	2	8307130	8307130	216
400	0.70	2	13017405	13017405	311
480	0.33	2	17430914	17430914	400
560	0.61	2	19572314	19572314	444
640	0.41	2	19402964	19402964	444
720	0.82	2	17441093	24672646	552
800	0.75	2	12810979	39000710	836

Tables 4.2, 4.3 and 4.4 show the result of model checking properties that are suggested in [KB07] as the properties for benchmarking. Table 4.2 shows that verifying that no two neighboring philosophers can eat at the same time on a connector with 800 philosophers can be done in 2 minutes and used 840 MB of memory. Table 4.3 shows the result of checking that deadlock cannot happen in this connector. As in [KB07], the model checking result is negative as it is possible for a deadlock to happen in the case that all philosophers have taken their left chopsticks before anyone can pick up their right chopsticks. Table 4.4 shows that checking that philosopher i eats after chopstick i and chopstick $i+1$ (note that here it is not enforced that these two chopsticks must be taken by philosopher i) are taken can be done in 2 symbolic steps.

Comparison between the implementation of BTSL* and the implementations of $\text{LTL}_{\text{I/O}}$ and BTSL model checkers is also done. Since the restricted BTSL* model checker adopts the same approach to solve PDL-derived formulas, there is no need to compare the execution result for the formulas $\forall\Box\exists\langle \mathsf{CIO}^*; take\_right_i\rangle true$ and $\exists\langle \mathsf{CIO}^*; take_i; take_{i+1}\rangle eat_i$. So in this section, we focus mainly on the formula $\forall\Box\neg(eat_i \wedge eat_{i+1})$ and also on another formula $\exists\Box\neg eat_i$ which stresses that there exists a situation where a philosopher starves.

Tables 4.5 and 4.6 show the execution of restricted BTSL* and $\text{LTL}_{\text{I/O}}$ model checkers, respectively, for the formula $\forall\Box\neg(eat_i \wedge eat_{i+1})$ (we drop the $\forall$ in $\text{LTL}_{\text{I/O}}$). As can be seen, model checking the philosopher connector against the desired property in restricted BTSL* is significantly slower than in $\text{LTL}_{\text{I/O}}$, and $\text{LTL}_{\text{I/O}}$ model checking is several times slower than the BTSL. The main reason why checking formulas in restricted BTSL* is slower than in $\text{LTL}_{\text{I/O}}$ is that we need to build the satisfaction set of all possible states in the constraint automaton of the connector. In $\text{LTL}_{\text{I/O}}$, model checking is focused only on the initial states. Since the number of initial states and the number of all states usually differ a lot, then the time needed to compute SCCs on reachable states from the set of initial states and the set of all states differ significantly as well.

Table 4.5: Result of model checking BTSL* formula $\forall\Box\neg(eat_i \wedge eat_{i+1})$

N	Time (s)	BDD Nodes	Peak	RAM (MB)
10	0.94	819943	819943	59
20	3.36	3002352	3002352	104
30	6.61	5504888	5504888	154
40	11.37	8264870	8264870	210
50	18.49	11593658	11593658	276
60	41.21	14870620	14870620	342
70	69.28	18652370	18652370	416
80	104.86	15916290	15916290	364
90	142.73	20313997	20313997	455
100	185.87	20137130	20137130	452

Table 4.6: Result of model checking $\text{LTL}_{\text{I/O}}$ formula $\Box\neg(eat_i \wedge eat_{i+1})$

N	Time (s)	BDD Nodes	Peak	RAM (MB)
80	2.45	1822151	1822151	104
160	18.99	450390	2924930	144
240	44.92	1770152	4545174	198
320	84.55	2385384	8159521	271
400	138.41	2943651	7066193	362
480	205.56	3121398	18076272	451
560	290.09	2444164	24896585	493
640	390.76	2134872	24310764	494
720	528.19	2235998	40580124	559
800	672.66	1504576	27790452	841

In Tables 4.7 and 4.8, the results for model checking the philosopher connector against the formula $\exists\Box\neg eat_i$ in BTSL and restricted BTSL* are shown. As expected, the time needed to verify the formula in restricted BTSL* is exponentially slower than in BTSL. Also, the restricted BTSL* formula consumes a lot more memory by virtue of the product automaton construction. In both cases, the formula is satisfied, which means there is a situation where a philosopher starves. This is as expected since the possibility of an unfair behavior where all other philosophers are infinitely faster than philosopher i is not excluded.

Table 4.7: Result of model checking BTSL* formula $\exists\Box\neg eat_i$

N	Time (s)	BDD Nodes	Peak	RAM (MB)
8	2.61	1816765	1816765	81
16	34.32	18146750	18146750	415
24	92.11	42527045	42527045	896
32	182.05	68956027	68956027	1417
40	322.73	75932014	75932014	1560

Table 4.8: Result of model checking BTSL formula $\exists\Box\neg eat_i$

N	Time (s)	Steps	BDD Nodes	Peak	RAM (MB)
40	0.87	21	319096	494369	44
80	1.28	41	1244916	1244916	68
120	3.12	61	2775536	2775536	101
160	24.71	81	2999582	4020898	128
200	234.41	101	3413162	3999046	130

To exclude such unfair behavior, we need the strong fairness described in Section 3.2. Here we require that the strong fairness formula to be

$$\forall(1 \leq i \leq N) : (\Box\Diamond enabled(\{take\_left_i\}) \rightarrow \Box\Diamond take\_left_i) \wedge (\Box\Diamond enabled(\{take\_right_i\}) \rightarrow \Box\Diamond take\_right_i) \ .$$

Note that it is not enough to apply the strong fairness condition only to a certain philosopher, since the other non-neighboring philosophers may act unfairly.

Table 4.9 shows the effect of applying such a fairness condition to the restricted BTSL* model checker. The "delta BDD nodes" column shows the number of nodes needed to represent the transition relation for the constraint automata of the philosopher connector. The blow up in the resulting product automaton is mainly caused by the NBA built from the path formula. Table 4.10 shows the size of the NBA based on the number of states and the number of symbolic edges (here symbolic means that an edge is labeled with an I/O constraint or the satisfaction set for states where *enabled* clause is true).

Table 4.9: Result of model checking BTSL* formula $\exists\Box\neg eat_i$ with strong fairness

N	Time (s)	Delta BDD Nodes	BDD Nodes	Peak	RAM (MB)
2	8.18	132	669997	669997	274
3	407.21	292	33390143	33390143	1510
4	N/A	450	N/A	N/A	Out of memory

Table 4.10: Characteristics of Generated NBA

N	Formula	States	Symbolic Edges
Arbitrary	$\neg\Box\neg(eat_i \wedge eat_{i+1})$	2	3
Arbitrary	$\neg\neg\Box\neg eat_i$	1	1
2	$sfair \rightarrow \neg\Box\neg eat_i$	114	461
3	$sfair \rightarrow \neg\Box\neg eat_i$	922	5647
4	$sfair \rightarrow \neg\Box\neg eat_i$	7582	76969

4.5 Chapter Summary

In this chapter, some implementation details are explained including why the underlying logic of BTSL* is restricted. The implementation result shows that model checking arbitrary BTSL* is not yet practical. The main culprit is the $Sat_{\mathrm{LTL}_{\mathrm{I/O}}}$ procedure where the emptiness of the set

of accepting runs starting from a state needs to be checked for every combination of states of the system's constraint automaton and initial states of the NBA. We also see that the general framework of Vereofy is relatively efficient even compared to the fine-tuned result given in [KB07].

Chapter 5

Conclusion

This thesis presents BTSL* as the counterpart of CTL* to reason about the states of a Reo circuit and how the data flows in runs within the circuit. Here we remove the assumption that all runs are infinite and give a semantics and the corresponding model checking algorithm for both finite and infinite runs. This removal proves to be non-trivial, but it does not change the time nor space complexity of the algorithm.

Another aspect explored in this thesis is fairness. Since the sources of the unfair behaviors are the nondeterministic behavior of the flow of data in the circuit and the unrealistic behavior of components within the circuit, transition-based fairness assumptions are used to exclude unfair runs from consideration during model checking. We consider three types of fairness: unconditional, strong and weak, and show how to apply the fairness assumptions in BTSL without having to use the full-fledged BTSL* model checking procedure.

The current implementation of BTSL* and the best known model checking algorithm for DLTL are not yet of practical level. Nevertheless, the study of the BTSL* gives insight to how to combine BTSL with more expressive path formulas.

5.1 Future Work

There are a few directions to continue this research. The first one is to discover the equivalent expressive power of BTSL*. In particular, it is interesting to compare the expressiveness of BTSL* with ReCTL* [Cla05]. Obviously, all ReCTL* formulas are included into BTSL*, but for BTSL* formulas of the form $\varphi_1 \, \mathbf{U}^{\alpha} \varphi_2$ with $\varphi_1 \neq \mathit{true}$, it is not so clear whether equivalent ReCTL* formulas exist. The illustration of CTL* expressive power [MR99] may provide a good starting point.

The unpracticality of BTSL* derives from the Dynamic LTL model checking algorithm. This may be alleviated by developing an algorithm to build the generated NBA symbolically.

Last but not least, the implementation needs to be extended to allow reasoning on finite runs.

Bibliography

[ABBR04] Farhad Arbab, Christel Baier, Frank De Boer, and Jan Rutten. Models and temporal logics for timed component connectors. In *In Proc. SEFM04. IEEE CS*. Press, 2004.

[AHK02] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[AHS93] Farhad Arbab, Ivan Herman, and Pål Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5:23–70, 1993.

[AP98] Farhad Arbab and George A. Papadopoulos. Coordination Models and Languages. In *Advances in Computers*, pages 329–400. Academic Press, 1998.

[AR02] Farhad Arbab and Jan J. M. M. Rutten. A Coinductive Calculus of Component Connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer, 2002.

[Arb96] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*, pages 34–56, London, UK, 1996. Springer-Verlag.

[Arb04] Farhad Arbab. Reo: A Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004.

[BB00] Pearl Brereton and David Budgen. Component-based systems: A classification of issues. *Computer*, 33(11):54–62, 2000.

[BB07] Tobias Blechmann and Christel Baier. Checking Equivalence for Reo Networks. In *Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS 2007)*, 2007.

[BK96] Jan Bergstra and Paul Klint. The ToolBus Coordination Architecture. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061, pages 75–88, Cesena, Italy, 1996. Springer-Verlag, Berlin.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[BLS08] Andreas Bauer, Martin Leucker, and Christian Schallhart. The Good, the Bad, and the Ugly—But How Ugly Is Ugly? Technical Report TUM-I0803, TU München, 2008.

[BM90] Jean-Pierre Banâtre and Daniel Le Métayer. The GAMMA Model and Its Discipline of Programming. *Sci. Comput. Program.*, 15(1):55–77, 1990.

[BRKM91] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision

Diagrams. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 417–420, New York, NY, USA, 1991. ACM.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling Component Connectors in Reo by Constraint Automata. *Science of Computer Programming*, 61:75–113, 2006.

[BW96] Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[CCO+05] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Compututing Journal*, 17(4):461–483, 2005.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CES86] Edmund Clarke, E. Allen Emerson, and Aravinda Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CG92] Nicholas Carriero and David Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):96–107, 1992.

[CGP99] Edmund Clarke Jr., Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, Massachussets, first edition, 1999.

[Cla05] Dave Clarke. Reasoning about Connector Reconfiguration II: Basic reconfiguration Logic. In *In Proc. FSEN05, Teheran, Electronic Notes in Theoretical Computer Science*, 2005.

[DA04] Nikolay Diakov and Farhad Arbab. Compositional Construction of Web Services Using Reo. In *In Proc. International Workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS 2004*, pages 13–14. INSTICC Press, 2004.

[Dij71] Edsger W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Inf.*, 1:115–138, 1971.

[DS96] Graham Dean and Ian Sommerville. PCL: A Configuration Language for Modelling Evolving System Architectures. *Software Engineering Journal*, 11(2):111–121, 1996.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 27–39, 2003.

[FL79] Michael Fischer and Richard Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 1985.

[GM06] Laura Giordano and Alberto Martelli. Tableau-Based Automata Construction for Dynamic Linear Time Temporal Logic. *Annals of Mathematics and Artificial Intelligence*, 46(3):289–315, 2006.

[GPVW96] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

[HSW01] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ϵ-free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001.

[HT99] Jesper Gulmann Henriksen and P. S. Thiagarajan. Dynamic Linear Time Temporal Logic. *in Annals of Pure and Applied logic*, 96:1–3, 1999.

[KB07] Sascha Klüppelholz and Christel Baier. Symbolic Model Checking for Channel-based Component Connectors. In *Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2006)*, volume 175(2) of *Electronic Notes in Theoretical Computer Science*, pages 19–37, 21 June 2007.

[KB08] Sascha Klüppelholz and Christel Baier. Alternating-Time Stream Logic for Multi-agent Systems. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2008.

[Kle] Joachim Klein. Personal communication. http://www.ltl2dstar.de.

[Kwi89] M. Z. Kwiatkowska. Survey of fairness notions. *Inf. Softw. Technol.*, 31(7):371–386, 1989.

[LPS81] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 264–277, London, UK, 1981. Springer-Verlag.

[MC94] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26:87–119, 1994.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[MR99] Faron Moller and Alexander Rabinovich. On the Expressive Power of CTL*. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 360–369, Washington, DC, USA, 1999. IEEE Computer Society.

[MSA06] Mohammad Reza Mousavi, Marjan Sirjani, and Farhad Arbab. Formal Semantics and Analysis of Component Connectors in Reo. *Electr. Notes Theor. Comput. Sci.*, 154(1):83–99, 2006.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[Nis86] Nissim Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[Oss] Jörn Ossowski. JINC, a BDD Library. http://www.jossowski.de.

[PC08] José Proença and Dave Clarke. Coordination Models Orc and Reo Compared. *Electronic Notes in Theoretical Computer Science*, 194(4):57–76, 2008.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE, 1977.

[Rud93] Richard L. Ruddell. Dynamic Variable Reordering for Ordered Binary Decision Diagrams. In *ICCAD 1993*, pages 42–47, 1993.

[Smu68] Raymond Merrill Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, 1968.

[Tol98] Robert Tolksdorf. Laura: A Service-Based Coordination Language. *Science of Computer Programming*, 31, 1998.

[TVMS07] Samira Tasharofi, Mohsen Vakilian, Roshanak Zilouchian Moghaddam, and Marjan Sirjani. Modeling Web Service Interactions Using the Coordination Language Reo. In Marlon Dumas and Reiko Heckel, editors, *WS-FM*, volume 4937 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2007.

[VW86] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.